A Framework to Support Opportunistic Groups in Context-Aware Applications

CMU-HCII-16-101

May 2016

Adrian A. de Freitas

Human Computer Interaction Institute

Carnegie Mellon University

Pittsburgh, Pennsylvania 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This work was supported by the National Science Foundation under Grant No. CCF-1404698, the Software Engineering Institute, Carnegie Mellon University, the United States Air Force Academy Department of Computer Science, and the United States Air Force. The views and conclusions in this document are those of the author, and should not be interpreted as representing the official policies, expressed or otherwise, of the United States Air Force Academy, United States Air Force, or the United States Government.

KEYWORDS

Ubiquitous computing, context-aware computing, opportunistic groups, internet-of-things, software toolkits, frameworks, computer-supported cooperative work, interaction techniques, networking.

ABSTRACT

Context-aware computing utilizes information about users and/or their environments in order to provide relevant information and services. To date, however, most context-aware applications only take advantage of contexts that can either be produced on the device they are running on, or on external devices that are known beforehand. While there are many application domains where sharing context is useful and/or necessary, creating these applications is currently difficult because there is no easy way for devices to share information without 1) explicitly directing them to do so, or 2) through some form of advanced user coordination (*e.g.,* sharing credentials and/or IP addresses, installing and running the same software). This makes these techniques useful when the need to share context is known *a priori*, but impractical for the one time, opportunistic encounters which make up the majority of users' lives.

To address this problem, this thesis presents the Group Context Framework (GCF), a software framework that allows devices to form groups and share context with minimal prior coordination. GCF lets devices openly discover and request context from each other. The framework then lets devices intelligently and autonomously forms *opportunistic groups* and work together without requiring either the application developer or the user to know of these devices beforehand. GCF supports use cases where devices only need to share information once or spontaneously. Additionally, the framework provides standardized mechanisms for applications to collect, store, and share context. This lets devices form groups and work together, even when they are performing logically separate tasks (*i.e.,* running different applications).

Through the development of GCF, this thesis identifies the conceptual and software abstractions needed to support opportunistic groups in context-aware applications. As part of our design process, we looked at current context-sharing applications, systems, and frameworks, and developed a conceptual model that identifies the most common conditions that cause users/devices to form a group. We then created a framework that supports grouping across this entire model. Through the creation of four prototype systems, we show how the ability to form opportunistic groups of devices can increase users and devices' access to timely information and services. Finally, we had 20 developers evaluate GCF, and verified that the framework supports a wide range of existing and novel use cases. Collectively, this thesis demonstrates the utility of opportunistic groups in context-aware computing, and highlights the critical challenges that need to be addressed to make opportunistic context sharing both practical and usable in real-world settings.

The contributions of this thesis are:

1. A conceptual model, based on an analysis of prior literature, which describes the conditions under which users and/or devices form and work in groups.
2. An implementation of the Group Context Framework, which highlights the software abstractions and architecture needed to support all of the group types identified in our conceptual model.
3. A demonstration of the value of opportunistic groups in context aware computing, through the creation of four major systems and numerous smaller applications.
4. A validation of GCF's robustness, through an examination of 65 ideas submitted by 20 developers.
5. An examination of the challenges associated with utilizing opportunistic groups in context-aware applications, based on our own experiences using GCF, as well as from issues raised by developers from academia and industry.

Acknowledgments

I would like to take this opportunity to thank everyone who helped me along this short but arduous journey. There were many, *many* times throughout this process that I felt alone, and uncertain as to whether or not my research was on the right track. Yet looking back, I'm amazed at how many people were willing to put aside time to offer me their encouragement or give advice. It's an incredibly humbling realization, and I think it is fair to say that I wouldn't have made it to this point without all of your support.

Before getting started, I want to apologize to everyone whose name I will undoubtedly omit. I promise that it is not intentional, or that your assistance was somehow less valuable than others. It just happened to be that I ended up needing a lot of help, and I'm not certain if I could remember everyone who helped me even if I had another three years to do it.

First and foremost, I would like to thank my wife, Michelle, who has been there for me since I was 17, and who is the only one who knows how truly demanding this was for me on both a mental and physical level. I can't count all the ways you have sacrificed these past three years to give me enough time to do my research and write my dissertation. But now that the dissertation is for the most part written, I just wanted to remind you that 1) yes, you still have a husband, and 2) yes, I will finally take out the garbage.

Also, special thanks to my son, Liam, for being the bright spot in my darkest of days. Even after having all of my papers rejected to a conference (which occurred more than once), I could always depend on you to give me a big smile and hug as soon as I got home. You are too young to understand today, but I hope that one day I can be as good a dad as you think I am.

To my parents, Wagner and Chong Sun, and sister, Genie, thank you for continually pushing me to be more than I am. We may never be in the same state thanks to the military, but I want you to know just how much I depend on your love and support.

Additionally, I would be remiss if I didn't thank all of my four-legged "research staff" (Rocky, Angel, Mia, Buzz, Neil, and Allie) for their silent, but never-ending encouragement. Yes, I realize that I am thanking my *pets*, but they were the only ones who ever stayed up with me at 3am, so I figure that they are worth mentioning.

To everyone in the Ubicomp Lab (past and present), thank you for the many entertaining conversations we have had over these past several years! Special thanks to Jung Wook Park, Julian Ramos, and Nikola Banovic for helping me recover my dissertation during the "great crash of February 29, 2016." Without your cool heads and fast reflexes, I would have had an even bigger mental breakdown than I already did. Also, I would like to thank Christian Kohler, Christine Bauer, and Alaaeddine Yousfi for listening to all of my gripes and/or complaints, and Sunyoung Cho for giving me the opportunity to work with you on SpaceGuide. I am continually amazed by all of the great things that are going on in the lab, and I hope that we continue to work on new and exciting projects in the future.

To my various interns (Junrui "Jackie" Yang, Akshaye Shreenithi Kirupa Karthikeyan Ranithangam, Barun Kwak, and Brandon Lee), thank you for all of your hard work. Without you, this thesis would probably have had considerably fewer demo applications than it did. I am grateful that you were willing to put in the time and effort to use GCF, and I wish you all the best in your future academic endeavors.

To Xiang 'Anthony' Chen, thank you for deciding to collaborate with me on the Qualcomm Fellowship. Your research was one of the reasons why I decided to come to CMU, and while we didn't progress past the finalist stage, I was truly honored to get the opportunity to know and work with you.

To Nikola Banovic and Annie Malhotra, thank you for letting Michelle, Liam, and me become a part of your personal lives. We truly enjoyed our excursions throughout Pittsburgh (*especially* at the Salt and Pepper Bakery), and for all of those cups of Turkish coffee that helped keep us aware until the early hours of the morning. I still think you should name your child "Lazar Dragan," (seriously, it's a *genuine* Serbian name) but regardless of what name you choose, we know that you will be incredible parents.

To Queenie Kravitz, I don't know if there is anything I can say that accurately describe how grateful I am for all of your encouragement and support these three years. So all I will say is "thank you from the bottom of my heart", and hope that you understand the level of gratitude that I am trying to convey.

To my "cheerleader," Michael Nebeling, thank you for going out of your way to collaborate with me on my research these past one and a half years. I have learned so much about the research process by working with you, and I thank you for your continual encouragement and lunchtime conversations. I wish you the best in your new job at Michigan, and hope that we will continue to find ways to collaborate in the future.

And finally, thank you Anind for having faith in me when I never did. As the "second biggest professional risk" of your career, I hope that your experience working with me has been as worthwhile as it has been for me. I don't know if you will ever take the time to read this section, but I hope you know just how much I came to depend on your leadership, guidance, and mentorship these past three years. In many ways, you were the mentor I always wish that I had had during my military career, and I sincerely hope that our brief time at CMU only signals the start of things to come.

Table of Contents

1. Introduction and Motivation

When Mark Weiser discussed the idea of context-aware computing in "A Computer for the 21st Century", he described a future in which computers would be able to dynamically modify their behavior in response to users and their environment [120]. Since then, we have made considerable progress towards achieving this grand vision. Digital assistants such as Google Now and Siri have demonstrated how knowing even a small amount of information about a user (*e.g.,* her location, activity) can significantly improve a system's ability to provide them with the right information at the right time. Meanwhile, the rapid proliferation of mobile/wearable devices and cheap sensor platforms (*e.g.,* GPS, accelerometers, heartbeat sensors) has dramatically increased both the quantity and types of contextual information that are widely available, and have created new opportunities for applications to monitor the user's state and respond accordingly.

Together, these factors have fundamentally redefined our expectations as to how computers should behave and operate. In the early days of computing, users were expected to manually input all of their information into their applications in order to gain access to their services. Today, however, there are many types of information that computers are expected to "just know" on their own. When we turn on a map application, we expect it to automatically zoom and center the map on our current location. Likewise, when we travel to a foreign country, we expect our smartphone will automatically update its internal clock to match the local time. While these examples are simplistic, they highlight the subtle but important role that context plays in our everyday computing experiences. When viewed in this light, context-awareness is no longer a "nice to have" feature. Instead, it is a fundamental building block for modern software systems, and a vital technique for developers to leverage when creating their applications.

Although prior research has shown the value of context-awareness in a wide range of application domains (*e.g.,* navigation systems, intelligent tutors [60], health care [12], Internet of Things [54,91]), our ability to create context-aware systems is still largely constrained by current technology. To date, most context-aware systems only take advantage of contexts that are either produced on the device(s) they are running on, or from external sources that are known *a priori* (*e.g.,* sensors in an environment that we frequently visit). This is because sharing context is still nontrivial, and requires some form of explicit coordination (*e.g.,* pairing devices, downloading and running the same application, trading credentials) between the device(s) that are producing context and the device(s) that consume it. While this one-time setup is practical when the need to share context is 1) known well in advance, and/or 2) occurs repeatedly, it is cumbersome when devices only need to share context once or spontaneously. As a result, researchers and developers oftentimes avoid creating applications that rely on one-time exchanges of contextual information, as the effort required to implement this functionality currently outweighs its benefits.

Nevertheless, there are many situations in which being able to opportunistically share context would be advantageous. For example, if a group of smartphones are physically near one another and are running a navigation application, it makes sense for them to be able to conserve resources by taking turns running their GPS and sharing the coordinates with each other. Similarly, if a group of co-workers just happen to meet in a break area and want to schedule a meeting, it makes sense for their devices to be able to quickly share their work calendars with each other so that they can determine the best time to meet.

At first glance, these examples seem too coincidental and spontaneous to warrant special consideration. It is important to realize, however, that opportunistic groupings such as these make up the vast majority of users' interactions with the physical world. It is not uncommon for people to engage in spur-of-the-moment interactions, such as asking a stranger for directions or receiving one-time advice from a friend or colleague, without explicitly planning for these exchanges in advance. Yet while people have an innate ability to take advantage of opportunistic encounters to share information and collaborate with others, our devices lack the ability to do the same. As long as this is the case, there will always be a logical separation between the way *we* interact with their environments and the way that *our devices* do. This in turn limits the types of interactions that these devices can naturally and seamlessly support.

In this thesis, we aim to extend the reach of context-aware computing by allowing devices to easily share context with each other, even when they are interacting with each other for the first time. To accomplish this, we have developed the Group Context Framework (GCF), a novel software framework that allows devices to opportunistically detect one another, form groups, and share context with minimal prior coordination. GCF consists of two main software abstractions. The first is an architecture that allows devices to collect, store, and share context in an application independent manner. The second is a robust communications suite that allows devices to openly request and receive context without requiring *a priori* knowledge of one another. By packaging these capabilities into a standalone middleware, GCF makes it easier for developers to utilize opportunistic sources of context in their applications. This increases the number and types of context that can be easily collected and shared, and expands the range of context-aware applications that can be practically created and deployed.

This thesis makes contributions to the fields of 1) mobile and ubiquitous computing, and 2) context-aware computing. Through an extensive literature review, we have examined a wide range of context-sharing applications, systems, and frameworks, and developed a conceptual model that describes the conditions that cause users and/or devices to form and operate in groups. We then used this model to create a framework that can intelligently and autonomously form opportunistic groups under the widest conceivable range of use cases. Through the creation of three functional prototypes, we evaluate GCF's robustness, and conduct a series of focused explorations to identify 1) the types of context-aware applications that can be easily created when devices can share context at will, and 2) the capabilities needed to support these interactions through software. We then conduct a postmortem analysis of these systems, and collect developer feedback to identify the challenges of using opportunistic groups from a developer, user, and system perspective. Collectively, this work demonstrates how opportunistic groups can be used to expand the application space for context-aware computing, and the potential dangers of doing so. Additionally, this work provides developers with a robust and flexible architecture to allow them to easily incorporate this functionality within their own applications.

Having motivated the importance of supporting opportunistic groups in context-aware computing, the rest of this chapter is organized as follows. In the next section, we provide operational definitions of the terms *context-aware computing* and *opportunistic group* as they pertain to this thesis. Afterwards, we describe the challenges in using opportunistic groups in context-aware computing, and motivate the need for a framework that can support grouping and context sharing across the widest conceivable range of use cases. We then conclude this chapter with a discussion of our research questions and expected contributions.

1.1. What is Context-Aware Computing?

What does it mean for computer system to be "context-aware?" Surprisingly, there is not a straightforward answer to this question. Although context-aware computing has been studied as early as 1992 with the Active Badge system [116], the term itself is still nebulous, and has been interpreted and redefined a number of time over the years [91]. Thus in order to ground this our work, we examine past efforts at defining context and context-awareness, and select an operational definition of each.

There are two key challenges in developing a meaningful definition of context-awareness. The first is deciding what qualifies as context. Although researchers oftentimes have a general intuition as to what context is, attempts at defining it regularly devolve into a list of representative examples. Schilit and Theimer, for example, define context as location, nearby people/objects, and changes to those objects over time [104]. An alternative definition, provided by Brown *et al*. and Ryan *et al.,* includes many of these dimensions, but also takes the user/device's orientation [13], as well as their identity [99] into account. In Hull *et al*., context is defined as a summation of the user's situation, such as his/her personal health, identity, companions, and available computing resources [59]. Finally, in Dey, context is viewed as a summation of the user's emotional state, focus of attention, location/orientation, date/time, and nearby objects/people [30]. While this work provides concrete examples of contextual information, none are all encompassing. Consequently, as noted by Dey, these definitions show us what context has been in the past, but do not provide general criteria for classifying new types of information.

In contrast with the "definition by example" approach, other work has attempted to define context using broader generalizations. Franklin and Flachsbart, for instance, define context as being the situation of the user, without specifically mentioning what types of information that entails [42]. Meanwhile, definitions provided by Ward *et al.* and Rodden *et al.* describe context as an aspect of an application's surroundings and setting, respectively [98,118]. In order to provide a more structured definition, Abowd and Mynatt describe context in terms of the "five W's" (*i.e.,* who, what, where, when, why), and claim that context is any piece of information that addresses one or more of these questions [2]. More recently, Anh and Kim classified context as a set of interrelated events with logical and timing relations [3]. This definition expands upon previous work by looking at context as a combination of discrete and continuous events. Although these definitions provide a wider lens for looking at context, they suffer from being *too* general. In other words, they provide a high-level understanding what context is, but are too broad to be useful as a classification tool.

In this thesis, we do not develop our own definition of context. Instead, we rely on one that has become widely accepted and referenced by the academic community. According to Dey [30]:

> *"**Context** is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves."*

We use this definition of context because it is both specific and generalizable. While Dey *et al.* specifically mentions four essential categories of contextual information (identity, location, activity, time) that are useful in a wide range of applications, this definition acknowledges that these categories are not all encompassing, and that additional types of information may be considered context depending upon the particular application being built. Additionally, in contrast with other definitions that try to describe context in terms of the user [42] or application [98,118], Dey *et al.'s* definition takes both types of entities into consider. Finally, this definition acknowledges that the sources of context are diverse, and can be either implicit (*i.e.,* detected by sensors on a device or in the environment) or explicit (*i.e.,* manually input by the user through an appropriate interface). Since one of the primary contributions of this thesis is to support opportunistic context sharing at the middleware level, the points raised by Dey's definition have largely influenced the overall design of our Group Context Framework. They emphasize the need for systems that can collect and disseminate a wide range of contextual information, regardless as to what that information is, what it describes, or how it is obtained.

The second challenge in defining context-awareness is determining what action(s) an application needs to perform for it to be considered aware. Over the years, researchers have attempted to address this issue by establishing taxonomies of context-aware features/capabilities. Schilit *et al.* defined four broad categories of contextual applications: 1) applications that modify the appearance objects on a user interface based on their proximity (proximate selection); 2) applications that add, remove, or modify UI components based on the user's current location (automatic contextual reconfiguration); 3) applications that perform different actions, or produce different output, depending upon the conditions under which these actions are invoked (contextual information and commands); and 4) applications that perform a specific set of actions once a set of contextual criteria is met (context-triggered actions) [103]. Alternatively, Pascoe proposed his own taxonomy in which he describes the *features* of context-aware application. In his taxonomy, applications are context-aware if they can sense the environment, react based on their surroundings, locate and interact with relevant (external) resources, and augment the user experience by providing timely and pertinent information [89]. Lastly, Chen and Kotz provide their own interpretation of context-awareness, in which they differentiate between applications that automatically adapt to discovered context (active context awareness) and applications that merely present context for users to view/interpret later (passive context awareness) [19].

While these taxonomies do a good job of highlighting examples of context-awareness, their focus on categorizing existing (at the time) systems makes them difficult to apply to new types of applications. To address this limitation, Dey *et al.* distills the notion of context-awareness into a simple definition:

"A system is ***context-aware*** *if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task."*

Similar to before, we use this definition because it provides an effective balance between simplicity and expressiveness. There are two aspects of Dey's definition that are particularly relevant to this thesis. First and foremost, this definition does not draw a distinction between context-aware services and information, but instead acknowledges that there are occasions where either can be useful. Secondly, whereas Schilit *et al.*, Pascoe, and Chen and Kotz's definitions focus on defining context-awareness with respect to mobile devices, Dey's definition takes a more holistic approach, and talks about context-awareness with regards to *systems*. From a framework development perspective, this is important, as it shows that context-awareness is not exclusive to a single platform, but a technique that is applicable to (and hence must be supported across) a broad range of devices and form factors.

Through this discussion, we have described previous efforts to understand both context and context-awareness, and have chosen an operational definition of each. These definitions provide us with a foundational understanding of both the types of contextual information that are relevant to an application, and the various ways in which this information can be used to create an improved user experience. This in turn has allowed us to make more informed decisions about what use cases we wanted to support when creating the Group Context Framework.

1.2. What is an Opportunistic Group?

Similar to context, the term *opportunistic group* is frequently mentioned in the academic community, but rarely defined. Many researchers have a general sense of what it means for a group to be opportunistic, but this rarely evolves beyond an "I know it when I see it" mentality. Thus, similar to the previous section, an operational definition is needed.

Prior work in opportunistic groups has primarily focused on increasing social interactions between users. In [60], Ikeda and Mizoguchi developed a conceptual model to help computer-based tutor systems determine when a student would best benefit from working with others. Their model introduced an *opportunistic group formation function* that determined the best time for students to transition between individual and collaborative learning exercises. The function then assigned each student to a group, and provided them with personalized learning goals and social roles that conformed to the group's overall learning objectives. In [41], Ferscha *et al.* created a group collaboration tool that formed *dynamic* groups whenever more than 50% of the members of a (predefined) group assembled in a designated meeting area. Their system could then alert the remaining members that a meeting was underway. In the Social Serendipity project [34], Eagle and Pentland developed a social networking platform that allowed users to register their interests/hobbies, as well as their smartphone's Bluetooth ID, with a centralized database. Users could then run a dedicated application on their phone to scan for nearby Bluetooth devices, thereby allowing them to discover when they were in proximity of other users with similar interests. More recently, Marcu *et al.* looked at groupings of educational faculty and/or administrative staff in order to support opportunistic sensemaking of students with special needs [77]. Their system, Lilypad, let staff members track student behavior and seamlessly share this information with each other, thus helping them generate organizational knowledge and make more effective decisions.

While this work demonstrates the usefulness of opportunistic groupings, they exclusively focus on groups of users (*e.g.,* students, faculty). We believe, however, that the notion of an opportunistic group applies to devices as well as users. Although work performed by Wang *et al.* differentiates between user and device groups, it only looks at groupings of devices that directly support a user level task (*e.g.,* linking devices because their respective users are all participating in the same meeting) [114]. In contrast, our interpretation of device groups is more broadly encompassing, and includes groupings that directly support the user, as well as groupings that occur at the system level (*e.g.,* linking devices in order to share resources and conserve battery life). By minimizing the amount of coordination and effort required for devices to form groups, we aim to increase the types of interactions that they can seamlessly take part in. This allows them to work together in support of user *and/or* system level goals.

In order to illustrate the importance and relevance of opportunistic groupings of devices in context-aware computing, consider the following inspirational scenarios:

Scenario 1: *Jack is riding a GPS-equipped bus and tracking his location on his smartphone. The phone detects the bus' ability to track its own location, and automatically forms a group with the vehicle so that it can access the bus' GPS data and conserve its own battery life.*

Scenario 2: *Michelle, Shannon, and Monique cross paths in a common area and engage in a lengthy conversation. Afterwards, the three decide that they would like to meet at a later date, and open up their personal calendar applications on their respective smartphones. The phones, sensing the common goal of scheduling a meeting, temporarily form a group, and present the users with a list of times when they are all available.*

These scenarios represent the types of serendipitous interactions we want to enable in this thesis. In each scenario, the devices involved have little to no prior knowledge of one another. Yet in both cases, they are able to dynamically discover each other and determine that they can assist one another. They then form a group so that they can share information/services.

There are two important characteristics of an opportunistic group that are highlighted through our example scenarios. The first is that the decision to form an opportunistic group oftentimes occurs without warning. In scenario 1, for example, Jack's smartphone has no way of determining that it can utilize the bus' GPS capabilities until the user boards the vehicle. Similarly, since the conversation involving Michelle, Shannon, and Monique in scenario 2 was unplanned, their smartphones did not have time to exchange calendar information or pair beforehand. As a result, the devices had to wait until their respective users were trying to schedule a joint meeting before the decision to form a group could be made.

Secondly, while the groupings described through these scenarios are obviously useful, they are also highly situational. For example, the opportunistic group formed between Jack's phone and the bus is only meaningful for as long as Jack remains on the bus. Likewise, the grouping between Michelle, Shannon, and Monique's smartphones should only last as long as is needed to identify a good time to meet again. In both cases, there is a relatively small window of time in which the group formed by these devices is both necessary and useful. Thus, in order to take advantage of an opportunistic group, the mechanism for finding and recruiting members must be generalizable and streamlined so that the cost (*e.g.,* time, effort) of forming the group does not outweigh the benefits.

Given these characteristics, we have developed the following definition of an opportunistic group:

Opportunistic Group: *"A spontaneous assemblage of two or more entities (*e.g.*, device(s), application(s), user(s)) created for the purpose of exchanging information and/or services."*

We have chosen a broad definition of opportunistic groups in order to highlight their unpredictable and ephemeral nature. There are three important aspects of this definition that are worth noting. First, our definition does not consider groupings of users, devices, and applications in isolation, but instead acknowledges that the composition of a group is highly flexible and can vary widely depending upon the situation. Second, as the motivational scenarios above show, our definition of *device* is broad, and includes both artifacts that users regularly interact with (*e.g.,* smartphones), as well as those that are embedded in the environment, and may not be directly visible (*e.g.,* buses, sensors deployed throughout a building, software services). Finally, our definition does not require opportunistic groupings to be user-initiated. On the contrary, while there are many situations in which users might want to form an opportunistic group (*e.g.,* sharing calendar information), there are also occasions where *devices* might need to form groups on the user's behalf (*e.g.,* borrowing GPS data from a vehicle's on-board sensor in order to extend one device's battery life). This latter point contrasts with our user-centric notion of groups, and highlights the need for opportunistic groups to be given special consideration.

1.3. Why are Opportunistic Groups Difficult to Use?

The example scenarios described above illustrate the potential benefits of allowing devices to form opportunistic groups and share context at will. To date, however, developers have largely avoided taking advantage of opportunistic groupings when creating their applications. While there are some systems that allow devices to dynamically work together, they focus on specific tasks, such as allowing devices to pass messages [11,68], conserve resources [58,64,71,110], or improving users' situational awareness [33,49]. This emphasis on narrow use cases limits their generalizability, and makes them poorly suited to support the wide variety of one-time, spontaneous interactions that we hope to enable through our work.

In order to better understand why opportunistic groups are rarely used in context-aware applications, we have reexamined our two motivational scenarios. In doing so, we find that many of the properties that make opportunistic groups unique and compelling from both a system and end-user experience standpoint also make them difficult to support using existing technologies and techniques. Specifically, there are four core challenges to utilizing opportunistic groups in context-aware applications:

1. **Grouping Opportunities are Spontaneous and Unpredictable.** Opportunistic groups, by their very nature, can occur at any place and time. Yet while users are able to fluidly react to their environments in order to take advantage of these serendipitous exchanges, applications are more rigidly defined, and must be explicitly told how they will act in response to each set of conditions. Since there is no simple way to enumerate all of the situations that a device might benefit from by forming a group, many developers tend to only focus on groupings that they know are either 1) critical for the application to work as intended, or 2) the easiest to predict in advance. This simplifies the process of adding grouping functionality to applications, but severely limits their ability to respond to new or unexpected situations.
2. **High Interaction Cost**. Current systems that allow for *ad hoc* device groupings are cumbersome from a user experience standpoint. For example, technologies such as Bluetooth and Wi-Fi direct are already pervasive, and technically allow devices to freely share information with one another without the need for *a priori* coordination. However, both require the user to undergo a lengthy pairing process (*e.g.,* sharing codes, passwords) before they can be used. Meanwhile, while some applications allow users to easily share information with one another without having to explicitly pair (*e.g.,* Bump [148], Android Beam, AirDrop), they are only useful if every user has the same application installed and running. In both cases, there is an initial cost (in terms of time and effort) that both the initiator of the group and the group member must be willing to pay before they can take advantage of a grouping situation. Users are oftentimes willing to take advantage of a grouping opportunity when they already have the tools to do so. If additional effort is required, however, users will view the cost-of-entry as being too high, and will forgo grouping altogether.
3. **Grouping Opportunities are Not Always Apparent/Visible to the User**. Users are not always aware of the grouping opportunities that surround them. In scenario 1, for example, the user may not be aware that the bus he is riding in has GPS tracking capabilities. Yet by forming an opportunistic group, his phone is able to utilize this sensor instead of its own and conserve battery life. As we continue to transition towards an Internet of Things, serendipitous groupings of devices such as these will become more pervasive, and it will become increasingly difficult for users to be expected to manage (or be aware of) these opportunities on their own. Consequently, there is an increasing need for devices to be able exercise some level of autonomy when deciding which groups (if any) to participate in. This way, the user can benefit from these groupings without having to go out of their way to setup and maintain them.
4. **Lack of Critical Mass**. In order for opportunistic groupings to be viable, there needs to be a single, standardized way for devices to form and work in groups. This problem is difficult to address at the application level, however, since developers have traditionally lacked the sheer influence needed to establish such a standard on their own. While developers and researchers have implemented a wide range of mechanisms for finding and forming groups, these systems all assume that grouping will only occur between devices that are running the exact same application at the same time. Yet most applications lack the critical mass needed for them to take advantage of opportunistic groupings except under carefully controlled conditions. As a result, developers do not seriously consider adding group functionality to their applications, as the cost of doing so seems large in comparison to the number of times that it will actually be used in real-world settings.

Although these challenges are significant, it is important to remember that the majority of our interactions with the world are unplanned. While we oftentimes have a general sense as to who or what we might do during the course of any given day, we regularly modify these plans in order to accommodate new, or spontaneous situations. Consequently, our devices need to do the same. On their own, a single opportunistic encounter between two or more devices may seem unimportant and impractical to support *via* software. Collectively, however, these serendipitous exchanges represent a vast range of interactions that has, to date, been relatively unexplored. By giving devices the ability to detect and form groups, we can allow them to take advantage of fortuitous encounters with other devices.

This allows them to leverage these encounters in order to gain access to important and relevant information and services, without requiring the user to manually enable these interactions.

The goal of this thesis is to create a general-purpose framework that lets devices autonomously form groups as needed. To achieve this, we have examined a wide range of systems that have been created to allow devices to share context, and have developed a conceptual model that describes the conditions that typically lead to group formation (see CHAPTER 2 for a more detailed account of these systems). We then created the Group Context Framework (GCF) to support the full range of group interactions. Using GCF, developers can choose to opportunistically interact with a specific device, or with any device(s) that satisfy a set of conditions. Additionally, developers will be able to easily extend our framework to define their own group selection strategy/criteria. Thus our system can support a wider range of use cases than is possible using other systems.

Our work with GCF will address the above challenges in the following ways:

1. **Supporting Spontaneous Groupings.** Our framework will let devices continuously scan their environments in search of grouping opportunities. This will allow it to form groups with other users or devices regardless if they have met before.
2. **Making One Time Groupings Practical.** By allowing devices to automatically form groups on the user's behalf, our framework will make it feasible for them to collaborate for short periods of time. This removes a significant barrier to forming a group, as it allows users to take advantage of grouping opportunities without requiring them (in many, but admittedly not all cases) to explicitly pair the devices.
3. **Making Grouping Transparent to the User**. By allowing developers to define the (general) conditions for grouping, GCF-enabled applications can take advantage of groupings that are both visible and hidden to the user. This increases the range of interactions that the device can participate in.
4. **Circumventing the Need for Critical Mass**. In contrast with current context-sharing systems, which are intended to facilitate collaborations on a per-application basis, GCF is specifically designed to support groupings of devices *across* entities. By providing a general-purpose middleware, our framework establishes a standardized means for representing and requesting contextual information. This allows GCF-enabled devices to provide context to each other when performing logically separate tasks, thus allowing developers to take advantage of opportunistic groupings even when their respective applications have not achieved critical mass on their own.

Developers will be able to utilize GCF in a variety of different ways. Although we primarily intend for GCF to support context sharing between different types of devices, the framework also provides developers with a simple and efficient way to access contextual information on the local device. Initially, we envision that developers will utilize GCF to access a single device's context. As more developers adopt the framework, however, the opportunities for groups of devices to serendipitously work together will increase. Developers will then be able to take advantage of these groupings without having to rewrite their code.

Furthermore, by making it easier to find, form, and use opportunistic groups, we aim to significantly increase the types of context-aware applications that can easily be created. Since our framework handles all of the problems associated with detecting devices and forming groups, developers are able to focus more of their attention on the types of interactions they would like to enable rather than on the low level mechanics. This simplifies the development process, and makes it easier to create and explore the unique types of context-aware applications that are possible through opportunistic groupings.

1.4. Research Questions

This thesis addresses the following research questions:

Research Question #1: How can we allow devices to opportunistically form groups and share context?

Our first research question examines the technical challenges associated with allowing devices to opportunistically form groups and share context. As mentioned previously, devices do not currently share information unless they are explicitly direct to do so. Given the rapid rate at which technology is spreading, however, it is becoming clear that this strategy is untenable, and that there needs to be mechanisms in place to let devices intelligently decide if and when they should work together. This would allow devices to easily take advantage of the information and services that surrounds them, without requiring developers to anticipate, or explicitly program, all of these exchanges in advance.

This thesis examines this problem from both a conceptual and engineering standpoint. Before we can build a system that allows devices to automatically form groups, we first need to know *how* and *why* device form groups in the first place. To develop this intuition, we examined twenty-nine different context-sharing systems, and the use cases they support. We then developed a conceptual model that describes the most common settings that cause devices to have to form groups and share information. Through this model, we find that currently context-sharing systems are currently focus on a specific task or group type. This allows these systems to excel for their intended purpose, but prevents them from providing a general purpose solution.

Using our conceptual model as a guide, we have identified three critical technical requirements that need to be satisfied to allow devices to form opportunistic groups. First, there needs to be a standardized mechanism for both requesting and representing contextual information that works across devices and platforms, regardless if devices are performing the same task or running the same application. Second, devices need to be able to freely communicate with one another, even when they are not connected to the same network or broadcast domain. Finally, there needs to be a way for devices to decide which groupings are relevant at a given time, and be able to take action without having to ask the user for permission.

In this thesis, we are interested in addressing these requirements at the middleware level. By creating the Group Context Framework, we will provide a standardized way for application developers to represent and share contextual information, while still giving users control over what information they are willing to share. Using GCF, developers can specify 1) the context(s) that they can provide, 2) the context(s) that they need, and 3) the general strategy for forming groups. The framework will then continuously and autonomously search for and form the appropriate group structures in order to satisfy these needs. Through these combined capabilities, we will give devices the ability to take advantage of grouping opportunities that are spontaneous and unpredictable. This reduces the cost of forming a group, and allows devices to take advantage of a wider range of groups without requiring every device to have the same app installed and running at all times.

Research Question #2: How does the ability to form opportunistic groups increase the range of context-aware applications that can be practically created?

The second research question focuses on identifying context-aware applications where opportunistic grouping and context sharing is both useful and/or necessary. In the past, this type of exploration has been difficult to perform since it required developers to create their own technology stack to allow devices to exchange information/services. With GCF, however, this functionality is now provided "for free." Developers may initially be put off by the thought of not having explicit control over how their applications interact with one another. Yet through our work, we aim to show how these abstractions actually *increase* the range of context-aware applications that can be easily created.

To investigate this research question, we have developed four systems that either directly use, or are built on GCF's core technology:

- In *Didja*, devices share multiple streams of sensor data with each other, and compare these readings in order to determine if they are likely experiencing the same contextual state (*e.g.,* riding the same bus, hearing the same conversation). This allows Didja to infer when devices are involved in the same activity, thereby allowing them to take advantage of precise grouping opportunities that are too nuanced to be detected *via* Bluetooth alone.
- In *Snap-To-It*, we created a universal interaction tool that lets users "select" and control nearby physical appliances (*e.g.,* printers, digital projectors) and/or objects by taking a photograph of them with their smartphone. It demonstrates how the ability to form opportunistic groups on command increases users' abilities and willingness to interact with the ubiquitously distributed devices in their environments.
- Our third system, *Impromptu*, is a "just in time" application delivery platform that provides users with contextually relevant applications/services . Here, apps (running in the cloud) request context from users, and offer their services when they are relevant. Users can then use GCF to form an opportunistic group with the app and receive custom interface(s) without having to install any additional software.
- Our fourth system, *Bluewave*, lets devices openly advertise and share context by programmatically manipulating their Bluetooth name. Bluewave lets users and/or devices broadcast small amounts of context with their immediate environment, and allows GCF to detect and form opportunistic groups in a wider range of real world environments.

By focusing on multi-purpose *systems* as opposed to single-use applications, we are able to explore a wide range of applications that make use of opportunistic grouping and context sharing in new and compelling ways. These systems also provided us with real-world experience using GCF, and allowed us to add functionality as needed. For example, our work with Snap-To-It demonstrated that GCF needed to let devices share arbitrary messages with each other (in addition to context) once a group is formed. Similarly, our work with Bluewave was motivated by the realization that there are many real-world situations where devices need to form an opportunistic group, but are not connected to the same network; by adding Bluewave's functionality to GCF, our framework now lets devices broadcast information using their device's Bluetooth name, eliminating the need for a direct connection in many situations.

Our goal in this thesis is not to enumerate *all* of the ways opportunistic groups can be utilized in context-aware computing, but rather to highlight specific use cases where doing so is valuable from both an end-user and system standpoint. Through this work, we will show how opportunistic groupings can improve users' access to timely information and services, and how GCF directly supports the creation of these applications. This will demonstrate how developers can use our framework, and provide a foundation for future research.

Research Question #3: What are the challenges associated with utilizing opportunistic groups in context-aware computing?

Our third research question examines the challenges associated with combining opportunistic device groups with context-aware computing. While our work with GCF identifies many application domains that benefit from opportunistic context sharing, it also illustrates the potential problems that can arise when devices are able to form groups and share information at their own discretion. Consequently, it is important to enumerate these challenges so that developers can take them into account when creating their own applications.

Our exploration of this question will occur in two parts. First, we will conduct a post-mortem examination of the applications developed developed or proposed in thesis, and identify the specific challenges with gathering and using opportunistic context at the developer, user, and technical level. Some of the challenges that we will highlight as part of this analysis include:

- Developer Challenges
    - **Context Availability.** What types of context need to be widely obtainable *via* GCF in order to support and enable a diverse set of opportunistic context-aware applications?
    - **Developer Tools.** What types of tools need to be provided in order to make it easy for developers to take advantage of the information and capabilities offered by our system?
    - **Trust.** What mechanisms need to be put into place in order to allow devices to trust that the information provided by other devices is accurate and truthful?
- User Challenges
    - **Privacy.** What information are users comfortable sharing in an opportunistic manner?
    - **Security.** How can GCF protect users against malicious applications/users?
- Technical Challenges
    - **Impact on Battery Life.** What is the impact of opportunistic grouping on a mobile device's battery life? Are there ways to mitigate this impact?
    - **Scalability.** How well does our system scale in an environment with dozens, or even hundreds of devices?
    - **Networking.** How do current networking and communication technologies limit the types of groups that can be practically formed using the framework?

Additionally, we have also solicited the aid of 20 context-aware application developers from both the academic and commercial sectors to examine our system, and brainstorm the types of context-aware applications that they would create with it. In all, we collected 65 application ideas, and analyzed them to see 1) whether or not our system can support these applications, and 2) if there are any additional challenges that we have not yet encountered through our own work.

Through this process, we aim to provide a comprehensive understanding of the benefits and potential pitfalls associated with utilizing opportunistic groups of devices. This knowledge, combined with our efforts at exploring the application space of opportunistic context-aware systems, will help to identify the technical and social boundaries of opportunistic context sharing, and will allow developers and researchers to be better informed when developing future generations of context-aware applications.

1.5. Expected Contributions

This thesis offers the following contributions:

1. A conceptual model, based on an analysis of prior literature, which describes the conditions under which users and/or devices form and work in groups.
2. An implementation of the Group Context Framework, which highlights the software abstractions and architecture needed to support all of the group types identified in our conceptual model.
3. A demonstration of the value of opportunistic groups in context-aware computing, through the creation of four major systems and numerous smaller applications.
4. A validation of GCF's robustness, through an examination of 65 ideas submitted by 20 developers.
5. An examination of the challenges associated with utilizing opportunistic groups in context-aware applications, based on our own experiences using GCF, as well as from issues raised by developers from academia and industry.

Our first contribution is intellectual. By providing a conceptual model of grouping, this thesis provides researchers with a more holistic understanding of the types of groups that *can* exist, and the conditions that cause them to form. This gives researchers a novel lens to frame their work, and allows them to decide which group types they explicitly or implicitly want to support through their applications.

The second contribution demonstrates how our conceptual model translates to a generalizable software architecture. Through GCF, we provide developers with a robust toolkit that supports a wide range of group types and use cases. This gives researchers a starting point from which to create their own context-aware applications, and provides a reference design for future generations of context-sharing system.

The third contribution illustrates the importance of opportunistic groups in context-aware computing. Through our example systems (Didja, Snap-To-It, Impromptu, and Bluewave), we demonstrate how opportunistic groups can be useful in a wide range of real-world use cases, and how they support interactions that are either too cumbersome, or incapable of being implemented using current technologies. In addition to serving as motivational examples, this work also helps us better understand and expand the design space of context-aware computing. This allows us to identify potential problems, and identify areas for future research.

The fourth contribution is a validation of our architecture. By having real-world developers from both academia and industry brainstorm ways to use GCF, we show that the framework has the right abstractions to support the vast majority of use cases. Additionally, we also use these examples to identify the (few) use cases GCF is unable to easily support, and offer possible ideas for future development.

The fifth contribution addresses the practical challenges of using opportunistic groups in context-aware computing. Although our work primarily focuses on the benefits of opportunistic groups, we also acknowledge that there are significant challenges that can prevent our vision from becoming reality. By highlighting these challenges, we help developers and researchers better understand the implications of using a technology like GCF. In doing so, we aim to fuel the need for future research, and help developers make more informed choices when developing and deploying applications that make use of opportunistic groups.

1.6. Thesis Outline

The research described in this thesis explores how opportunistic groupings of devices can be used to create a new generation of context-aware systems. This includes applications that increase users' abilities to interact with new or infrequently encountered devices, as well as applications that increase users' ability and likelihood of interacting with each other. In support of this goal, we have organized this document as follows:

In CHAPTER 2, we provide an overview of existing context-sharing applications and/or frameworks. We highlight the capabilities of each system, evaluate their ability to support opportunistic groupings of users/devices, and present a conceptual model that shows the motivations for forming and working as a group at a systems level. Through this analysis, we show how current systems are insufficient to support opportunistic context sharing on their own, and motivate the need for a generalizable framework such as GCF.

In CHAPTER 3, we describe our first implementation of the Group Context Framework. We provide an initial set of requirements for the framework based on the observations made in CHAPTER 2, and describe how our architecture satisfies each of them. We then present two sample applications that show how our framework supports all of the group types identified by our conceptual model.

In CHAPTER 4, we present four systems that were created using GCF. In comparison to the pedagogical examples presented in the previous chapter, these systems are more fully featured, and demonstrate how opportunistic groups can be used to increase users' access to information and services. For each system, we provide a brief description, and show how it utilizes the framework. We then describe how the lessons learned from creating these systems have caused us to modify the framework.

In CHAPTER 5, we describe GCF's final architecture, based on the lessons learned in the previous chapter. We present a generalizable design process for building opportunistic context-aware applications, and conduct three case studies to show how it can be used.

In CHAPTER 6, we validate our framework through a multi-week brainstorming study with 20 developers from both academia and industry. Our results show that our finalized framework supports a wide range of new and existing context-aware applications, and demonstrate how the ability to share context across applications creates new opportunities for devices to request and receive the information they need.

In CHAPTER 7, we leverage our own experiences using GCF, as well as feedback from developers, to identify the unique challenges of utilizing opportunistic groups in context-aware applications. This exploration is three-pronged, and identifies specific challenges at the developer, user, and technical level. We describe how GCF facilitated these explorations, and show how the lessons learned through this exploration can be applied to future generations of context-aware systems.

Finally, in CHAPTER 8, we conclude by summarizing the work outlined in the above chapters, and identifying possible areas for future research.

2. Background and Related Work

Developing a new framework to support opportunistic grouping and context sharing requires an understanding of relevant prior work. In this chapter, we identify 29 applications, systems, and toolkits that have been developed over the years to support grouping and context sharing between two or more devices. We summarize the capabilities of each system, and assess the extent to which they support opportunistic groups (if at all). Using these systems as a guide, we then present a conceptual model that describes the various situations in which one might form an opportunistic group, and show how current context-sharing systems only focus on some of these situations at the expense of others. Through this exploration, we highlight the limitations of current context-sharing systems, and motivate the need for a comprehensive framework that can support the full range of group interactions.

2.1. Grouping and Context-Sharing Systems

Over the years, researchers have developed a wide range of applications and middleware solutions to allow devices to form groups and share context. In this section, we examine this work in order to explore both the types of use cases these systems are designed to support, as well as the technologies/techniques that they draw upon to facilitate these interactions. Specifically, we have identified five broad categories of grouping and context-sharing systems:

1. Systems that use groups to increase users' situational awareness
2. Systems that use groups to improve communication and/or collaboration
3. Systems that use groups to support new interaction opportunities
4. Systems that use groups to extend a device's capabilities
5. Systems that use groups to share or conserve resources

Our goal in this section is not to provide a comprehensive look at *every* system that supports grouping and context sharing between devices, but instead to identify *representative use cases* where doing so makes sense from both a user and developer standpoint. This provides a better understanding of the ways in which groups have been used thus far, and highlights the limitations of current approaches.

2.1.1. Using Groups to Increase Situational Awareness

Some of the earliest context-sharing systems focused on improving users' situational awareness. In these examples, each user has a device (*e.g.,* a badge, PDA, and/or smartphone) that automatically tracks and transmits their contextual state (*e.g.,* "Out of the Office," "In a Meeting") to a centralized server. The systems then distribute this information to other users to keep each other informed of their respective activities and/or availability. Although these systems allow users to easily view each other's context and determine if and when they should interact with one another, they assume that all potential group members are known *a priori*. As a result, these systems work well for user bases that are predictable, well-defined, and long lasting (*e.g.,* users that all work in the same building, project teams), but are not intended for use in situations where the total set of potential group members is unknown or constantly fluctuating.

2.1.1.1. Active Badge and ActiveMap

The Active Badge system [116], and its follow up, ActiveMap [81], are location tracking systems that facilitate and encourage informal, opportunistic interactions between office coworkers. In these systems, each user is provided with a uniquely identifying badge that can be tracked *via* a series of overhead infrared sensors. As users walk throughout the workplace, their location is constantly monitored and logged on a centralized server. Other users can then view their coworkers' locations by using a dedicated desktop application, or by looking at a number of preconfigured displays and/or maps strategically placed throughout the environment. By providing users with a persistent way to track each other's movements, Active Badge and ActiveMap increase awareness of other coworkers beyond their immediate neighbors. This increases their ability to detect and engage in spontaneous or one-time conversations/collaborations.

Active Badge and ActiveMap are designed to support serendipitous interactions at the user level. These systems do not have any understanding or internal representation of an opportunistic group, but rather treats all users as if they belong to a global "workplace" group. Additionally, while both systems can support a large number of users, these users must be identified in advance so that they can be issued a badge and registered with the server. Finally, while both systems are intended as a tool to detect opportunities for informal interactions, they lack the intelligence to recognize these opportunities on its own. Instead, users must analyze the incoming location data and determine which groupings, if any, are relevant to them at any given moment. This increases their cognitive load, and can cause them to miss out on useful grouping opportunities if they are not paying close attention.

2.1.1.2. Hubbub

In Hubbub, Issacs *et al.* developed a text-based instant messenger application that helps users maintain background awareness of their friends and/or co-workers [61]. Hubbub expands upon the idea of a traditional instant messenger through its novel use of audio cues. When users first sign into the system, they select a short sound clip that serves as their unique ID. In addition, the system also assigns sound clips to common activities or responses (*e.g.,* "Hello", "I'm Busy"). By playing these cues in succession, Hubbub is able to quickly convey contextual information to the user, such as when their friends have logged in and/or are able to respond to messages. This allows users to be kept informed of their friends' status and availability without having to look at the application.

Hubbub is similar to Active Badge/Map in that it facilitates opportunistic grouping at the user level. By notifying users of when their friends become active, Hubbub allows them to determine the most opportune time to initiate a conversation and receive a timely response. Yet while their overall intent is similar, Hubbub differs in two key ways. First, the application does not rely on specialized hardware. Instead of relying on customized badges, Hubbub is designed to share context that can be easily captured on a desktop or handheld (*e.g.,* Palm handhelds) device. This allows the system to operate in a wider range of environments, thus increasing the opportunities for users to take advantage of its capabilities. Additionally, since Hubbub is based on traditional instant messenger applications, it allows the user to specifically define the users from which he/she will send and receive context. The system is still limited in that it cannot present context from users that are not identified in advance, and does not contain any intelligence to detect or suggest useful grouping opportunities on its own. Yet by allowing users to define their own friends list, Hubbub allows users to be kept apprised of their friends' availability while limiting the impact to their overall cognitive load.

2.1.1.3. ConChat

ConChat is another instant messenger based application that is designed to help increase situational awareness amongst distributed users [94]. In addition to providing users with simple information concerning their availability or status, ConChat is also capable of accessing the sensors in a pervasive environment to obtain more detailed information, such as room conditions (*e.g.,* light, sound, temperature), the identities of other people in the room, and other applications and devices running nearby. This information can then be processed by the system's rule-based architecture in order to infer higher levels of context (*e.g.,* whether or not the recipient is participating in a meeting or talking to another person), and provided to the user in order to determine whether or not to initiate a conversation.

ConChat is notable in that it significantly expands the types of context that can be shared between devices. In contrast to the work above, which only look at a limited subset of context (*e.g.,* location, availability), ConChat provides a standardized model for specifying, obtaining, and processing a wide range of contextual information through a single architecture. Similar to the work described above, ConChat provides the means for devices to share information, but has no notion of a group outside of a shared communications channel (*i.e.,* a server). Thus, while it drastically increases both the quality and quantity of context that can be shared, it still relies heavily on the user to be able to manually parse this information in order to find and form meaningful groupings.

2.1.1.4. Community Bar

In the Community Bar system, Tee *et al.* used screen sharing to keep users apprised of each other's activities [108]. Here, users run a dedicated application that captures their context (*i.e.,* a screenshot of their desktop) at predefined intervals. These images are then published to a dedicated channel and shared with others. By periodically glancing at these images, users are able to see what documents/artifacts their teammates are working on at any given time. This allows them to more easily coordinate activities, track each other's progress, and detect opportunities to engage in spur-of-the-moment conversations.

Community Bar is primarily geared towards well-defined groups, such as users working on a joint project or enrolled in the same class. In order to share screenshots of their desktop, the user must first define a "place" object from within the application that serves as a common communications channel. Other users, in turn, must explicitly join the place before they can see the screenshots or post their own. By enforcing this level of coordination, the Community Bar system is optimized for users that know that they need to work together beforehand. It promotes a simple and effective way of allowing users to post and share context (*i.e.,* screenshots), but is only practical for groupings that are both long lasting and known in advance.

2.1.2. Using Groups to Improve Communication and Collaboration

The systems described in the previous section are based on a simplistic model of grouping. They facilitate context sharing amongst a broad group of users (*e.g.,* all of the workers in a building) in order to help them identify useful grouping opportunities, but assume that specific groupings are created and managed by users themselves. In contrast, the systems described in this section provide explicit support for a small group of users that share a common goal or task. In addition to keeping group members apprised of each other's activities, these systems also provide tools that allow group members to easily communicate and collaborate with one another. Although these systems support a wider range of interactions between users, they still heavily rely on *a priori* information in order to determine which users are members of the group, and what their respective roles are. Consequently, like the previous examples, these systems support groups that are well defined and meet regularly, but are not intended for for one-time, spontaneous interactions.

2.1.2.1. TeamSpace

In the TeamSpace system, Ferscha used virtual environments in order to provide geographically separated teams with a common work and collaboration space [40]. When users log into TeamSpace, they are provided with a three-dimensional model of a virtual room. They can then interact with specific objects (*e.g.,* tables, cupboards, projectors) in order to create, view, or manipulate work artifacts (*e.g.,* documents, presentations). When multiple team members log into TeamSpace at the same time, they are able to see virtual representations of each other, as well as the artifacts that they have created. Additionally, users can use TeamSpace to visually observe what team members are working on without having to ask them. This allows them to be kept apprised of other team members activities as if they were all in the same physical space.

TeamSpace utilizes a traditional client-server architecture. In order to use the system, a group coordinator (*i.e.,* a user or administrator) must specify which users are associated with a given team. The system then uses this information to redirect team members to the same virtual room when they log in. Although this setup provides an easy way for the system to identify group members, it also assumes that team membership is both static and known in advance.

This makes the system useful for building team awareness, but prevents the system from easily supporting collaborations between users that are not aware of each other ahead of time.

2.1.2.2. Group Interaction Support System

In the Group Interaction Support System (GISS), Ferscha *et al.* developed a prototype chat application that improves users' abilities to communicate and interact with one another [41]. In their system, users create chat groups on a centralized server that correspond to physical locations in an environment (*e.g.,* a building, a room), or a central theme. Other users can then log into the chat server on their mobile device, view the existing groups, and decide which one(s) they want to join. As users join groups, their context (*e.g.,* location, activity, system time) is periodically uploaded to a central server and fused with all other group members. The resulting *group context* is then distributed to give group members a collective sense of where they are and what they are doing.

While GISS allows users to directly specify which groups they want to participate in, the need to constantly update their group membership can be mentally taxing. To address this, GISS also allows devices to form *dynamic* groups. Whenever the system detects that more than 50% of the members of a (predefined) group are located in the same physical space, it automatically forms a new subgroup that represents a meeting. Users can then chat and share notes with each other without having to set up a group on their own. Similarly, the system also allows an administrator to designate certain locations (*e.g.,* common areas) of an environment as public meeting spaces. User devices then automatically join the group whenever their devices detect that they are collocated in these areas.

Although GISS' support for dynamic grouping increases the types of groups that users can participate in, the system is still highly reliant upon *a priori* knowledge to enable these interactions. Before GISS can set up a dynamic group, for example, it must be provided with both 1) a list of potential group members, as well as 2) a list of locations where a group can form. The authors explicitly mention that this functionality is intentional, and is intended to only allow grouping to occur in locations "where it makes sense to form groups." Yet by restricting dynamic groups to a handful of preprogrammed locations, the system's ability to support dynamic groupings is limited, and only works in situations that are identified or planned in advance.

2.1.2.3. Context Aware Ephemeral Groups

In the Context Aware Ephemeral Groups (CAEG) work, Wang *et al.* developed a software model to support social groups in a ubiquitous computing environment [114]. In CAEG, each group is defined by a *user group profile* that describes both 1) the purpose of a group, as well as 2) the user's role within it (*e.g.,* moderator, note taker). Additionally, each user group contains one or more *group session profiles* that describe the exact contextual conditions (*e.g.,* start time, end time, location) that need to be met before a group can form. To create a group, a coordinator (*i.e.,* a user or administrator) generates both profile objects and distributes them to all potential group members' devices. These devices then continuously monitor their context, and form *ad hoc* groups with one another when the corresponding grouping conditions for a particular session are satisfied.

Similar to the other work discussed thus far, the CAEG model relies heavily on *a priori* coordination. In order to create a group, coordinators must generate the corresponding user group and session profiles, and distribute it to all potential group members. This level of coordination is cumbersome, and prevents the system from being useful in one-time grouping situations. Furthermore, while CAEG explicitly looks at groupings of users *and* devices, it only forms a device group to support user level goals (*e.g.,* working on a shared document) as opposed to system level goals (*e.g.,* sharing sensor data or resources). Thus, while the system is capable of forming groups when it encounters the right situations, its ability to take advantage of this functionality is limited to situations that the user (or another coordinator) has specified in advance.

2.1.2.4. Panoply

Panoply is a mobile middleware that lets users quickly collaborate and share digital content [37]. In Panoply, group coordinators (*i.e.,* users) create *spheres of influence* (*i.e.,* a group) that correspond to either a specific geographic location, or a general topic. The coordinator then manually invites members to join the sphere by giving them (*i.e.,* emailing) a digital *voucher* that specifies 1) the identity of the group, 2) the IP address of a centralized server (which coordinates communication amongst group members), and 3) the conditions under which their devices should join (*e.g.,* "join the group when you detect the following Wi-Fi hotspot"). As users move through the environment, their devices periodically check their vouchers to see if the conditions for joining a group have been met. If so, the device connects to the group's server, and begins sending and receiving information.

Panoply has been used in the Smart Party system [39] to allow users to share their music libraries when in a shared public space, and in the nan0sphere system [38] as a way for users to construct interactive narratives based on locations that they had previously visited. Yet while the system does allow devices to autonomously form groups and share context (*e.g.,* music files and location data, respectively), it requires 1) group coordinators to manually identify all possible group members and distribute vouchers ahead of time, and 2) users to download and run a dedicated application for each group that they want to participate in (*e.g.,* an application to share music libraries). This greatly limits the types of group interactions that it can seamlessly support.

2.1.2.5. Context-Sharing Architecture for First Responders

In [73], researchers at Carnegie Mellon's Software Engineering Institute developed a reference architecture to let first responders and soldiers share information and collaborate during missions. In their system, each responder runs an application on their mobile device that continually monitors their context and transmits it over a common communication channel (*e.g.,* a local area network). A *rules engine* (running on each client) then analyzes incoming context in order to identify important events (*e.g.,* shots fired, assistance required), and displays this information to the user. A key feature of the architecture is that the rules engine is user specific, and can be customized so that users only receive updates that are relevant to their organizational role. This lets responders at all levels be kept apprised of the events that matter most to them, while minimizing the risk of cognitive overload.

Lewis *et al*.'s architecture is limited by its reliance on *a priori* coordination. Although the architecture's rules engine provides a flexible and extensible way to share and analyze context, it assumes that all users belong to a single "first responder" group, and that the events that they need to know of are identified (and programmed into the engine) in advance. Since the architecture is intended for organizations with a well-defined command and control hierarchy (*i.e.,* military, police, fire), this limitation is largely mitigated. Nevertheless, the emphasis on predefined groupings limits the types of use cases that this architecture can support, and prevents the architecture from working when multiple organizations need to spontaneously collaborate (as can occur during a crisis scenario).

2.1.3. Using Groups to Take Advantage of New Interaction Opportunities

In this section, we provide examples of systems that automatically form groups when they discover new or relevant interaction opportunities. In these examples, devices openly share some contextual information over a pre-negotiated communications medium (*e.g.,* Bluetooth, Wi-Fi). They then analyze this shared context and form groups whenever another device's context matches or complements their own. Although these systems let devices form groups without having to know of each other in advance, they assume that 1) every device reports its contextual state in the exact same manner, and that 2) the conditions needed to form a group are known beforehand. As a result, while these systems are capable of forming groups with devices that they have never met, they tend to be application-specific, and only work when users are all using the same system at the same time.

2.1.3.1. Smart-Its Friends

In the Smart-Its Friends system, Holmquist *et al.* developed a series of portable sensor motes that can automatically form groups (*i.e.* pair) when they experience the same environmental conditions [55]. In their system, each sensor is equipped with an accelerometer, and continuously broadcasts its context using a low-powered radio transmitter. Neighboring sensors can then listen for these accelerometer readings, compare them with their own, and pair when they detect that they are being shaken at the same time.

Smart-Its Friends demonstrates how devices can form groups without having to explicitly know of each other in advance. By defining a communications protocol for sharing accelerometer data, the system allows devices to share context and group, regardless if they have ever met before or will meet again. Yet while the Smart-Its system provides a more generalizable way to form groups, the current implementation is hardware specific, and is only meant to share a single type of context. Consequently, while the system allows devices to opportunistically detect and pair with one another, its ability to support other use cases is extremely limited.

2.1.3.2. Serendipity

In the Social Serendipity project, Eagle and Pentland used Bluetooth radio identifiers to identify opportunities for social interaction between users in a collocated space [34]. In this system, users register their device's Bluetooth ID, profile (*e.g.,* interests) and matchmaking preferences with a centralized "dating" service. They then install a dedicated application on their mobile phone that continuously scans for nearby Bluetooth devices and cross-references their IDs against a database of known users. As users come in range with one another, the system retrieves their profiles and calculates their similarity to one another based on their shared interests. If this similarity score exceeds a threshold, the users receive a notification on their mobile device alerting them of each other.

Although Serendipity supports opportunistic grouping, it is heavily reliant upon its centralized database in order to map Bluetooth names to an individual user. As a result, the system only works with users that have already opted into the system, and have uploaded their Bluetooth ID to the service. Additionally, since user devices must continuously scan for other devices, Serendipity's Bluetooth based discovery system is power intensive. To mitigate this, the researchers have scaled back the system so that it only scans for Bluetooth devices once every five minutes. This extends battery life, but prevents the system from detecting users that have similar interests and are only together for a brief amount of time.

2.1.3.3. Flocks

Flocks is a mobile middleware that supports social networking applications in *ad hoc* environments [11]. In Flocks, users' context (likes, friends, preferences) and grouping preferences (*e.g.,* "form a group with other people who like X") are represented as a set of ordered tuples and characteristic equations, respectively. These values are then stored on the user's mobile device, and shared with other devices when they enter communications range. When users' tuples satisfy a characteristic equation, the system automatically forms a group (*i.e.,* a "Flock") between them. The grouped users can then send messages to each other, which are dynamically routed by the system to the correct destinations.

By allowing users to specify the general conditions for group membership, Flocks provides a way for devices to automatically form user groups. This allows users to communicate with a wider range of people without requiring them to enumerate or maintain a list of all possible group members on their own. Yet while Flocks' ability to form groups dynamically reduces users' cognitive load, it still requires users to input their context (***e.g.,*** "Likes Badminton") and specify the conditions needed for another user to join their flock. As a result, while the system is more dynamic than comparable list-based methods, it still only supports groupings that the user defines in advance.

2.1.3.4. MobilisGroups

In MobilisGroups, Lubke *et al.* developed a location-based service that supports pervasive social computing applications. In their system, users create advertisements for upcoming groups by specifying their date, time, and location to a centralized server. Other users then run a custom mobile application that continually monitors their position and cross-references it against the MobilisGroups service. As users move throughout an environment, the system presents information about nearby grouping opportunities on their device's display. Users can then decide whether or not they would like to join the group and send/receive messages.

MobilisGroups is similar to Flocks in that it allows users to form groups without having to manually specify each individual member. However, whereas Flocks relies on user preferences to automatically form groups, MobilisGroups discovers nearby grouping opportunities, and requires users to manually select the ones they would like to join. This gives users more explicit control, but forces them to be present for every grouping decision. Furthermore, while Flocks utilizes *ad hoc* networking to connect nearby devices, MobilisGroups relies on a client/server architecture to store information about every possible group. This extends the range of groups that users can see, but only works so long as every potential group member is connected to the same server.

2.1.4. Forming Groups to Extend a Device's Capabilities

The fourth type of context-sharing systems are those that form groups in order to gain access to additional services and/or information. Unlike the systems described in the previous sections, which are primarily focused on groupings of users, the systems presented here are more device oriented, and are focused on helping devices gain access to services and/or information that they would not have access to otherwise. Although these types of groupings are potentially more plentiful, they are also harder for users to discover and form on their own. As a result, systems that fall into this category either provide mechanisms for devices to automatically detect and form groups, or provide users with additional mechanisms to interact with their environments.

2.1.4.1. Personal Server

In the Personal Server system, Want *et al.* developed a portable computing device that allows users to access their files in a wide range of environments [117]. The Personal Server is a small, "headless" computer that contains its own processing, storage, and networking capabilities, but lacks any dedicated input/output. As users move throughout their environment, devices in the infrastructure (*e.g.,* desktops, monitors) automatically detect the server and connect to it *via* Bluetooth. The user can then use these external devices to access their server's files and services.

The Personal Server was created to address two critical limitations of mobile devices, namely that 1) mobile devices lack the screen real estate or input options needed for them to be easily accessible, and 2) even the best networking infrastructure is not 100% reliable, and cannot guarantee users continuous access to their files. By allowing the device to form groups with nearby computers and displays, the Personal Server provides users with the best experience possible given the available hardware. Since all grouping in the Personal Server system is automated, users are able to quickly access their files without having to perform any manual configuration. Yet this automation can also be problematic in environments with multiple displays/computers, as the user may not intuitively know which computer the server has paired with. Furthermore, allowing nearby infrastructure to directly connect to the Personal Server creates several user privacy concerns, and can allow malicious users/systemsto access a user's context (*i.e,* files) without their permission.

2.1.4.2. Mobile Gaia

In the Mobile Gaia system, Chetan *et al.* developed a middleware to support pervasive computing applications in *ad hoc* environments [20]. In this system, devices form clusters known as *personal spaces*, and openly share resources (*e.g.,* sensors, context) with each other. To use Mobile Gaia, each personal space must be identified in advance and given a unique and well-known name. Devices are then designated as being either a *client* or a *coordinator*, and are provided with a list of personal spaces that they are authorized to join or organize, respectively. When a coordinator detects that a potential client is nearby (*via* Bluetooth), it transmits a the name(s) of the personal space(s) that it is responsible for. The client then checks its own access list to determine if the group's name is contained within. If so, the devices trade credentials with one another, and form a group.

Although Mobile Gaia allows devices to automatically form groups and share context, the system is highly reliant upon *a priori* coordination between potential group members. In order for devices to form a personal space, every device must already have the name of that space listed in their respective access list (as well as the corresponding authentication credentials). Furthermore, since client/coordinator roles are fixed, devices can only form a group when at least one client and coordinator are present. In both cases, this reliance on pre-coordinated information limits the types of serendipitous encounters that the system can easily support. Mobile Gaia is invaluable in situations when all potential group members can be identified and configured in advance. Yet the system can only form a group under very specific circumstances, thus limiting its overall usefulness in new or unexpected situations.

2.1.4.3. Solar

In Solar, Chen and Kotz created a middleware architecture to support context-aware applications in pervasive environments [19]. In their system, devices in the environment (referred to as planets) oversee one or more types of contextual information, and are responsible for both collecting raw data from sensors and publishing context to an *event stream*. In addition, each planet periodically broadcasts its presence to a centralized *star* server, which serves as a centralized registry. When applications need contextual information, they transmit a request message to the star that contains the type of information they would like to obtain. The star then determines which planet(s) can provide the requested context, and provides the application with the network location of the correct event stream(s).

Solar provides a scalable architecture for requesting and processing contextual information in a pervasive environment. While each planet is responsible for a predetermined set of sensors, Solar's architecture is dynamic, and allows stars to reassign context collection and processing tasks so that no single device is overburdened. Yet while Solar allows devices to automatically form groups in order to share context, the system only manages contexts that are produced by the environment (*i.e.,* sensors, planets), and does not take into consideration contexts that can be produced or sensed by mobile devices. Thus, while the system can intelligently manage requests for context within a pre-established environment, it lacks the flexibility to support use cases where context producers are continuously moving in and out of range. This prevents Solar from being used in a wide range of mobile environments.

2.1.4.4. SenseWeb

In the SenseWeb system, researchers from Microsoft developed a useable platform for opportunistic sensing tasks [63]. In their system, contributors (*i.e.,* users) deploy sensors throughout an environment, and register them to the SenseWeb service. The system can collect data from these sensors as needed in order to meet other applications' needs. When an application needs context, it sends a query message to SenseWeb that contains the type(s) of information that it needs. A coordinator agent then intelligently scans SenseWeb's list of sensors, and dynamically tasks a subset of them to satisfy the request.

SenseWeb is similar to Solar in that it handles many of the tasks associated with finding the right sensors for a given sensing task. The architecture provides a standardized way to access and request contextual information, and allows applications to easily form groups and obtain context without having to know exactly which sensors are available in advance. Yet while the platform makes it easier to form groups with sensors from around the world, it assumes that 1) all sensors are registered in advance, and 2) they never move. Thus, while the system is capable of forming groups over a larger geographic area than Solar, the system suffers from the same technical limitations, and is entirely dependent upon contributors deploying enough sensors for the service to be viable.

2.1.4.5. Virtual Personal Worlds

In the Virtual Personal Worlds (VPW) project, Hong *et al.* created a new model that allows users to easily discover and take advantage of services in a ubiquitous computing environment [56]. In their model, users' devices continuously scan their environments in search of relevant services (*e.g.,* printers, scanners, price comparison tools). These services, which are referred to as *virtual objects*, are then stored on the device's local memory. As users move throughout their environments, they accumulate more virtual objects. They can then freely access these objects and either make use of their services directly, or combine them to perform highly customized tasks.

The VPW model demonstrates how opportunistic groupings of devices can increase users' access to relevant information and services in a ubiquitous computing environment. Instead of having to manually search for virtual objects, the VPW model calls for devices to automatically scan the environment and add services as they are found. Yet while the VPW increases users' access to services, the model becomes unwieldy in an environment with hundreds, or possibly thousands of devices. By forming a group with *every* virtual object, the model maximizes the user's ability to interact with the environment, but forces them to search through a lengthy list of virtual objects in order to find ones that are relevant at a given time. This inability to filter virtual objects based on the user's current context makes the VPW approach cumbersome, and prevents the system from providing users with an intuitive experience.

2.1.4.6. Participatory Sensing Platforms

In addition to the work described above, researchers have also deployed numerous systems that allow users to contribute their device's sensors in support of large-scale sensing tasks (for a more in-depth summary of these systems, refer to [52]). Work in this area has been especially popular in the fields of urban planning and citizen science, and has helped developers collect information about bike riding habits [36], pollution levels [15], highway congestion [109], endangered species populations [87], and find lost objects [106,124].

In each of these examples, the ability to form a group with hundreds, or potentially thousands of devices provides systems with a wealth of information that they could not readily obtain on their own. Yet while these systems are becoming increasingly more popular, they still require users to manually download and run a dedicated application before their phone can join the group and share context. PRISM [22] and AnonySense [106] attempt to mitigate this problem by providing researchers with a single platform that can dynamically *push* sensing tasks to users' phones, and share data with multiple applications. Yet even in these systems, the user still has to manually opt into the system by downloading and running a dedicated app. This creates a barrier to entry, and prevents these systems from casting as wide a net as possible.

2.1.5. Using Groups to Improve Efficiency

The final set of systems we cover in this chapter utilizes grouping in order to conserve resources. In these systems, devices openly scan their environment in search of other devices with similar capabilities (*e.g.,* sensors) to their own. These devices then autonomously form a group and take turns collecting and sharing contextual information in order to extend their overall battery life. Similar to the previous section, the groups formed by these systems are highly opportunistic, and are oftentimes formed without the user's explicit knowledge. However, since these systems focus on a very specific use case (*i.e.,* saving energy), they tend to be highly selective of the groups they choose to participate in. This causes them to overlook groupings that are potentially useful to the end-user, but too costly from an energy consumption standpoint.

2.1.5.1. Energy Efficient and Accuracy Aware (E2A2)

In E2A2, Huang *et al*. developed a crowdsourcing-based location service that allows nearby devices to share GPS data [58]. In their system, each device periodically transmits its GPS coordinates and activity (*e.g.,* walking, riding a bus) to a centralized server. The server then dynamically groups collocated devices, and provides devices with a single coordinate that represents the group's centroid location. By allowing devices to dynamically join and leave groups as they move in an out of range, E2A2 lets devices turn off their GPS or sample their sensors much less frequently than they would if left on their own. This allows individual devices to reduce battery consumption by up to 33%, without violating the accuracy constraints imposed by typical location-aware applications.

Although E2A2 supports opportunistic grouping, the system is restricted in both the types of context it can share (*e.g.,* location) and the use cases its supports. E2A2 only works when devices are 1) connected to the same global server, 2) only need to know the user's current (or group) location, and 3) are frequently in close contact with each other (less than 25 meters). Thus, while the system prevents nearby devices from collecting redundant context, it lacks the flexibility to support a wide range of context-aware applications.

2.1.5.2. Remora

In the Remora system, Keally *et al.* showed how sensor sharing can be used to optimize individual activity recognition tasks in body sensor network applications [64]. In their system, each user has a number of wearable sensors that broadcast their readings to a centralized computer (*i.e.,* an Android smartphone). The phone then analyzes these readings in order to determine what activity the user is performing. When two or more Remora users are nearby, their devices are able to "overhear" each other's radio transmissions, and determine if they are likely performing the same activity. The systems can then take turns running their sensors and classifying the current activity for the entire group.

Remora is similar to systems like E2A2 in that they allow devices to form groups and share context (in this case, activity classifications) in order to save power. However, there are two important differences. First, Remora does not rely on a centralized server to aggregate individual sensor readings and form groups. Instead, the system can automatically detect nearby users by their sensors' radio transmissions, eliminating the need for backend infrastructure. Secondly, Remora can also *combine* readings from multiple users' sensors when classifying the group's activity. This allows the system to turn off higher power (and potentially more accurate) sensors, without affecting the system's accuracy.

Yet despite these differences, Remora suffers from the same fundamental limitations as E2A2. The system only focuses on a single type of context (*i.e.,* activity), and only forms groups when doing so will conserve power. Thus, while the system works well for its intended purpose, it is not intended to be used as a general purpose context sharing platform.

2.1.5.3. CoMon

The Cooperative Ambience Monitoring (CoMon) Platform is a software library designed to support applications that must sense their environments for long periods of time [71]. In CoMon, applications keep track of the types of contextual information that they use, and perform Bluetooth discovery to detect nearby devices that need the same information. The devices then form an opportunistic group and take turns running their sensors and providing context to each other. This allows them to collectively save power without decreasing the quality of information being provided to their respective applications.

Unlike E2A2 and Remora, which are application specific, CoMon is designed to support context sharing *across* applications. By packaging its functionality into a generalizable software toolkit, CoMon allows applications to work together as long as they both need the same context. This allows them to form groups under a wider range of circumstances, thereby increasing their opportunities to conserve battery life. Yet while CoMon supports opportunistic grouping, the system's focus on improving battery life causes it to only consider groupings with 1) devices that it encounters regularly, or 2) devices that remain in range for extended periods of time (*e.g.,* more than 5 minutes). This can cause the system to waste power searching for long lasting groupings in environments where users are continuously moving in and out of range, thus limiting the system's overall usefulness.

2.1.5.4. ErdOS

In contrast with CoMon, which works at the application layer, ErdOS is an *operating system* module that allows devices to form opportunistic groups in order to conserve energy [110]. Similar to other systems, ErdOS uses Bluetooth to detect nearby devices and share context. The system, however, also uses machine-learning algorithms in order to predict the best times to form a group based on the user's prior activities, application usage patterns, and social circles. This allows the device to only scan the environment when the chances of forming a group are high, thus increasing energy savings. Furthermore, the system introduces a *fairness metric* that allows devices to determine how often they have assisted others in the past. This allows devices to evenly distribute sensing workloads, and prevents freeloaders from continually borrowing resources without contributing anything to the group in return. Finally, since ErdOS can access requests for contextual information at the operating system level, the system is already compatible with existing applications. Thus, unlike CoMon, developers do not need to modify their code in order to take advantage of ErdOS' functionality.

ErdOS addresses many of the challenges associated with enabling opportunistic resource sharing in a mobile environment, such as detecting nearby devices, efficiently distributing work tasks, and providing provisions to enforce fairness and user privacy [111]. Yet while ErdOS is intended to be backwards compatible with existing context-aware applications, the system only works with contexts that can be reliably sensed by another device. While this works for some contexts (*e.g.,* location), there are many types of applications (*e.g.,* a compass app) that can only use sensor data from the local device. Since ErdOS does not know *how* context is being used by an application, the system has no way to determine when it is safe to get context from another device, and when context must be produced on its own. As a result, the current implementation only supports context sharing amongst a limited number of (safe) sensors. This mitigates many problems, but severely limits the types of groups that the system can take advantage of to increase battery efficiency.

2.2. A Conceptual Model to Grouping

Through our exploration of related work, we have identified five high-level use case categories that show how grouping and context sharing has been used to facilitate interactions between users and/or devices. Yet while these categories represent the current state of the art, they are obviously incomplete. Although prior work can show us how grouping has traditionally been used in context-aware systems, they do not fully describe how groups can or might be utilized in the future. This makes relying on these use cases problematic from a design standpoint, as it does not guarantee that the resulting framework will cover the widest possible range of use cases.

To address this problem, we have developed a conceptual model that describes the most common scenarios that lead to group formation [43]. This model (Table 1) is inspired by our examination of existing context-sharing systems, and looks at the motivation to form a group in terms of 1) the *task(s)* that group members are trying to perform, and 2) the *data* (*i.e.,* information/services) they need to do so. By looking at combinations of these two factors, our model identifies groupings that are easily recognizable (and are extensively supported by existing systems), as well those that those that are more obscure. This allows us to define a design space for opportunistic groups in context-aware applications, and determine what software abstractions are needed to support the full range of group interactions.

Our conceptual model consists of four quadrants, each of which represents a different group type:

- **Collaborative groups** (Quadrant I) are the most easily recognizable form of grouping from a user perspective, and form when members' tasks and information requirements directly align with one another. In these situations, group members all need access to the same information for the same reason, and work together so that they can access this information as quickly and/or efficiently as possible.
- **Cooperative groups** (Quadrant II) occur when members are performing the same task, but require access to different sets of data. In these groups, individual members lack all of the information needed to accomplish a task on their own. By working together, however, each member is able to contribute towards the group's collective information needs, allowing them to solve problems that they could not accomplish on their own.
- **Convenient collaborations** (Quadrant III) occur when members need the same data, but for different reasons. For example, a smartphone that needs to determine the user's current time zone might form a convenient collaboration with a nearby navigation application in order to obtain GPS data. Convenient collaborations are characterized by their spontaneity, and can be difficult to detect at either the system or user level. Nevertheless, these types of groupings offer a unique opportunity for devices to work together, and allow group members to accomplish their individual tasks with a minimum amount of duplicate effort.
- ***In-Situ* groups** (Quadrant IV) are groups that form when there is a complete disconnect between members' tasks and data requirements. Although in-situ grouping opportunities rarely occur at the *user* level, they allow *devices* to freely share information/resources in support of their (logically separate) tasks. Given that the majority of encounters with other users/devices occur at this level, *in-situ* groupings offer a unique opportunity for context-aware systems to work together, even when they are not running the same application.

When we examine the various systems described above, we see that they can all be easily placed within our model. TeamSpace, for example, can be classified as a collaborative grouping system, as it allows group members to easily view and work on the same documents at the same time. Other systems, such as ActiveMap and Serendipity, support cooperative grouping, and allow devices to share individual context with each other (*e.g.,* their location and personal interests, respectively) so that they can identify useful interaction opportunities. SenseWeb and Solar support convenient collaborations, as they allow multiple applications to take advantage of the same sensors at the same time for different purposes. Finally, systems such as ErdOS and Mobile Gaia support *in-situ* groupings, and allow different applications to freely borrow services (*i.e.,* sensors) from each other in order to achieve their individual goals.

Although our model covers the most common scenarios that lead to grouping, it is important to realize that opportunistic groups are not limited to a single quadrant. On the contrary, supporting opportunistic groups is challenging precisely because they span our entire model. While prior work has shown that it is possible to form groups of devices across a wide range of scenarios, there is no single solution that operates across our entire model (see Table 2 for a complete breakdown). Instead, most group-based systems tend to focus on one or two group types while ignoring others entirely. This makes these systems easier to implement, but limits their ability to support a wide range of grouping opportunities.

Table 1. A conceptual model of groups

	Need Same Data	Need Different Data
Performing Same Task	**Collaborative Group** *"We are in a group because we are doing the same thing with the same data."* Example: TeamSpace	**Cooperative Group** *"We are in a group because we each contribute unique information."* Example: Active Map
Performing Different Tasks	**Convenient Collaboration** *"We are working together because we happen to need the same thing."* Example: SenseWeb	***In-Situ* Groups** *"We are working together because we happen to be nearby."* Example: CoMon

As long as context-sharing systems are designed with a limited set of group types in mind, there will always be a subset of use cases for which these systems are ill-equipped to handle. Thus, in order to support opportunistic groupings, there is a need for a general-purpose framework that supports grouping and context sharing in all of its forms. Such a system not only needs to support all four group types, but do so in a manner that minimizes the need for human intervention so that the cost of forming a group (in terms of time and attention) does not outweigh its benefits. This vision for a simple but extensible way for devices to form and work in groups serves as the central motivation for this thesis. It describes the types of interactions that we aim to enable through the Group Context Framework, and serves as the basis for all of our design choices.

2.3. SUMMARY

In this chapter, we have presented a wide range of systems that allow users and devices to form groups and share contextual information. Through this process, we have identified a number of common limitations. Many systems, for example, support grouping at the device level, but assume that group members are all running the same application at the same time, or are connected to a centralized (known *a priori*) server. Meanwhile, other systems (*e.g.,* Flocks, CoMon, Virtual Personal Worlds) allow devices to autonomously discover group members in real time, but are oftentimes optimized or intended for specific tasks and/or group types, and as such are not generalizable towards other application domains.

In order to summarize the limitations of current context-sharing systems, we have evaluated each system according to the following properties:

- **Reusability.** Specifies whether a system (*and* the context it uses/produces) is only intended for a single application or use case, or if it is designed to support multiple types of applications.
- **Group Membership.** Describes the process by which groups are formed. In a *static* group, all group members are known in advance by the system (*e.g.,* "all users of this application are part of the same group"). In a *dynamic* group, the system automatically adds and removes group members based on rules (*e.g.,* "group with all male users between the ages of 20-25"). Finally, in a *user-specified* group, membership is controlled and managed by the user (*e.g.,* a friend's list) or a group coordinator.

Table 2. Capabilities of current context-sharing systems.

System Name	Reusability		Group Discovery			Group Range		Group Longevity		Supported Contexts			Communications		Group Types			
	Supports Multiple Apps	Supports Cross-App Context Sharing	Static	Dynamic	User Specified	Long	Short	Repeated	One Time	Sensors (Real-Time)	Software (Interpreted Context)	User Input	Static	Ad Hoc	Collaborative (Same Task/Data)	Cooperative (Same Task, Dif Data)	Coincidental (Dif Task, Same Data)	In-Situ (Dif Task/Data)
Active Badge and ActiveMap			x				x	x		x			x			x		
Hubbub					x	x	x	x		x		x	x			x		
ConChat					x	x	x	x		x	x	x	x			x		
Community Bar					x	x				x		x	x		x	x		
TeamSpace					x	x		x				x	x		x	x		
Group Interaction Support System (GISS)				o	x	x	x	x	o	x	x	x	x		x	x		
Context Aware Ephemeral Groups				o	x		x	x	o	x	x	x	x		x	x		
Panoply	x			o	x	x	x	x	o		x	x	x			x		
First Responder Reference Architecture	x		x			x	x	x		x			x			x	x	
Smart-Its Friends				x			x			x			x			x		
Social Serendipity	x		x		x		x	x		x		x	x			x		
Flocks	x			o	x		x	x				x		o	x	x		
MobilisGroups	x				x		x	x	x	x		x	x		x	x		
Personal Server	x			o	x		x	x	x		x	x		o	x	x		
Mobile Gaia	x	x		o	x		x	x	o	x				o		x		x
Solar	x	x	x	o			x	x	o	x			x			x		
SenseWeb	x	x	x	o		x		x		x			x	x		x	x	
Virtual Personal Worlds	x			o	x	x	x	x			x		x			x		
Participatory Sensing Platforms*	o	x	x			x		x		x			x			x		
E2A2	x			x			x	x		x			x		x		o	
Remora				x			x	x		x	x			x	x	o		
ErdOS	x	x		x			x	x		x				o			x	x
CoMon	x	x		x			x	x		x				o			x	x

X = Fully Supports, O = Planned or Partial Support
*Multiple systems are represented by this row.

- **Group Range.** Refers to the geographic distance between the members of a group. In a *short-range* group, members are typically located in the same space or general area (*e.g.,* in the same room or building). In a *long-range* group, members are physically separated from one another (*e.g.,* in different buildings, across the Internet).
- **Group Longevity.** Describes the amount of times that a group will be used over the course of its lifetime. A group is considered to be *repeatable* if it lasts for more than one session. A group is considered to be *one-time*, on the other hand, if it is no longer applicable after a single session.

- **Supported Contexts.** Specifies the types of contextual information that the system is designed to share. *Sensor* based contexts are those that can be detected via physical hardware (*e.g.,* accelerometers, GPS). *Software* based contexts are those that can be detected or interpreted by an algorithm (*e.g.,* walking, idle). Lastly, *user input* contexts are those that are manually provided to the system by the user (*e.g.,* the user's current mood).
- **Communications.** Describes the means by which devices in a group communicate with one another. In a *static* system, group members communicate through a pre-coordinated channel, such as a server or a socket. In an *ad hoc* system, group members create a temporary communications channel between each other.
- **Supported Group Types.** Refers to the system's ability to support each of the four group types (collaborative, cooperative, coincidental, *in-situ*), as specified by our conceptual model.

The results of our analysis (Table 2) reveal five limitations with current context sharing systems:

- **Reusability: Inability to share context across different applications.** Many context-sharing frameworks are intended for a single application and/or use case, and are not designed to let devices share context across applications. While there are a handful of frameworks that do provide this capability (*e.g.,* SenseWeb), they only share contexts that can be captured directly from sensors (*e.g.,* sensor data). This prevents these systems from sharing contexts that must be inferred across multiple sensors (*e.g.,* a user's activity) or collected through software (*e.g.,* a user's language preferences or shopping list).
- **Group Discovery: Lack of support for dynamic groups**. Current support for dynamic groups is extremely limited. Many of the systems described in this section either assume that all users are either part of the same group (*e.g.,* ActiveMap), or require the user to explicitly state which groups he/she would like to participate in (*e.g.,* TeamSpace, MobilisGroups). Although there are some systems that allow devices to form groups on their own (denoted with an 'O'), they still rely on a user or external server to provide them with rules. The CoMon and ErdOS systems are the closest in spirit to the types of dynamic groups that we want to support in this thesis, as they allow devices to form groups and share context without external assistance. Both systems, however, are focused on conserving energy, and, as such, only support opportunistic groups with devices that can increase their battery life.
- **Communications: Over-reliance on a single, well-known communications channel**. Most context-sharing systems rely on a centralized server or broadcast channel in order to allow devices to communicate and share context. This is problematic from an opportunistic grouping standpoint, as it assumes that every potential group member knows about this server/channel *a priori*, and is already connected. While systems such as Mobile Gaia and the Personal Server overcome this limitation by utilizing *ad hoc* networking technologies (*e.g.,* Bluetooth) to allow devices to detect each other and form groups, these systems require the user to manually pair their devices before they can work together. This limits their ability to form groups to groupings that the user is explicitly aware of.
- **Group Longevity: Emphasis on long lasting groups.** Context-sharing systems/frameworks are not typically designed for groups that only occur once or sporadically. While systems such as the Personal Server and MobilisGroups allow users to form one time groups with nearby devices and social groups, respectively, they require the user to manually form/disband these groups on their own. Meanwhile, systems like GISS and CAEG allow devices to form temporary groups, but only in locations and/or times specified by a group coordinator in advance. This prevents these systems from supporting the vast majority of one-time interactions.
- **Group Types: Inability to support the full range of group interactions.** As mentioned previously, the most significant limitation of current context-sharing systems is their inability to support the full range of group

interactions. While many systems excel at one or two group types, they tend to completely ignore others. This prevents devices from being able to form groups under the widest possible set of use cases.

Given these limitations, it is clear that current systems that support grouping and context sharing are insufficient to support opportunistic groups in all of their forms. While our analysis shows that at least one system satisfies each property, there is no one system that supports them all at the same time. Rather than require developers and researchers to combine these systems using their own custom wrappers, our work with the Group Context Framework provides an all-in-one solution that supports grouping across our entire conceptual model. In the following chapter, we present our architecture for the Group Context Framework, and show how our exploration of prior work has been used to create a generalizable framework for grouping and context sharing. We then perform the same analysis on our framework, and show how it addresses or overcomes each of the limitations described above.

3. Developing a Framework to Support Opportunistic Grouping and Context Sharing

In **CHAPTER 1**, we described the importance of opportunistic grouping in context-aware applications, and the need for a framework that allows devices to form groups and share information/services without having to explicitly coordinate with each other. We then examined a wide range of existing grouping and context-sharing systems in **CHAPTER 2**, and showed how their focus on specific use cases and/or group types limits their ability to support a wide range of opportunistic interactions.

In this chapter, we present our implementation of the Group Context Framework (GCF) [43], which supports all of the opportunistic group types identified by our conceptual model (Table 1). First, we describe our high level requirements, and the overarching principles which guided our design. Afterwards, we introduce GCF's core components, and show how they allow devices to request and receive context with minimal prior coordination. Through two applications, GroupMap and GroupHike, we demonstrate 1) how GCF can be used by application developers to easily request and receive context, and 2) how it allows devices to form groups and share context when performing different tasks. Finally, we evaluate GCF using the same criteria we established in **CHAPTER 2**, and discuss how our framework addresses each of the limitations identified in prior work.

The work described in this chapter directly addresses our first research question ("How can we allow devices to opportunistically form groups and share context?"). Through the design of GCF's architecture, we are able to identify the high level requirements and software abstractions needed to make opportunistic grouping and context sharing viable in a large number of real-world situations. Moreover, by creating a working implementation of this architecture, we provide researchers and developers with a flexible middleware solution that makes opportunistic grouping and context sharing viable in a large number of real-world situations. Our framework not only makes it easy for applications to request and share context, but it also allows devices to autonomously take advantage of these grouping opportunities without having to interrupt the user. This allows devices to take advantage of grouping opportunities that are not always immediately apparent, thereby increasing the range of opportunistic interactions that we can explore.

3.1. High Level Requirements and Design Principles

In the previous chapter, we identified five issues with current context-sharing systems that prevent them from allowing devices to opportunistically form groups and share context under the widest possible set of circumstances. As a reminder, these limitations are:

1. Inability to share context across different applications.
2. Lack of support for dynamic groups.
3. Over-reliance on a single, well-known, communications channel.
4. Emphasis on long lasting groups.
5. Inability to support the full range of group interactions.

In this section, we show how these limitations have influenced GCF's design. Specifically, we have identified three high level requirements (and seven sub requirements) that our framework needs to satisfy in order to address or mitigate the problems in current context-sharing systems:

Requirement #1: Provide Standardized Mechanisms for Requesting and Sharing Context. One of the main reasons why context is not commonly shared between applications and/or devices (limitation #1) is that there is no simple way for them to openly request and receive information from each other. Context-aware applications are typically created in a vacuum, with each developer responsible for collecting and using context on her own. As a result, developers are oftentimes so focused on making their own applications work that they do not consider how the context(s) they collect might be useful to others (or provide functionality that will enable said interoperability).

GCF will satisfy this requirement in two ways. First, our framework will *define a standardized communications protocol to let devices request context from themselves, as well as with their physical and/or logical neighbors*. Other devices will then be able to analyze these requests, reply if and/or when they are willing to assist, and transmit packets that contain the requested data. By defining a common language for requesting, advertising, and delivering context, GCF will make it possible for devices communicate and share information, regardless if the applications were created by different developers or running on heterogeneous platforms. This will increase the opportunities for devices to assist one another, without forcing or depending on application developers to implement this functionality on their own.

Secondly, GCF will *provide software abstractions to make context easier to package, reuse, and share*. Our framework will allow developers to create modules that are each responsible for collecting a single type of context (*e.g.,* location, temperature), and formatting them in an application agnostic manner. Other applications can then either include these modules in their code, and/or request context from them without having to worry about whether it is formatted the correct way. By isolating the context being collected and/or inferred from an application's internal logic (a concept referred to by Dey as "Separation of Concerns" [31]), GCF will let developers treat context as a reusable and widely available resource rather than an application specific commodity. This will *allow devices to collaborate and cooperate with each other, regardless if they are performing the same task or require the same information* (addressing limitation #5).

Requirement #2: Support Multiple Communications Technologies. Our second high level requirement is the need to support a wide range of communications technologies. As we saw in prior work, developers have utilized numerous communication architectures and network topologies in order to allow devices to share context. Some systems, like ActiveMap, use a client server architecture to allow users to upload and share location data. Others, such as Flocks, rely on peer-to-peer technologies to let them find nearby users with similar personal interests. In both cases, the selected communications technology is application specific and known *a priori*. This works when an application only needs to communicate with other instances of itself, but prevents multiple applications from doing the same when they spontaneously meet.

Rather than enforce a single communications standard (which will undoubtedly limit the types of applications we can support through our framework), GCF will *come with prebuilt support for a wide range of communications technologies and protocols*. Using our framework, developers will be able to broadcast requests for context to all devices on a local area network using UDP multicasting, share context with a predefined relay server *via* a standard TCP/IP socket, or implement their own custom communication stack. Additionally, GCF will also *let applications use multiple communication technologies* simultaneously. Our framework will provide abstractions so that applications can request and share context with multiple devices without having to know how they are connected at the network layer. This will increase the range of devices that GCF can communicate with at runtime, and make it better suited for situations when devices need to communicate in an *ad hoc* manner (addressing limitation #3).

Requirement #3: Allow Devices to Automatically Form and Maintain Groups. The third high level requirement we have identified is the need to let devices form and maintain groups on their own. As we observed in the previous chapter, many context-aware systems still rely heavily on users to specify the users and/or devices they would like to interact with, or for developers to specify the exact conditions (*e.g.* location, date, time) that can lead to a group. This works for groupings that either the user is directly aware of, or can be programmatically specified in advance, but becomes problem when the members or conditions that lead to a group are either 1) highly transient, 2) not known *a priori*, 3) not immediately apparent to the user.

GCF will address this problem by *allowing devices to form groups on the user's behalf*. In our system, application developers will only need to specify the types of context that they require and the size of the group they would like to form. The framework will then automatically discover devices that can provide this information, and form a group with them (addressing limitation #2). In addition to finding groups, GCF will also *provide a way for devices to dynamically update group membership over time*. Our framework will automatically detect when devices move in and out of range, or when better sources of context have been found, and adjust its group accordingly. This will allow our framework to be used in highly fluid situations, and makes our system practical for groupings that only are viable for a short amount of time (addressing limitation #4).

To summarize, GCF's high level requirements are:

1. **Provide standardized mechanisms for requesting and receiving context**
   - 1.1 Define a standardized communications protocol to request and receive context
   - 1.2 Provide software abstractions to make context easier to package, reuse and share
   - 1.3 Allow devices to collaborate with each other, regardless of if they are performing the same task or require the same information
2. **Support multiple communication technologies**
   - 2.1 Support a wide range of communication technologies
   - 2.2 Let devices use multiple communication technologies simultaneously
3. **Allow devices to automatically form and maintain groups**
   - 3.1 Allow devices to form groups on the user's behalf
   - 3.2 Allow devices to dynamically update group membership over time

In addition to these requirements, GCF's design is also motivated by the desire to make it as developer friendly as possible. With this goal in mind, we have designed the framework according to the following principles:

- **Ease of Use**. Developers are more likely to use a toolkit if it can be easily used. GCF will support this by making its functionality as simple and transparent as possible. Our system will provide developers with a single interface for requesting and receiving context, regardless as to whether the context is obtained from the application itself, or from another device. Additionally, our toolkit will also come with a number of standard modules for collecting and sharing commonly used contexts, such as location, accelerometer, and activity. This will increase GCF's usefulness "out of the box," and let developers quickly incorporate its functionality into both new or existing applications.
- **Extensibility.** While we expect GCF to support a wide range of use cases in its default configuration, we acknowledge that there are many aspects of grouping and context sharing that cannot be predicted in advance. To address this, our framework's components will be designed so that developers can add in support for new context types, grouping strategies, and communication technologies as needed. This will allow our framework to support novel use cases as they are discovered.

By adhering to these principles, we aim to make GCF accessible to both developers that will only use the framework in its current form (referred to by Hudson as *library programmers* [31]), as well as those that will extend it to support their needs (*i.e. toolkit programmers*). This will increase the chances of developers using our framework, which in turn will maximize the range of applications and use cases that can be potentially explored.

3.2. Framework Components

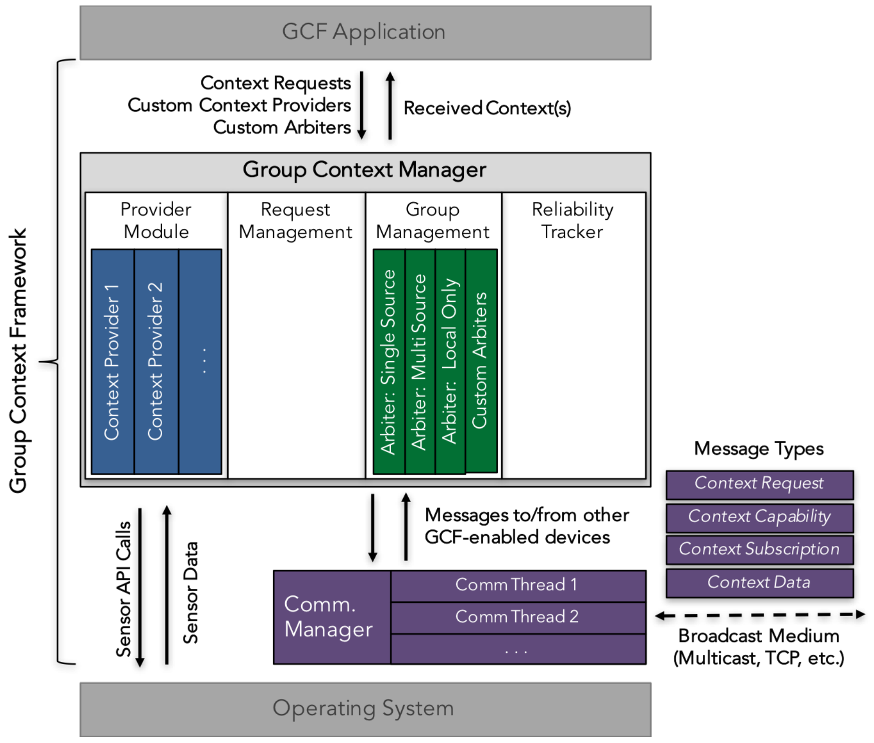


Figure 1. GCF High Level Architecture

GCF's high level architecture is presented in Figure 1. Our framework consists of:

- A **communications manager** (purple) that facilitates message passing between devices and applications (*via* one or more active network connections).
- A set of 0..n **context providers** (blue) that collect, process, and disseminate a specific type of contextual information. (*e.g.,* location, temperature).
- A **group context manager,** which tracks an application's requests for contextual information, and forms groups as needed.

In this section, we describe each component's roles and responsibilities. We then show how they interact with each other at runtime to allow devices to opportunistically form groups and share context.

3.2.1. *Communications Manager*

The communications manager is responsible for passing messages between GCF-enabled devices. It allows the framework to broadcast requests for context, receive advertisements, and subscribe/unsubscribe to other devices' context providers without requiring developers to have to worry about how individual devices are connected (*e.g.,* Bluetooth, Wi-Fi, through a centralized server) at the network layer.

GCF defines four types of messages that are used across our framework to request and receive context (Table 3):

Table 3. Communication Message Types

Message Type	Description	Example Message (Serialized as JSON)
Context Request	Denotes a device's desire for contextual information.	{ "deviceID":"Device A", "contextType": "LOC", "requestType": "SINGLE_SOURCE", "refreshRate": 10000, "payload": [], "messageType": "R", "version":1 }
Context Capability	Denotes a device's willingness to provide contextual information. Sent in response to a Context Request message.	{ "deviceID": "Device B", "contextType": "LOC", "alreadyProviding": "FALSE", "batteryLife": "47", "heartbeatRate": 30000, "sensorFitness": 1, "payload": [], "messageType": "C", "version": 1 }
Context Subscription	Used to subscribe/unsubscribe from a context provider.	{ "deviceID": "Device A", "destination": "Device B", "contextType": "LOC", "updateType": "Subscribe”, "refreshRate": 10000, "heartbeatRate": 30000, "payload": [], "messageType": "S", "version": 1 }
Context Data	Used to share context data.	{ "deviceID": "Device B", "contextType": "LOC", "destination": "Device A”, "payload": "[\"LATITUDE=123.45\”,\"LONGITUDE=67.890"]", "messageType": "D", "version":1 }

Context request messages express a device and/or application's desire for contextual information. Each context request contains (listed in the order that they appear in Table 3):

- The *unique device identifier* of the device making the request. By default, GCF will automatically assign a device a ID by using the device's hardware name or (in the case of Android) operating system ID. However, any globally unique value will suffice.
- The *type of context* (*e.g.,* location, accelerometer) being requested.
- A *type* value to let the framework know what type of group being formed. For *single-source type* requests, the framework will examine all devices and form a group with the "best" device that can provide the requested context (the criteria and methodology used is specified in the following sections). For *multi-source* request, the framework will group with all devices that are willing to provide the information. Finally, for *local* requests, the framework will only generate the requested context using local resources.
- A refresh rate, in milliseconds, specifying the rate at which context is to be delivered to the requesting device.
- (Optional) A payload field, which allows the requesting device to insert custom requirements. For example, if a device is requesting temperature context, this field might contain a value indicating what units the device would like the data reported in (*e.g.,* Fahrenheit). Similarly, if a device is requesting audio amplitude data,

this field might tell the context producer (*i.e.,* the device listening on its microphone) to only report values that are above a predefined amplitude, in addition to once every time period.

Context capability messages advertise a device's advertise a device's willingness to provide contextual information. These messages are sent in response to a context request message, and specify:

- The *unique identifier* of the device that is willing to provide the context.
- The *type of context* offered by this device. This is the same value that is specified in the request message.
- A flag value indicating whether or not this context is already being actively collected and/or shared with other devices. This value is useful in situations when the requesting device needs context, but does not want to wait for the other device's sensors to "spin up."
- The device's *remaining battery life*, in percent.
- A *heartbeat rate*, which specifies the amount of time (in milliseconds) before the requesting device must resend its request in order to keep receiving context.
- A *fitness value*, which represents the quality of context produced by this device. This value is context-specific, with higher values denoting higher quality. For more information on how this value is calculated, please refer to the section on Context Providers.
- (Optional) A *payload* field, which contains additional performance metrics that might be useful to a device when deciding what device(s) to group with. For example, a temperature sensing device might include a value in this field that specifies the range of temperatures that it can reliably sense.

Context subscription messages are sent by a device in order to establish or dissolve a group. They contain:

- The *unique identifier* of the device that is requesting the context.
- The *unique identifier* of the device that is providing the context.
- The *type of context* being requested.
- A flag specifying whether the purpose of this message is to subscribe to another device's context provider (*i.e.,* forming a group) or to unsubscribe (*i.e.,* disbanding a group).
- The desired *refresh rate*, in milliseconds. This value should be equal to the value specified in the original context request.
- The agreed upon *heartbeat rate*, in milliseconds. This value should be equal to the value specified in the context capability message.
- (Optional) A payload field, which contains the same values as in the original request message.

Finally, **context data** messages contain contextual information. These messages contain:

- The *unique identifier* of the device that is producing this context
- The *type of context* contained within this message
- The *intended recipient* for this message. This field allows a device to create context for a specific device, or (if blank) for all devices that are currently subscribed to it.
- A *payload* field, which contains the actual context data.

One of the key features of the communications manager is to make it easy for devices to utilize multiple connections at the same time. To achieve this, each communications manager maintains a collection of *communication threads* that represents its currently active network connections (*e.g.,* a TCP socket, a UDP multicast). When an application needs to communicate with a device (or set of devices across a broadcast channel), the communications manager spawns a new thread for that particular IP address, port, and protocol. Applications can then send messages through the communications manager, which are either forwarded to all threads at once (simulating a network broadcast), or to a specific one. By 1) abstracting multiple *network* connections into a single *logical channel*, and 2) keeping track of which devices it has received messages from on each thread, the communications manager makes it easy for applications to communicate with a wide range of devices using heterogeneous communication technologies. Developers only have to direct the communications manager to send a message, and the framework will automatically find the most efficient path (and protocol) to get it to its destination.

The communication manager is compatible with any broadcast network technology. GCF comes with prebuilt communication threads for TCP sockets and UDP multicast, as these technologies are widely used, support a large number of simultaneous users, and allow for unrestricted communications across a subnet (typically limited to a single building or floor). However, we also allow developers to create their own communication threads as needed. For example, in our own work (see **CHAPTER 4**) we have created communications threads that can communicate with a custom socket server in order to share context across arbitrary distances and/or mobile network providers. We have also created threads that can send and receive messages to MQTT brokers [125] to allow GCF to communicate with Internet-of-Things appliances. Through this approach, we allow GCF to be easily extended to support both existing and future communication technologies, without requiring significant changes to the underlying architecture.

3.2.2. Context Provider

Context providers are tasked with producing, storing, and distributing contextual information. Each provider is responsible for a specific type of context (*e.g., location, time, temperature)*, and a GCF-enabled device can have zero or more context providers running in the background depending upon what information it is able to produce and what it is willing to share.

Context providers perform two tasks. Their primary responsibility is to convert raw data into usable context. The means by which this is accomplished varies depending upon the provider. A *location provider*, for example, may need to use a combination of GPS and Wi-Fi sensors in order to determine the user's geographic coordinates. A *shopping list provider,* on the other hand, may not need to use any sensors, but simply store a list of items provided by a user through a shopping list application. Finally, an *activity provider* may need to use a combination of both sensors and software in order to infer what physical activity the user is performing (*e.g.,* walking, running, standing still), and what activity she is performing on her phone.

The second responsibility of a context provider is to keep track of active subscriptions. Each context provider maintains a list of all of the devices that have requested and subscribed to its services. The provider then periodically transmits the requested context to each device at the specified interval, and listens for periodic heartbeat messages to verify that the device is still in communications range. By listening for heartbeats and canceling subscriptions after a period of inactivity, the context provider is able to autonomously determine which device(s) still need information. This allows a provider to turn itself off when devices no longer need its services, thereby conserving battery life.

As of this writing, GCF already includes a library of context providers for commonly used contexts (*e.g.,* location, light intensity, compass, temperature, barometric pressure, Bluetooth proximity, accelerometer, and audio magnitude). However, we also expect that developers will want to create custom context providers for their particular application, and have made it easy to do so through our framework. As shown in Figure 2, creating a new context provider involves inheriting from the base ContextProvider class (provided by GCF), and implementing the following methods:

1. In the constructor (line 4), developers assign their provider with a string value that is associated with their particular context (*e.g.,* Location = "LOC", Bluetooth = "BT"). This value needs to be globally unique,

```java
1.  public class ExampleContextProvider extends ContextProvider
2.  {
3.    // Constructor
4.    public ExampleContextProvider(GroupContextManager gcm)
5.    {
6.      // Registers the context provider's globally recognized name
7.      super("EXAMPLE", gcm);
8.    }
9.
10.  // Code to initialize the provider
11.  public void start()
12.  {
13.    // Start Sensor(s)
14.  }
15.
16.  // Code to halt the provider
17.  public void stop()
18.  {
19.    // Stop Sensor(s)
20.  }
21.
22.  // Return TRUE if the provider should respond to a request; FALSE otherwise
23.  public boolean sendCapability(ContextRequest request)
24.  {
25.    return true;
26.  }
27.
28.  // Calculates the "quality" of this provider
29.  public double getFitness()
30.  {
31.    // Higher values denote higher quality
32.  }
33.
34.  // Sends context to all subscribers
35.  public void sendContext()
36.  {
37.    gcm.getCommManager.sendContext(new ContextData(...));
38.  }
39. }
```

Figure 2. Sample Java implementation of a context provider

as it is what allows our framework to link context requests to individual providers across applications and/or devices.

2. In the START() method (line 11), developers are able to turn on or initialize the sensors that it needs to begin collecting context. This method is automatically called by the framework when the first device subscribes (*i.e.,* when the number of subscriptions changes from 0 to 1).
3. In the STOP() method (line 17), developers are given the opportunity to turn off sensors and perform cleanup tasks. This method is automatically called by the framework when the last device unsubscribes (*i.e.,* when the number of subscriptions changes from 1 to 0).
4. In the SENDCAPABILITY() method (line 23), the developer is able to examine individual requests for context (both internal and external), and decide whether or not the provider should respond with a Context Advertisement message. By default, providers will always respond to requests from the local device, and will only stop responding to external requests once the device's battery goes below 20%. However, developers can override this method to handle special cases, such as 1) when the device cannot produce

the requested context, or 2) when the request is unreasonable (delivering context every 1ms). This method returns TRUE if the provider should advertise its services, and FALSE otherwise.

5. In the GETFITNESS() method (line 29), developers report the quality of information that is produced by this context provider. Since the notion of quality is context-specific, we require each provider to define its own quality metric. For example, a location context provider might use the accuracy of coordinates returned, measured in meters. Likewise, for a camera sensor, one possible metric might be its image quality in megapixels. GCF assumes that higher fitness values correspond to higher quality, and that this metric is standardized across all providers of the same type.
6. In the SENDCONTEXT() method (line 35), developers are able to specify how the context provider delivers information to other applications/devices. The framework automatically calls this method at the requested interval, and allows developers to transmit the same context to all devices, or to deliver a specific context value to each one.

Context providers are the primary mechanism by which devices form groups and collaborate in GCF. They provide a standardized, application independent interface for collecting and distributing contextual information. Additionally, since context providers are designed to be reusable, they can be openly distributed and used by other GCF-enabled devices. This allows functionally different applications to use the same providers, which in turn allows these applications to share information without having to explicitly know of each other *a priori*.

3.2.3. Group Context Manager

The group context manager (GCM) serves as the controller for the entire framework. It is created at startup, and is responsible for monitoring an application's requests for context, searching for device groupings that best satisfy these needs, and delivering context to the application as it arrives.

The GCM consists of four modules. The **request management module** is responsible for tracking a device's active requests for contextual information. It broadcasts context request messages (*via* the communications manager), periodically contacts providers so that they continue to provide data, and disconnects from providers in the event that the application no longer needs information or when a communications timeout occurs.

The **provider module** is responsible for managing an application's context providers. When created, the GCM does not contain any context providers. Instead, providers are created and registered to the GCM so that developers and/or end users can explicitly specify which contexts they are willing to produce and share. When a request arrives, the GCM queries the provider module in order to determine if a provider with the matching context type is both registered and willing to satisfy the request (*i.e.,* SENDCAPABILITY() returns TRUE). If both conditions are satisfied, a capability advertisement message is then created and sent back to the requester.

The **reliability monitor** maintains a log of communication timeouts for all previously encountered devices. This information is then used to discourage the framework from forming grouping with devices that have been unreliable in the past.

Finally, the **group management module** is responsible for evaluating and subscribing to devices that are capable of producing the desired context. It accomplishes this through a series of *arbiters* that examines each context capability message, and selects the best one(s) according to a specified grouping strategy. More information on this process is provided in the following section.

3.2.3.1. Selecting Group Members

A key feature of GCF is its ability to automatically discover and form groups. For example, in order to receive temperature information from a single device once every five seconds, a developer only has to write the following line of code:

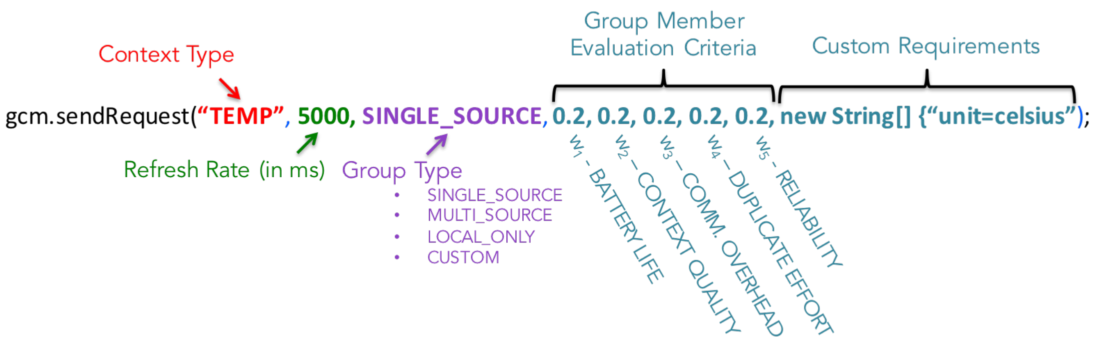


Figure 3. A sample call to the group context manager's SENDREQUEST() method, requesting temperature ("TEMP") data once every 5 seconds.

As shown in Figure 3, a typical context request contains:

- The type of context being requested ("TEMP" = Temperature).
- The desired refresh rate, in milliseconds.
- The type of group being formed. In the above example, the request is for a single source of context.
- (*Optional*) A set of weights that tell GCF how to evaluate potential group members (more information concerning how these weights are used are provided below).
- (*Optional*) A set of one or more custom requirements for this request (*e.g.,* "report temperature in degrees Celsius"). This value is inserted in the context request message's payload field.

In order to form a grouping that best satisfies this request, the GCM transmits a context request message to all devices in communications range, and collects the context capability message(s) sent in response. An *arbiter* then processes these capabilities to determine which device(s) to group with. GCF comes with three prebuilt arbiters: a single-source arbiter, a multi-source arbiter, and a local arbiter. These arbiter match to a corresponding request type, and form groups according to a predefined strategy, as discussed below.

Single Source Arbitration

A single source context request is useful in situations when a device requires contextual information, but is not concerned about where it comes from. This strategy takes advantage of the fact that there are many types of context (*e.g.,* location, temperature) that are not device specific, and can be reliably reported by any device that is within communications range.

In single source arbitration, the arbiter examines the context capability messages provided by each device and selects the "best" one. To calculate the best device, GCF considers the following criteria:

- Battery Life (*b*): To maximize the lifespan of all participating devices, the arbiter will favor devices with the largest remaining battery life. To compensate for variances in battery size and drain rates between mobile devices, each device reports its battery life in estimated minutes remaining.

- **Context Quality (q):** It is not possible to have a universal metric to measure quality. Instead, as previously discussed, context providers must define their own quality metric. The arbiter then compares each provider by this metric and favors those that are of higher quality.
- **Communication Overhead (c):** To ensure that a device is in range, GCF allows context providers to specify a heartbeat interval. Requesting devices must then send a message at least once per interval in order to keep receiving information. The single source arbiter favors context providers with longer heartbeat intervals, as this minimizes overhead for the requesting device.
- **Duplicate Effort (d):** To increase efficiency, the arbiter will try to group with devices that are already providing the requested context to other users. This metric takes advantage of GCF's reliance on broadcast technologies, as it allows a device to provide context to multiple devices without increasing energy consumption.
- **Reliability (r):** GCF uses a reliability value in order to estimate a device's ability to dependably provide context. When GCF first encounters a device, it assumes that is reliable and assigns it an r of 1.0. Then, after each disconnect or timeout, this value is halved. By decreasing the value of r and incrementing it slowly over time, the framework is able to favor context providers that have been reliable in the past, while still giving providers the chance to be reconsidered after enough time has passed.

Each of the above factors are normalized to a 0.0 – 1.0 numeric range. A composite fitness score is then calculated for each responding device using the following equation:

$$\text{Fitness} = (w_1 * b) + (w_2 * q) + (w_3 * c) + (w_4 * d) + (w_5 * r)$$

$$w_1 + w_2 + w_3 + w_4 + w_5 = 1.0$$

where w_1 through w_5 are weights that correspond to each of the five criteria listed above.

By setting w_1 through w_5, developers are able to specify which criteria matter most for their specific application. When using the standard form of *sendRequest*, GCF uses a set of default weights that we have found to yield good overall performance. However, if developers favor some criteria over others, they can modify these values as they see fit. For example, if sensor quality is especially important for a particular application, developers can assign a higher value to w_2, as shown in Figure 4.

```
gcm.sendRequest("TEMP", 5000, SINGLE_SOURCE, 0.0, 1.0, 0.0, 0.0, 0.0, new String[] {"unit=Celsius"});
```

Figure 4. A sample GCF request for temperature data. By setting w_2 to 1.0 and all other weights to 0.0, developers can direct GCF to only consider sensor quality when evaluating potential group members.

After calculating the fitness values, the device with the highest fitness score is selected, and the GCM automatically transmits a context subscription message to it. In the event of a tie, the first device to respond is selected.

Multi-Source Arbitration

Multi-source arbitration is intended for situations when a device requires information from all of its neighbors simultaneously. This method of group formation is useful for applications where each device provides a unique and equally valuable context, (*e.g.*, a map application displaying everyone's positions using their individual GPS receivers).

Unlike single-source arbitration, the multi-source arbiter greedily subscribes to all devices in range, up to a pre-defined or developer defined limit. In this fashion, the arbiter is able to create an *ad-hoc* sensor network, and gather context data across a wide geographic area.

Local Arbitration

Local arbitration is intended for situations when a device requires information that can only be produced internally. An example of this is a compass application, as information from both the accelerometer and magnetometer must both be obtained from the local device in order to provide the user with an accurate heading. The local arbiter uses a simplified fitness function that only takes a context provider's quality (q) into account. Other factors (*i.e.,* battery life, reliability, *etc.*) are ignored as their values will be the same.

Custom Arbiters

The Single Source, Multi-Source, and Local arbiters included with GCF are expected to support a wide range of use cases. However, there may be occasions where a developer needs more explicit control over how the framework finds and forms groups. For example, if a developer is creating an application that tracks the location of first responders, he/she might only want to collect context from a small subset of users (*i.e.,* devices on a predefined white list) as opposed to everybody. Similarly, if a developer is creating an "icebreaker app" (*e.g.,* an app that tells users about the interests and hobbies of nearby people in order to encourage friendly conversation), he/she may only want the device to group with devices that the user has not already encountered before.

To support these specialized use cases, GCF allows developers to create their own arbiters. Our framework comes with a generic ARBITER class that defines its core methods. Developers can then create their own arbiter by inheriting from this base class (Figure 5), and performing the following steps:

```
1.  public class ExampleArbiter extends Arbiter
2.  {
3.   // Constructor
4.   public ExampleArbiter()
5.   {
6.    // This number represents the "group value." It must be unique.
7.    super(100);
8.   }
9.
10.  // Returns the Context Capabilities to Subscribe to Given the Arbiter's Policy
11.  public ArrayList<ContextCapability> selectCapability(
12.   ContextRequest request,
13.   GroupContextManager gcm,
14.   ArrayList<ContextCapability> subscribedCapabilities,
15.   ArrayList<ContextCapability> receivedCapabilities)
16.  {
17.   // Determine which device(s) to group with here
18.   ArrayList<ContextCapability> result = new ArrayList<ContextCapability>();
19.
20.   // GCF will group with all devices contained within this array
21.   return result;
22.  }
23. }
```

Figure 5. Sample implementation of a new arbiter.

1. **Define the arbiter's unique group type.** In the constructor (line 7), developers assign their arbiter a unique group value (*i.e.,* an integer). The value does not have to be globally unique, but must be different from the values that GCF uses for its default arbiters, as shown below:
    a. SINGLE_SOURCE = 0
    b. MULTI_SOURCE = 1
    c. LOCAL_ONLY = 2

2. **Determine which device(s) to group with.** In the SELECTCAPABILITY() method (line 11), developers implement their custom grouping logic. In this method, developers have access to:
   - The context request that was broadcasted *via* the group context manager.
   - The group context manager.
   - A list of context capabilities that the device is already subscribed to for this context type (*i.e.,* current group members).
   - A list of received context capabilities (*i.e.,* potential new group members)

   The method returns a list of devices that GCF will group with (line 21). The framework will then automatically use the results of SELECTCAPABILITY() to 1) subscribe to devices that are in the list, but are not currently subscribed to, and 2) unsubscribe from any devices that are currently subscribed, but are not in the list.

3. **Register the arbiter with the framework.** Once the arbiter has been created, the developer must register it with the framework so that it can be used. This is done by calling Group Context Manager's REGISTERARBITER() method, as shown below:

   ```
   gcm.registerArbiter(new CustomArbiter());
   ```

4. **Request context using the arbiter.** To use the arbiter, the developer calls the SENDREQUEST() method, and provides it with the group value specified in Step 1. For example, if a developer wants to request location context every five seconds using a custom arbiter, he/she would write the following line of code:

   ```
   gcm.sendRequest("LOC", 5000, 100, new String[0]);
   ```

   The framework will then associate the custom arbiter with this request, and use it to process capability messages for this context type.

Periodic Refresh
GCF repeats the request and selection process after a specified amount of time has elapsed, or when the current context provider stops providing context information or moves out of range. This allows devices to continually scan the environment and automatically switch between context providers in order to offer a consistent or higher quality of service. When this process is repeated on multiple devices, it allows devices to dynamically reassign context collection tasks in order to efficiently meet the group's needs.

3.3. EXAMPLE REQUEST FOR CONTEXT

To illustrate how GCF's components operate at runtime, this section describes a sample interaction between four GCF devices (labeled Devices A-D). For this example, Device A is running a GCF-enabled application that needs location data (*i.e.,* GPS coordinates) once every 60 seconds. Devices B and C have a context provider that can satisfy this request, while Device D does not.

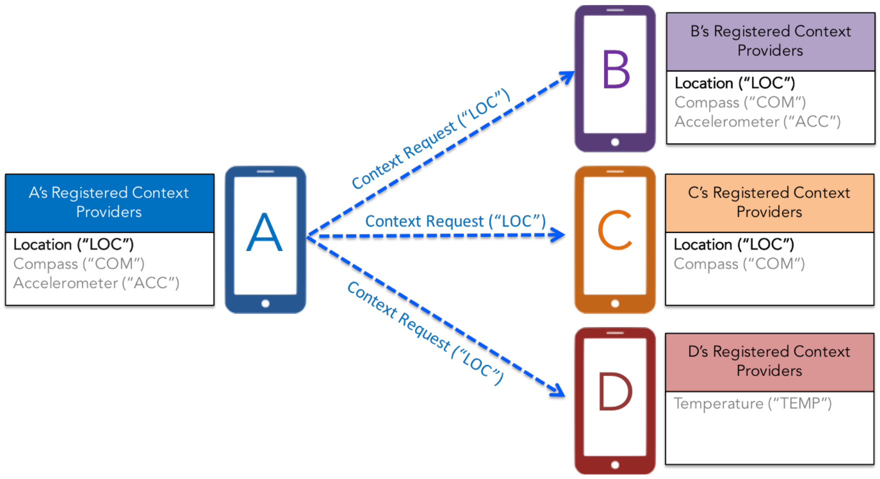


Figure 6. Device A requests location data from Devices B, C, and D.

Phase 1: Initial Request. Device A initiates communication by directing its group context manager to send a Context Request message to all devices in communication range (Figure 6). This is done by calling the Group Context Manager's SENDREQUEST() method (using a version of the method that uses GCF's default weights):

```
gcm.sendRequest("LOC", 60000, SINGLE_SOURCE, new String[0]);
```

The GCM creates a Context Request message containing the specified refresh rate and grouping strategy, and sends it to the Communications Manager to be transmitted on a broadcast channel (*e.g.*, a multicast socket). When Devices B, C, and D's group context managers receive the message, they query their list of active context providers to see if they can produce this information.

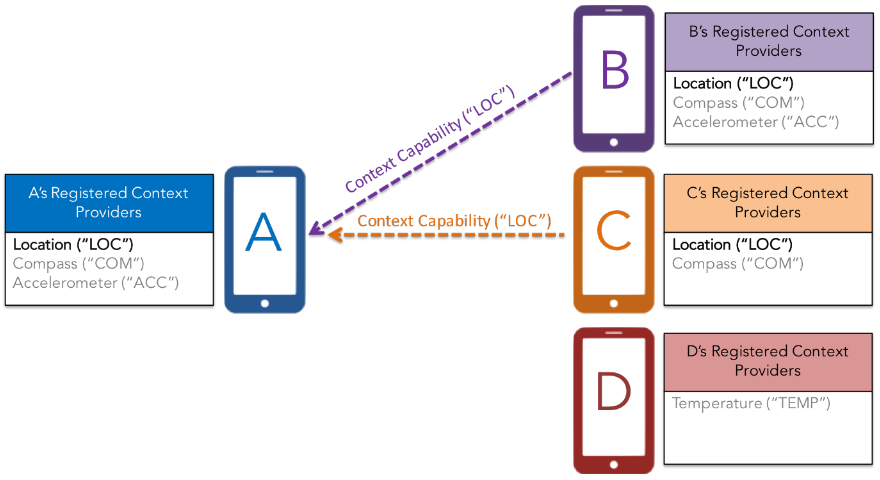

A's Registered Context Providers
Location ("LOC")
Compass ("COM")
Accelerometer ("ACC")
A
Context Capability ("LOC")
Context Capability ("LOC")
B
B's Registered Context Providers
Location ("LOC")
Compass ("COM")
Accelerometer ("ACC")
C
C's Registered Context Providers
Location ("LOC")
Compass ("COM")
D
D's Registered Context Providers
Temperature ("TEMP")

Figure 7. Devices B and C send an advertisement denoting their willingness to provide context to Device A.

Phase 2: Initial Response. In this phase, Devices B and C determine that they have a context provider of type "LOC." They each call the context provider's SENDCAPABILITY() method to verify that it is able to produce the context at the specified rate. IF the method returns TRUE, each GCM sends a Context Capability message to Device A to inform it of their willingness to assist (Figure 7).

Device D, having determined that it does not have a context provider of type "LOC," ignores Device A's request.

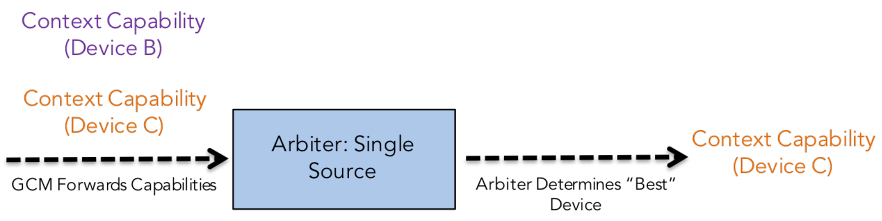


Figure 8. Device A's arbiter evaluates each received capability message, and determines the "best" device to group with

Phase 3: Capability Evaluation. After waiting a set amount of time (*e.g.,* one second), Device A's GCM forwards the received capability messages to its single source arbiter. The arbiter uses the performance metrics contained within each context capability message to compute a weighted fitness score, and selects the device with the highest score (Figure 8). For the purposes of this example, we will assume that Device C has the highest fitness value.

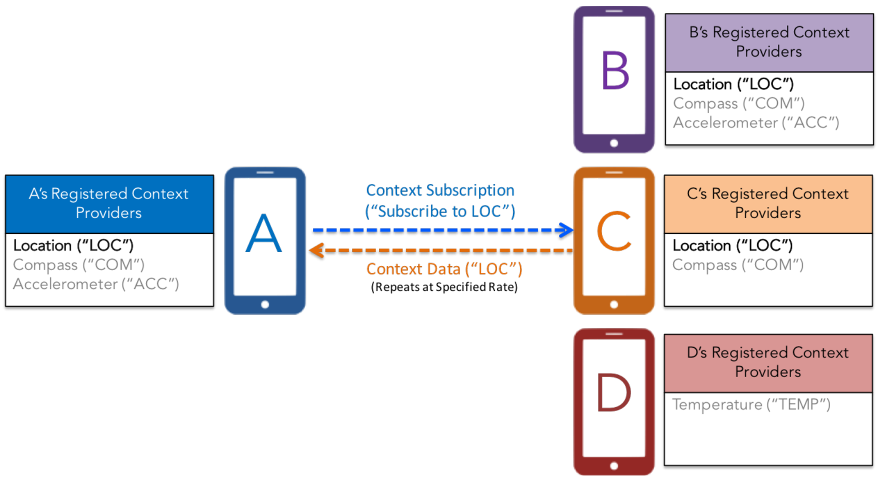


Figure 9. Device A subscribes to Device C's location provider and begins receiving context

Phase 4: Context Subscription and Delivery. Device A sends a Context Subscription message to Device C (Figure 9). Upon receipt, Device C activates the corresponding context provider, which causes it to start collecting and transmitting data at the specified rate.

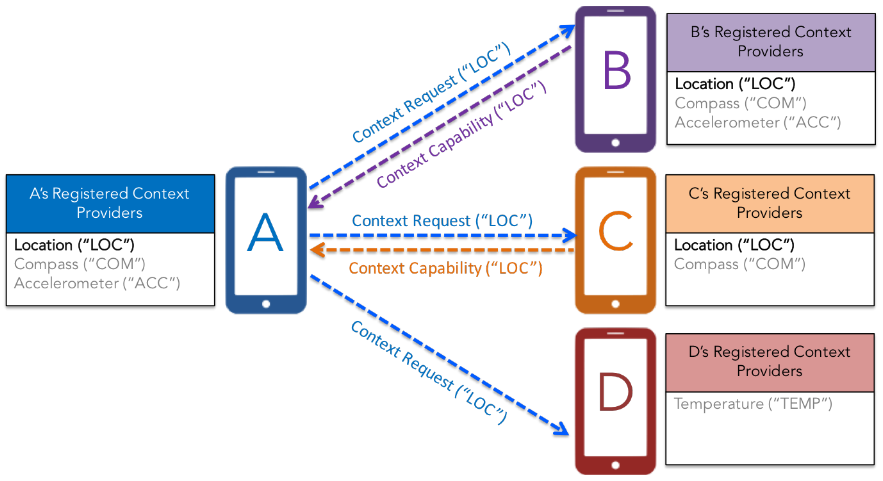


Figure 11. Device A periodically transmits a "heartbeat" message to Device C to keep receiving context.

Phase 5: Heartbeat. At predefined intervals (specified by Device C in its Context Capability message), Device A retransmits its request for location context (Figure 11). This message serves as a heartbeat for Device C, letting it know that Device A is still in communications range and needs data. Furthermore, it gives other devices the chance to reply with context capability messages, which allows Device A to switch to a better provider in the event that one becomes available.

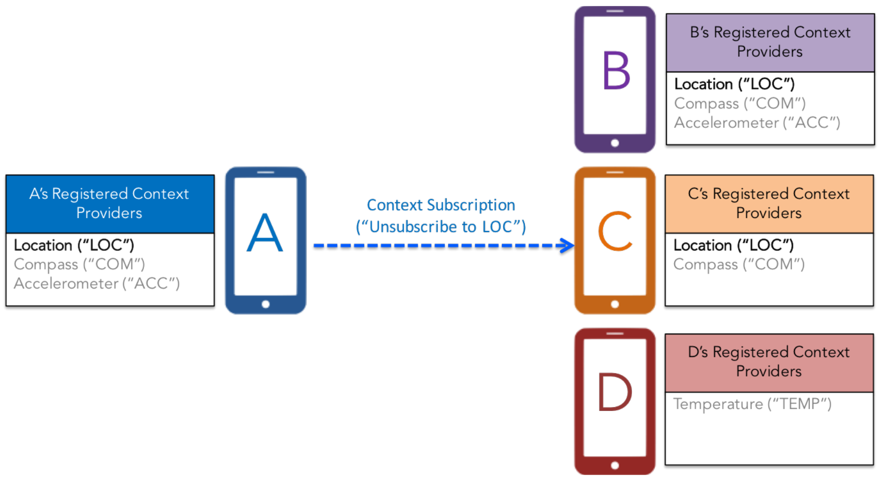

A's Registered Context Providers
Location ("LOC")
Compass ("COM")
Accelerometer ("ACC")
A
B
B's Registered Context Providers
Location ("LOC")
Compass ("COM")
Accelerometer ("ACC")
C
C's Registered Context Providers
Location ("LOC")
Compass ("COM")
D
D's Registered Context Providers
Temperature ("TEMP")
Context Subscription ("Unsubscribe to LOC")

Figure 10. Device A unsubscribes from Device C's location provider when it no longer needs context.

Phase 6: Context Unsubscribe: When Device A no longer needs context, it sends a Context Subscription message to Device C notifying it that wants to unsubscribe from its location context provider (Figure 10). Upon receipt, Device C removes Device A from its list of subscribers, and halts the context provider.

3.4. SOFTWARE IMPLEMENTATION

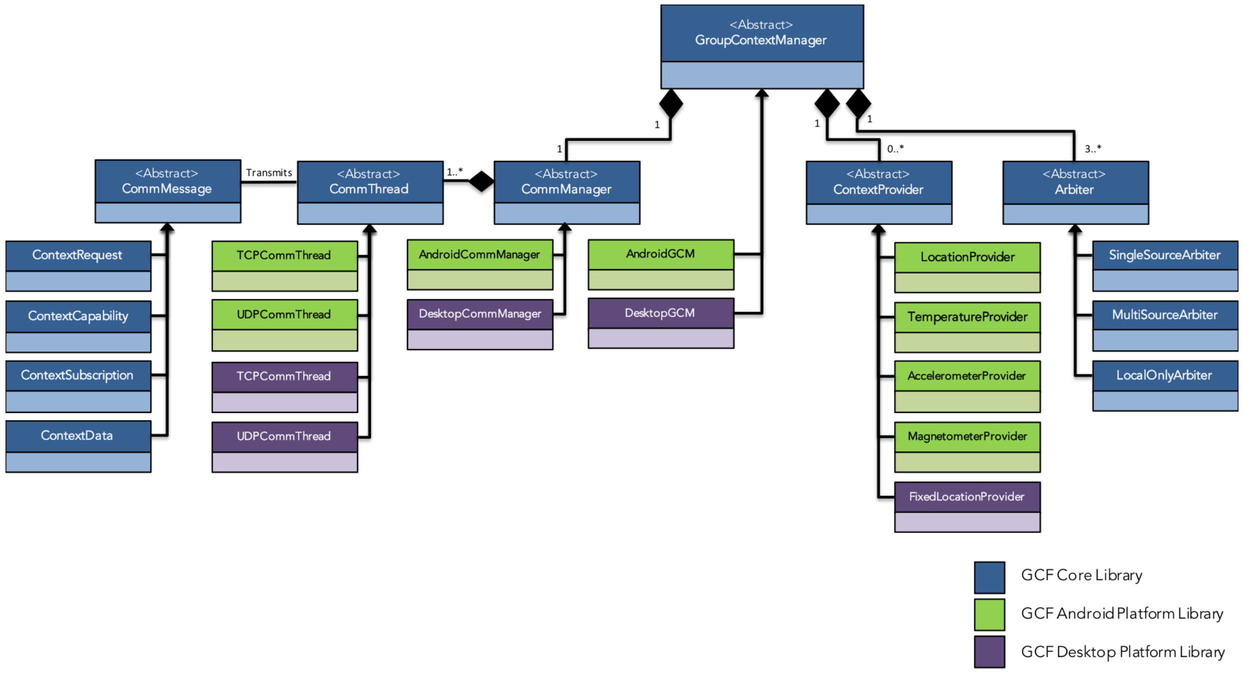


Figure 12. GCF's partial UML class diagram, showing our system's core components (blue), and platform specific libraries (green, purple).

GCF is currently implemented as an application library for the Java runtime platform. A partial UML diagram, showing the framework's major classes, is provided in Figure 12 (see **APPENDIX A** for the complete system architecture):

Not surprisingly, the base classes in this diagram (*i.e.,* GROUPCONTEXTMANAGER, COMMMANAGER) match the components identified in our high level architecture. For cross-platform compatibility reasons, however, GCF's functionality is divided into multiple libraries. The *core library* (blue) houses the framework's platform agnostic components and logic, and provides abstract classes to let developers create their own group context manager, context providers, arbiters, and communication managers/threads. The *platform libraries* (purple, green), on the other hand, implement the abstract interfaces defined by the core library, and allow us to tailor GCF's behaviors to take advantage of each platform's unique sensing capabilities, features (*e.g.,* Android Intents, desktop event handlers), and common programming practices. All supported platforms can communicate using TCP and UDP networking protocols, and all messages are serialized as JavaScript Object Notation (JSON) objects using Google's Gson library [150].

GCF is optimized for both the Android and desktop operating systems (*e.g.,* Mac OS X, Windows, Linux), as both platforms are widely used. However, since the framework is based on Java, it can be easily ported to run on a wide range of hardware. We have already deployed GCF-enabled applications on cheap, low-powered hardware such as the first generation Raspberry Pi [126], and have even gotten the framework to run on devices running Android Gingerbread (a six-year-old operating system as of this writing). Consequently, while GCF will eventually need to be ported to other popular platforms (*e.g.,* iOS) in order to maximize the chances for devices to find and form opportunistic groups, the current version is already sufficient to explore a wide range of context-aware applications.

3.5. Example GCF Applications

In order to showcase GCF's support for multiple group types, we have created two simple mobile phone applications using our framework. The first application, GroupMap, satisfies quadrants I and III of our model by allowing devices to share and broadcast information about their location, respectively. The second application, GroupHike, satisfies quadrants I and II by allowing devices to both *share* location information, and *combine* accelerometer data in order to gather information about an unknown environment. When used concurrently, these prototypes also demonstrate the potential of *in-situ* groupings (Quadrant IV) by showing how GCF-enabled applications can serendipitously work together.

3.5.1. GroupMap

GroupMap is a mapping tool that allows users to track their location. Figure 13 shows a screenshot of the application and its two modes of operation. When users start GroupMap, they are provided with an adjustable map that is centered on their current location. Users can then use to app to view their own location (Figure 13a-b), or the location of any other nearby users simultaneously (Figure 13c).

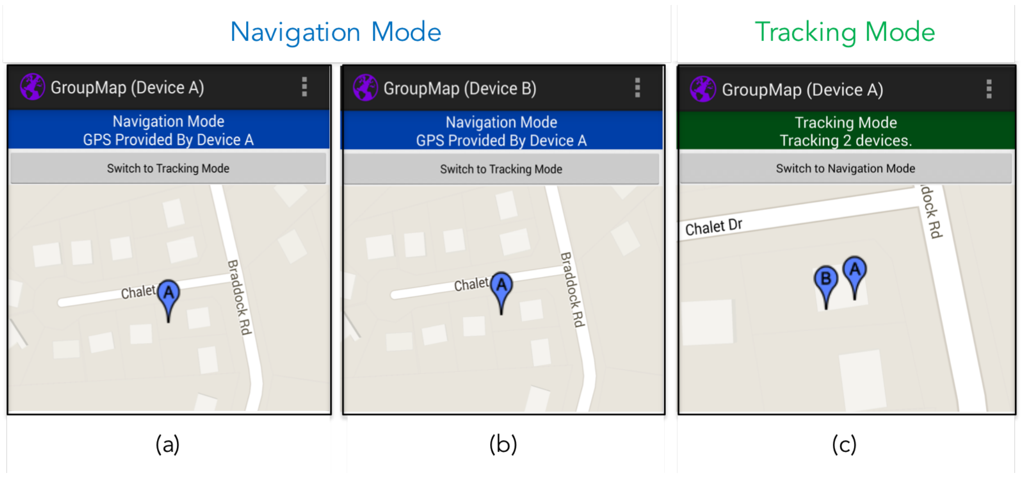


Figure 13. GroupMap screenshots. Through the application, Devices A and B can either share GPS coordinates in Navigation Mode (a and b), or see all users in range in Tracking Mode (c).

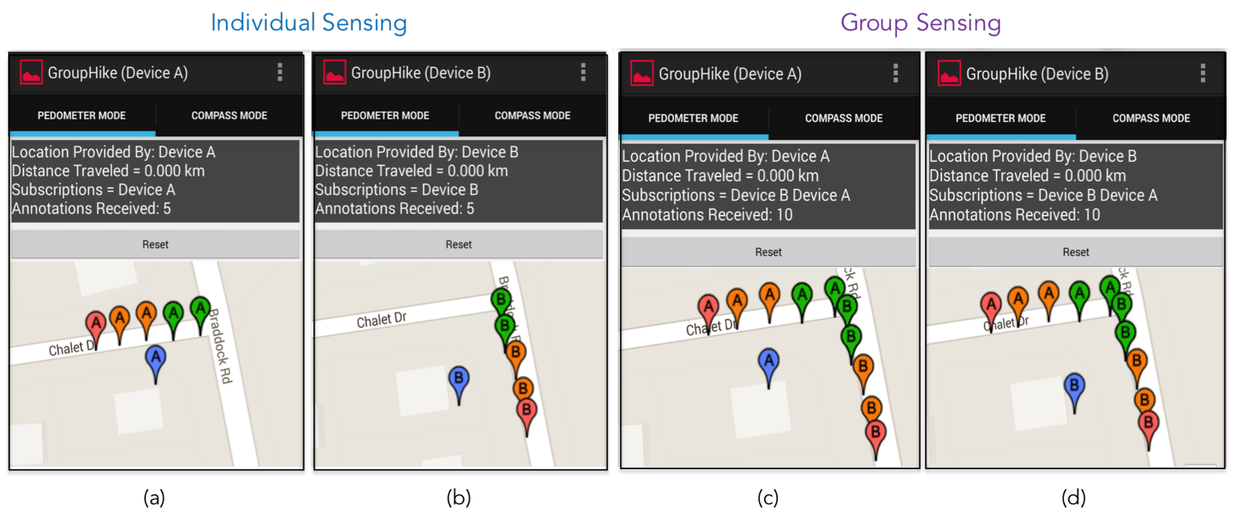
GroupMap utilizes a *location provider* that uses either the phone's GPS or network sensors to determine the user's location. When the application starts, it immediately broadcasts a single-source request for location information. If a compatible device is nearby (as determined by Bluetooth) and is willing to share, the application subscribes to it and begins pulling location data (Figure 3b); else, it uses its own location provider.

GroupMap is primarily a collaboration tool (Quadrant I), as it allows users to share information when they are performing the same task (*i.e.*, navigation/exploration) and require access to the same information (*i.e.*, location data). However, the application can also support convenient collaborations (Quadrant III). By switching to *tracking mode* (Figure 3c), the application begins subscribing to *all* location context providers in broadcast range. This supports two distinct groups of users at the same time: those who only want to know their own location and those who want to know everyone's location.

While GroupMap utilizes grouping for different purposes depending upon the currently selected mode, GCF abstracts the process by which these groups are formed. From a developer standpoint, switching between modes is accomplished by changing the location request type from *single* to *multiple source*. The framework then finds the best group for providing context, and delivers data as it arrives.

3.5.2. GroupHike

GroupHike is a pedometer application designed to monitor a user's hiking experience (Figure 14). The application has two modes of operation. In *pedometer mode* (Figure 14a-b), GroupHike monitors the path that the user is traveling along and creates an annotation whenever the user has a sudden change in instantaneous velocity (as the result of *e.g.*, a hole, a steep step). This information is shared with other users and used to gauge the "difficulty" of traveling along a stretch of terrain (Figure 14c-d). GroupHike also has a *compass mode* (Figure 5, right side) where it uses a combination of magnetometer and accelerometer readings to provide the user with his or her current heading.



Location Provided By: Device A
Distance Traveled = 0.000 km
Subscriptions = Device A
Annotations Received: 5

Location Provided By: Device B
Distance Traveled = 0.000 km
Subscriptions = Device B
Annotations Received: 5

Location Provided By: Device A
Distance Traveled = 0.000 km
Subscriptions = Device B Device A
Annotations Received: 10

Location Provided By: Device B
Distance Traveled = 0.000 km
Subscriptions = Device B Device A
Annotations Received: 10

Figure 14. GroupHike screenshots. Devices A and B collect terrain data individually (a and b), and share it with each other when in range (c and d). Icons show the device's location (blue) and terrain difficulty (green, orange, red).

GroupHike uses three context providers: 1) a *location provider*, identical to the one used in GroupMap, 2) a *compass provider,* which tracks orientation data, and 3) a customized *annotations provider* that stores locations where the user experiences a sudden change in acceleration. When operating in pedometer mode, the application sends a single-source request for location data in order to estimate the user's current position, and a multi-source request for annotations data in order to provide the user with real time terrain information from other users. In compass mode, the application sends a local request for compass data since a user's heading is device specific, and cannot be derived from other nearby devices.

Similar to GroupMap, GroupHike's pedometer mode allows users to share location data to conserve battery (Quadrant I). It also supports cooperative grouping (Quadrant II) by allowing devices to share their own annotation data in order to accomplish the same task (*i.e.*, gauging the difficulty of a particular stretch of trail). By grouping with all devices in communications range and obtaining their individualized annotations, GroupHike builds a comprehensive map of the environment without requiring users to explore it on their own.

3.5.3. Sharing Context Across Applications

Together, GroupMap and GroupHike demonstrate how GCF supports Quadrants I, II, and III of our model. Yet the most interesting feature of our framework is its ability to support *in-situ* interactions between devices (Quadrant IV).

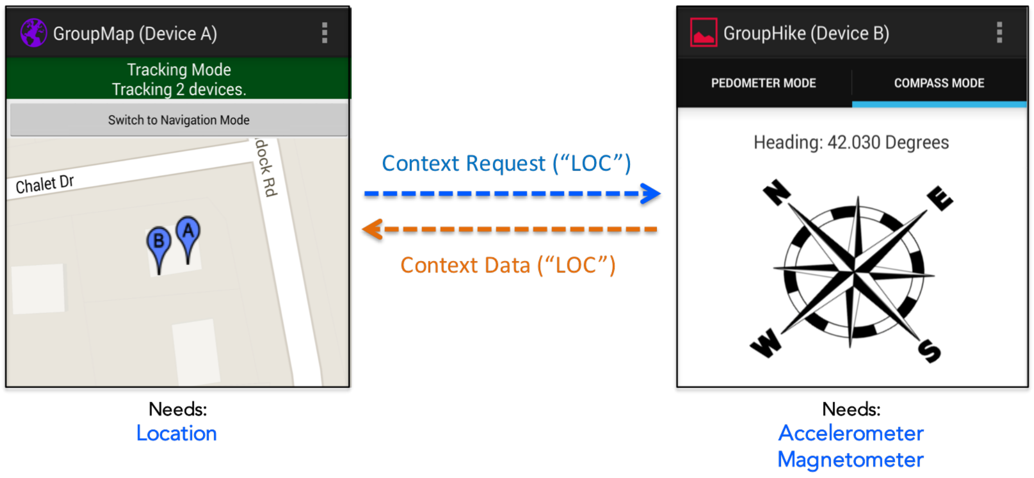


Figure 15. Demonstration of *in-situ* grouping. In this example, Device A's request for location data from nearby devices (left) is answered by Device B's location provider (right). By using GCF, the two applications can share context with each other.

In the real world, the chance of encountering another device that has the same exact application installed and running is relatively low. As a result, many developers only design their applications to work with other devices when it is absolutely necessary. GCF addresses this problem by allowing devices to group and share context *across* applications. This eliminates a significant hurdle to grouping, and allows devices to assist each other, even when performing logically separate tasks.

Figure 15 shows an example of *in-situ* grouping. In this demonstration, Device A (left) is running GroupMap in tracking mode, while Device B (right) is running GroupHike in compass mode. In their current configuration, Device A is utilizing its location provider while Device B is only utilizing its compass provider. Yet because Device B also has a location provider, it is able to respond to Device A's multi-source request for location data. This allows GroupMap to benefit from the information provided by GroupHike, despite the fact that the two devices are performing two distinct tasks (*i.e.,* monitoring user locations vs. monitoring a user's direction), and using different sets of data (*i.e.,* location vs. compass).

Through the GroupMap and GroupHike prototypes, we have demonstrated how GCF can support grouping across all four quadrants of our model. Initially, we expect developers to adopt our framework because it simplifies the process of requesting and receiving contextual information on a single device. As more applications use GCF, however, the chances of opportunistic groupings increase, and developers can take advantage of the benefits of these groupings without having to alter their code or coordinate with others. This "value added" strategy distinguishes GCF from other systems, and provides a compelling reason to adopt our framework.

3.6. Examining the Framework

Now that a functional version of GCF has been created, it is important to see how well the framework differentiates itself from existing solutions. In this section, we conduct this investigation in three parts. First, we examine at GCF from an *architectural standpoint* to see how the framework's components address the five limitations described in **CHAPTER 2**, and satisfy our high level system requirements. Next, we examine GCF from a *usability standpoint* to see how well it supports the high level design principles (*i.e.,* ease-of-use, extensibility) we identified in the beginning of this chapter. Finally, we evaluate GCF from a *functionality standpoint* to see how our framework compares to existing context-sharing systems, and to identify areas for further improvement.

3.6.1. Examining GCF's Architecture

In section 2.3, we identified five limitations of current context-sharing systems that prevent devices from forming opportunistic groups and share information. We then showed in section 3.1 how these limitations led to the creation of our high level requirements. In this section, we examine GCF's architecture to see how well it supports both sets of constraints. The results of this analysis (which are summarized in Table 4) show that our framework touches upon each of the limitations and requirements that we have identified thus far.

From an architectural standpoint, GCF differs from current context-sharing systems in the following ways:

- To allow devices to share context between applications (limitation #1), GCF introduces the *communication message* and *context provider* abstractions. The former defines a common language to let devices request and receive context from each other (satisfying requirement 1.1). The latter provides a standardized module that is capable of collecting and sharing context in a way that is independent from an application's internal logic (satisfying requirement 1.2). When used together, these components let devices openly request and receive context with each other without requiring application developers to explicitly coordinate with one another. This makes context-sharing more ubiquitous, and provides a level of cross-compatibility that is not currently found in existing context-sharing systems.
- To support dynamic groups (limitation #2), GCF provides the *arbiter* abstraction. This component lets the framework automatically evaluate potential groups members according to a predefined (but highly customizable) policy, and select group members that best satisfy an application's information and/or functional requirements. This in turn allows the framework to form groups both when the members of that group are known *a priori* (as is commonly assumed or required by current context-sharing systems), as well as when group members meet spontaneously (satisfying requirement 3.1).
- To eliminate the need for a single, well-known communications channel (limitation #3), GCF provides the *communications thread* and *communications manager* abstractions. The communications thread hides the low level details of a particular communications technology and/or protocol (*e.g.,* TCP, UDP) by providing applications with a generic interface for sending and receiving messages (satisfying requirement 2.1). The communications manager, on the other hand, oversees one or more communications threads simultaneously (satisfying requirement 2.2), and determines which thread(s) a message needs to be sent over to ensure it arrives at its intended destination. Together, these components allow applications to communicate with other devices without having to worry about how they are actually connected at the network layer. This contrasts with current context-sharing systems (which require devices to be connected to the same local area network or server), and allows our framework to work over any combination of dedicated and *ad hoc* channels.
- To support one-time groupings (limitation #4), GCF provides the Group Context Manager. This module continually searches for group members and subscribes/unsubscribes from their context providers, and automatically modifies a device's group memberships as other devices come in and out of communications range. Whereas other context-sharing systems require users to explicitly pair devices, trade credentials, or download a dedicated app, the group context manager automates these processes. This reduces the overhead required to form or join a group in many situations, and lets users take advantage of groupings that are only useful once or for a short period of time (satisfying requirement 3.2).

Through the combination of these various components, GCF is able to support grouping across all four quadrants of our conceptual model (limitation #5). Our framework allows devices to communicate their information requirements, and automatically identifies opportunities for devices to share and/or conserve resources. This maximizes the changes for devices to work together, regardless if they are running the same application or need the same information

Table 4. An analysis of GCF's architecture, showing how our framework's components (right) address the limitations identified in CHAPTER 2 (left), and satisfy our high-level system requirements (middle).

Limitation of Current Context Sharing Systems	Derived High Level Requirement(s)	Supporting GCF Component(s)
1. Inability to share context across applications.	• R1.1 Establish a common communications protocol to request and receive context • R1.2 Provide software abstractions to make context easier to package, reuse and share	• Communication Messages • Context Provider
2. Lack of support for dynamic groups.	• R3.1 Allow devices to form groups on the user's behalf	• Arbiters
3. Over-reliance on a single, well-known, communications channel.	• R2.1 Support a wide range of communication technologies • R2.2 Let devices use multiple communication technologies simultaneously	• Communications Thread(s) • Communications Manager
4. Emphasis on long lasting groups.	• R3.2 Allow devices to dynamically update group membership over time	• Group Context Manager
5. Inability to support the full range of group interactions.	• R1.3. Allow devices to collaborate with each other, regardless of if their tasks and/or information requirements directly align	• All Components

(satisfying requirement 1.3), and provides a more generalizable solution than what existing context sharing systems can offer.

3.6.2. *Examining GCF's Usability*

In addition to providing the architectural support needed to address our high level requirements, GCF is also intended to be both easy to use and extensible from a developer standpoint. In this section, we examine our framework along both dimensions. In doing so, we find that our framework includes many features that allow it to be easily incorporated into both new and existing context-aware applications.

- Ease of Use. GCF provides developers with a streamlined way to add opportunistic grouping to their applications. To use the framework, developers only need to import the associated JAR libraries for their platform into their project, and create an instance of the Group Context Manager. They can then request context from other devices by issuing commands to the GCM (as shown in Figure 4), and receive the data through an Android Intent and/or desktop event handler. Since the process of requesting context is the same regardless of 1) what information is being requested, and 2) which device(s) that information comes from, GCF provides a simple way for developers to access a wide range of context, and offers a compelling reason to use our framework even when they only need information from the local device. Moreover, since all of GCF's functionality is accessible *via* the Group Context Manager, developers only need to interact with a single object to access our framework's entire feature set. This minimizes our system's learning curve, and prevents developers from having to directly interact with our framework's core components (or know how they work) in many situations.
- Extensibility. GCF's architecture is also designed to be highly extensible. As shown in Figure 12, our framework defines abstract classes for each of its core components. Developers can then inherit from these classes or extend our current class libraries as needed. In most use cases, we expect that developers will only need to create their own custom context providers, and have included templates in the source code to make doing so as streamlined as possible. If necessary, however, developers can also create their own arbiters, communications threads/messages, communications managers, and group context manager. This lets developers easily build upon our framework, and even gives them the option to port our system to new platforms and/or operating systems.

3.6.3. *Examining GCF's Functionality*

Finally, we evaluate GCF using the same methodology and criteria we established in **CHAPTER 2** (Table 2) in order to evaluate our framework's capabilities and limitations**.** The results of this analysis are presented in Table 5.

Table 5. Analysis of GCF's functionality.

	Supports Multiple Apps	Supports Cross-App Context Sharing	Static	Dynamic	User Specified	Long	Short	Repeated	One Time	Sensors (Real-Time)	Software (Interpreted Context)	User Input	Static	Ad Hoc	Collaborative (Same Task/Data)	Cooperative (Same Task, Dif Data)	Coincidental (Dif Task, Same Data)	In-Situ (Dif Task/Data)
System Name	Reusability		Group Discovery			Group Range		Group Longevity		Supported Contexts			Communications		Group Types			
Group Context Framework	X	X	X	X	O	O	O	X	X	X	X	X	X		X	X	X	X

X = Fully Supports, O = Partially Supports

As this table shows, GCF offers several improvements over the state-of-the-art. In contrast with existing context-sharing systems, which only support one or two group types, our framework simultaneously supports all four of the group types identified in our conceptual model. Additionally, GCF also supports both hardware and software sensed contexts, and provides automated grouping mechanisms that make one-time or spontaneous groups practical. While previous systems provide some of these capabilities, they typically do so at the expense of other features. CoMon and ErdOS, for example, allow nearby devices to form groups for the purposes of conserving battery life, but do not consider use cases where devices need to share a wider range of information (*e.g.,* shopping lists) over larger distances. Meanwhile, systems such as GISS and CAEG are designed to share both sensed and inferred contexts; they assume that the devices sharing information have *a priori* knowledge of one another, and thus are not suited for one-time grouping opportunities. In contrast, GCF provides all of this functionality through a *single system.* This makes the framework more generalizable than existing work, and allows it to be useful in a wider range of use cases.

Our analysis, however, also reveals three noticeable gaps in GCF's functionality:

1. GCF currently **lacks the ability to automatically form groups based on distance**. While the framework allows devices to communicate over any broadcast channel, it has no way of inferring where these devices are physically located. Consequently, the distance between group members is entirely dictated by the range of the communications channel. For now, developers can work around this problem by requesting location context from other devices, and programmatically filtering them based on their proximity. However, this process is inefficient, and can be circumvented in many situations by giving the framework the ability to "sense" when it is near other GCF-enabled-devices.
2. Secondly, GCF provides **limited support for user-defined groups**. Although one of the main aims of GCF is to allow devices to automatically form groups, there *are* occasions where the user must have some say as to which device(s) to group with. For example, if a device needs to form a group with a printer in order to print a document, it makes sense to give the user a choice as to which device to use rather if multiple viable options are available. Yet while the current framework does not explicitly prevent these types of interactions, it currently relies on the developer to enable this functionality from within their application. This introduces an additional layer of complexity, and makes these types of simple interactions more cumbersome than they need to be.
3. The third, and most critical limitation of GCF is its **overdependence on shared connectivity**. Currently, our framework requires devices to be on the same communications channel (*e.g.,* a TCP socket or UDP multicast address) before they can send and receive messages. This assumption is reasonable in environments with a local

area network, such as buildings and school campuses. It is unreasonable, however, in outdoor or public environments such as parks or malls, as these areas either lack network connectivity altogether, or do not guarantee that devices will always be connected to the same subnet. Consequently, there are still environments where GCF does not work as intended, and this prevents our framework from being able to find and form opportunistic groups in the widest possible set of conditions.

The severity of these limitations vary. The first two limitations identify specific use cases where GCF does not offer a completely automated solution. Consequently, our framework does not actively *prevent* developers from being able to form groups with nearby devices or manually, respectively, but simply requires developers to build upon the existing framework's functionality (*e.g.,* implement a custom arbiter) to provide this functionality. The third limitation (*i.e.,* dependence on shared connectivity) is admittedly more problematic, as it identifies a range of environments where our framework is currently unable to operate. However, these areas are becoming increasingly less common as more environments become instrumented. Consequently, while future chapters will show how we have implemented all of the functionality outlined in Table 5, there are still a wide range of environments where the current version of our framework *can* be used as intended.

By examining GCF through these three lenses, we are able to show that our work provides a strong foundation for future work. At the architectural level, our system already provides the abstractions needed to support our high level requirements, and address the limitations found in prior work. From a usability standpoint, GCF also benefits from being easy to use, and can be extended by developers to support additional use cases as they are discovered. Finally, from a capability standpoint, GCF already allows devices to form groups and share context in many situations. As mentioned above, the system is still missing some features that either make it more difficult to use than we would like, or prevent it from working in some environments. It is important to note, however, that our system represents a work-in-progress as opposed to a feature-complete product. In the following chapters, we will continue to use GCF to build a wide range of context-aware applications. In doing so, we will gain experience developing opportunistic applications using our framework, which in turn will help us identify opportunities for further improvement.

3.7. Summary

In this chapter, we presented our implementation of the Group Context Framework. First, we introduced our framework's high level requirements and design principles, which we derived from an analysis of the capabilities and limitations of prior work. We then described our framework's main components, and showed how they interact with each other to allow devices to openly request and share context. Through the creation of two prototype applications (GroupMap and GroupHike), we demonstrated how GCF can be used by developers to support grouping and context sharing across all four quadrants of our conceptual model. We then examined our toolkit from an architectural, usability, and functionality standpoint, and showed how our system addresses many, but admittedly not all, of the shortcomings found in existing context-sharing systems.

The work described in this chapter takes a first, but important step towards creating a fully featured and reusable framework that supports opportunistic grouping and context sharing in all of its forms. However, there is still much room for further refinement. Although GCF already contains the core functionality needed to support a wide range of context-aware applications, the architecture presented in this chapter is meant to serve as a foundation rather than a feature complete solution. GCF was created with hooks to allow it to be extended as needed, and in subsequent chapters, we will show how this flexibility allows our framework to be used as a research platform for a diverse range of novel context-aware applications.

4. Exploring A Range of Context-Aware Applications that Can be Built Using Opportunistic Groups

In **CHAPTER 3**, we focused on the *design* of the Group Context Framework, and the identification of the specific features (architectural and software) our framework needs to provide in order to let devices easily form opportunistic groups and share context. Our actual usage of the framework so far, however, has been limited. Although our early work with the GroupMap and GroupHike prototypes shows that GCF is functional and supports grouping across all four quadrants of our conceptual model, these applications were chosen for their pedagogical value as opposed to their novelty. Consequently, we still lack general knowledge of 1) what types of context-aware applications would stand to benefit the most from opportunistic grouping, and 2) how well our framework supports the creation of these applications once they are identified.

In this chapter, we fill this gap by presenting four context-aware systems that were created using GCF. These systems are more fully featured than the example applications presented in **CHAPTER 3**, and allow us to focus on specific application domains and use cases where the ability to form opportunistic groups is either necessary and/or valuable.

The specific systems that we will cover in this chapter are:

1. **Didja:** a context-comparison toolkit that allows devices to form opportunistic groups with other users and/or the environment when they are likely experiencing the same contextual state (*e.g.,* participating in the same conversation, riding the same bus).
2. **Snap-To-It:** a universal interaction tool that lets users opportunistically group with and control physical appliances (*e.g.,* printers, digital projectors) and objects when they take a photograph of them with their smartphone.
3. **Impromptu:** a "just in time" software delivery platform for mobile devices. Here, users share their context (*e.g.,* activity, location) with applications (not installed on their phone), and form opportunistic groups with them to receive contextually relevant information and services.
4. **Bluewave:** an extension to GCF that lets devices openly advertise and share context by programmatically manipulating their Bluetooth name. Bluewave supports use cases where devices only need to interact once or spontaneously, and allows GCF to discover and opportunistically form groups in environments where devices are physically nearby, but not connected to the same broadcast domain (*e.g.,* a local area network).

Collectively, this work addresses our second research question ("How does the ability to form opportunistic groups increase the range of context-aware applications that can be practically created?"). By focusing on multi-purpose systems instead of single-use applications, our work is able to examine a wide range of possible use cases, and show how the ability to form opportunistic groups can improve users' abilities to interact with each other, as well as with other devices and/or services. Additionally, these systems let us stress test GCF and determine what modifications need to be made to our framework in order to support the widest possible set of use cases. Obviously, this investigation cannot provably show *all* of the ways that opportunistic groups can be used in context-aware computing. It should, however, provide a better understanding of which use cases are worth studying in further detail, which in turn provides a foundation for future research.

4.1. Didja: Using GCF To Form Opportunistic Groups with Users and/or the Environment

Our first system, Didja, is a software toolkit that lets devices detect and form opportunistic groups in arbitrary environments [44]. In Didja, devices use GCF to request and share multiple streams of context with each other, such as their audio amplitude, light intensity, and temperature readings. Each device compares these streams with its own

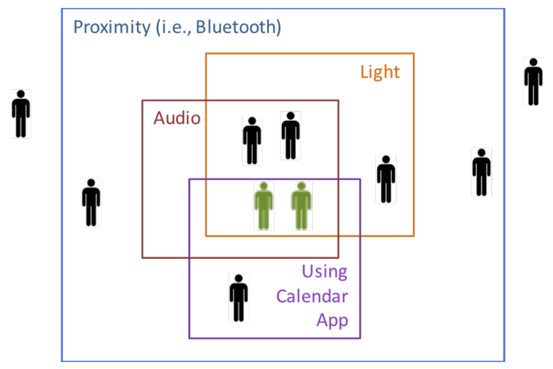


Figure 16. Didja uses GCF to request and receive multiple streams of context from nearby devices. It compares this context with its own sensor readings, and forms groups with devices that match along multiple dimensions. (green; center).

sensor values to determine which device(s) are likely sensing the same thing. Didja then forms an opportunistic group with these devices (Figure 16), and establishes a logical communication channel so that they can transmit information and/or commands to each other. The system also checks to make sure that devices' sensors continue to sense the same context, and adds and/or removes members accordingly.

Our motivation for creating Didja is based on the realization that opportunistic groups oftentimes form when users/devices are experiencing the same contextual state. For example, when users are participating in a meeting (planned or otherwise), it can be inferred that they are in a group because they are hearing the same conversation. Similarly, when users are riding in the same bus, they can be considered members of a *bus group* because they are traveling at the same speed and direction, and are experiencing the same vibrations (*e.g.,* bumps in the road) at the same time. In both examples, the group being formed is too precise to be detected *via* Bluetooth proximity alone, but not so close that members are within touching distance of each other. As a result, it is important for devices to have a more nuanced way of forming groups that does not solely rely on users having to explicitly pair their devices or trade credentials. Didja satisfies this need by automating the group formation process, using contexts that are commonly available on a smartphone. This allows our system to take advantage of a wider range of subtle and explicit groups, without requiring the environment to be instrumented in any way.

Didja builds on a long line of research that has used context comparison as a means of forming associations between devices. To date, researchers have tried to detect groupings of users and/or devices using a single modality (*e.g.,* accelerometers [55,72], near-field communications [34], vision [26], sound [21]) in order to conserve computational resources and battery life. This strategy, however, suffers from a single point of failure, as no single sensor works well in all situations. In contrast, Didja is specifically designed to examine *multiple* contexts simultaneously. This overcomes any one context's limitations, while simultaneously allowing us to make more precise and reliable determinations of group membership. Furthermore, while prior work has used context comparison for specific use cases (*e.g.,* pairing devices [55], exchanging encryption keys [75,79]), Didja provides a *generalizable* way to detect opportunistic groups. When developers use Didja, they can customize it by specifying 1) the types of contexts they want the system to compare, 2) the degree to which these contexts need to match, and 3) the type of comparison that they would like the system to perform. The gives developers the flexibility to customize the toolkit, while still providing users with an automated way of forming groups.

Didja addresses our second research question by allowing us to investigate how opportunistic groups can support seamless interactions between users and/or their surroundings. In the following section, we describe how Didja is implemented using GCF. Afterwards, we present two real-world applications that were created using Didja, and show

how our system's ability to detect groups is useful in a broad range of application domains. Through a series of experimental trials involving both moving and stationary devices, we evaluate Didja's accuracy, and show that the system is able to identify the correct group over 88% of the time. Lastly, we highlight the challenges that we encountered while building Didja with GCF, and show how these difficulties influenced our framework's final design.

4.1.1. System Design

Didja is implemented as a custom arbiter in GCF. As mentioned in **CHAPTER 3**, arbiters are specialized modules within GCF that analyze context capability messages according to a predefined strategy, and tell the framework which device(s) to form a group with. Didja's arbiter offers the same core functionality as GCF's default arbiters (*e.g.,* single-source, multi-source, local only), but differs from them in two important ways:

- First, our arbiter can request and receive context on its own. At startup, Didja calls the Group Context Manager's SENDREQUEST() method, and transmits a multi-source request for context from all devices in communications range. This allows Didja to continually discover new devices and analyze their sensor streams without requiring developers to obtain and deliver this information on its behalf.
- Second, Didja's strategy for forming groups is not fixed, but instead can be configured on a per-application basis. Our system lets applications specify 1) the type(s) of context that Didja should examine, and 2) the degree to which sensor streams need to be similar in order for them to be considered a match. When context from another device arrives, it evaluated by Didja's *context comparator* using a pre-specified evaluation metric (*e.g.*, statistical correlation, proportional analysis). The results from this comparison are then referenced against the current rule set to determine whether or not the two devices are experiencing the same contextual state.

Figure 17 shows an overview of Didja's group formation process. When the Didja arbiter is first instantiated, it automatically forms a group with every device in communications range and subscribes to its context providers. This *candidate* group is hidden from the application, and represents every device that might be experiencing the same context as the user. As context from each of these devices is received and analyzed, Didja forms an *application* group consisting of devices whose sensor streams satisfy the current rule set. This "group within a group" acts as a whitelist. When a developer needs application specific context, they call the SENDREQUEST() method, and tell the Group Context Manager to use the Didja arbiter. The arbiter in turn will ensure that the application only receives context from devices that are members of the application group, and will automatically subscribe and unsubscribe as devices are added and removed from this group, respectively.

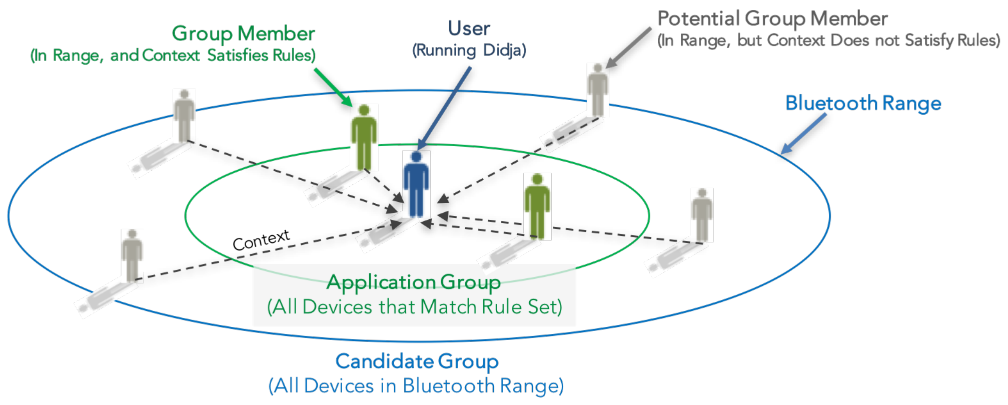
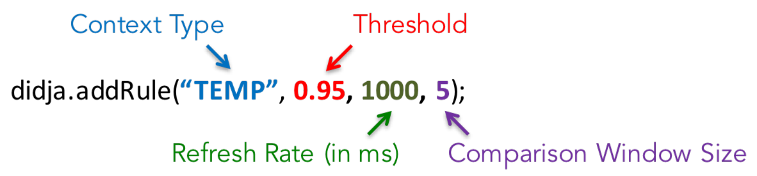

Group Member
(In Range, and Context Satisfies Rules)
User
(Running Didja)
Potential Group Member
(In Range, but Context Does not Satisfy Rules)
Bluetooth Range
Context
Application Group
(All Devices that Match Rule Set)
Candidate Group
(All Devices in Bluetooth Range)

Figure 17. Didja group formation process.

A significant challenge in finding and forming opportunistic groups is that the conditions for grouping are application specific depending upon the size and type of group that is trying to be detected. For this reason, our work with Didja focuses on providing a systematic and automated way of analyzing context that is both easy to use, while still being robust across a wide variety of use cases. We now examine the components of our toolkit, and show how it can be customized to accommodate a range of applications.

4.1.1.1. Rules

Didja uses rules to define the criteria for group membership. For each application, developers provide Didja with a list of context(s) that they want the system to take into consideration when forming a group. These rules are then used by the toolkit to request context from other devices and perform comparisons.



```
didja.addRule("TEMP", 0.95, 1000, 5);
```

Figure 18. Sample Didja Group Membership Rule

An example rule is provided in Figure 18. Each Didja rule consists of four parameters: context type, threshold value, refresh rate, and window size. The *context type* is the GCF identifier that is used to represent a particular sensor or family of sensors. It allows devices to talk about the same context (*e.g.*, "location") regardless of how that context is actually produced (*e.g.*, GPS, cellular triangulation).

The *threshold value* represents how similar two context streams need to be before they are considered to be matching. Each time a context is received, it is analyzed using a predefined metric. The result of this comparison is a numeric value that determines how similar two streams are to one another. The threshold value compensates for individual sensor differences by allowing applications to specify when two context values are considered to be "close enough" to be considered a match.

The *refresh rate* specifies the interval between sensor readings. This value varies depending on the particular context being used, but typically ranges between 500 and 5000 milliseconds.

The *window size* specifies the number of sensor readings that are to be used during the comparison process. Since sensor readings can significantly vary due to noise, they are problematic to compare. Instead, Didja records the last *n* readings for each device, and uses all of these values during comparison. There is a noticeable tradeoff between window size and accuracy. Larger windows are less influenced by erratic sensor readings and provide better estimations of group membership, but require more time to fill and take longer to forget problematic readings. Shorter windows consume less memory and allow for faster determinations of group membership, but are more susceptible to sensor noise. From experience, we found that a window of 10 works well for contexts that do not change often (*e.g.*, light), and that a window of 30 works well for sensors that do (*e.g.,* audio).

4.1.1.2. Context Comparison Techniques

Comparing two contexts to one another is a nontrivial task, as there is no single comparison method that works for all contexts. For this reason, we have designed Didja to compare sensor streams using three techniques: *proportional comparison*, *correlational comparison*, and *exact match*.

Proportional comparison is used to compare numeric sensor values that do not change dramatically over time. Many sensors report the same (or very similar) value (*e.g.*, those that measure light intensity, barometric pressure and temperature) if run repeatedly. For these types of context, a reasonable measure of similarity can be obtained by dividing the average value of each device's sensor readings. To account for individual sensor differences, we also define a noise range for each sensor type. If two sensor values are within this range, we return a similarity of 1.0.

$$similarity = \begin{cases} \frac{\min(a_{avg}, b_{avg})}{\max(a_{avg}, b_{avg})}, |a_{avg} - b_{avg}| > noise \\ 1.0, otherwise \end{cases}$$

While proportional comparison is applicable to a wide range of sensor types, it assumes that any sudden change in a sensor's reported value is caused by noise. As a result, proportional analysis works reliably when the environment is static, but takes longer to detect sudden, intentional changes (*e.g.,* turning off the lights in a room).

The second type of comparison, correlational comparison, is used to compare contexts that significantly vary over time. During our initial trials, we found that audio amplitude readings can significantly vary over the course of a few seconds, especially when users are listening to music or participating in a conversation. For these types of sensors, similarity can be measured by computing the correlation coefficient (*e.g.*, Pearson's R) between the host device and all other devices.

The last type of comparison, exact match, is designed for contexts that are represented as strings or nominal values. Google's Activity Recognition Toolkit [127], for example, reports the user's activity as nominal values (*i.e.,* "in_vehicle," "on_foot", *etc*.). Bluetooth also reports the discovery of nearby devices by listing their IDs. Exact match is the simplest form of comparison. It simply performs a strict string comparison, and returns a value of 1.0 if the values match, and 0.0 otherwise.

We have preconfigured Didja to select an appropriate context comparison method for a wide range of commonly used contexts. However, developers can also direct the toolkit to use a specific comparator, or define their own comparators as the need arises. Through these alternatives, Didja is infinitely extensible, and can be used on a wider range of contexts than what is directly covered in this thesis.

4.1.1.3. Group Formation Strategies

Didja's rule-based architecture provides a straightforward and efficient way to define group membership criteria. However, precisely defining the exact conditions that will lead to a group is extremely difficult when groupings are spontaneous and/or unpredictable. Developers may have a general sense of how their application should form groups (*e.g.,* "group with everyone in the same room") but it can be challenging to translate these vague notions into a robust rule set. Likewise, users may want to define their own custom groups, but lack either the capacity or patience to express these groupings as a set of structured rules.

To mitigate these issues, we have created several developer and user level settings that allow Didja to specify its group formation strategy. While our toolkit still uses rules in order to define the criteria for grouping, it can interpret or modify these rules differently depending upon the currently selected policy. Didja supports four grouping strategies:

- **Match All Rules.** In this mode, Didja only forms a group with devices that satisfy *all* rules. This mode is the strictest, and provides the tightest control over the exact conditions that must be met in order to form a group.
- **Match N.** Rather than require devices to satisfy *all* rules, *Match N* allows Didja to form a group as long as at least *N* rules have been met. While a higher N value will reduce the chance of false positive matches, we allow programmers to specify the value of N to fit their particular application.
- **Max N**. There are many situations where a user or developer wants to form an opportunistic group, but cannot define the exact conditions that will result in a grouping. *Max N* attempts to address this problem by searching for a grouping of devices that satisfies the *most* rules at once.

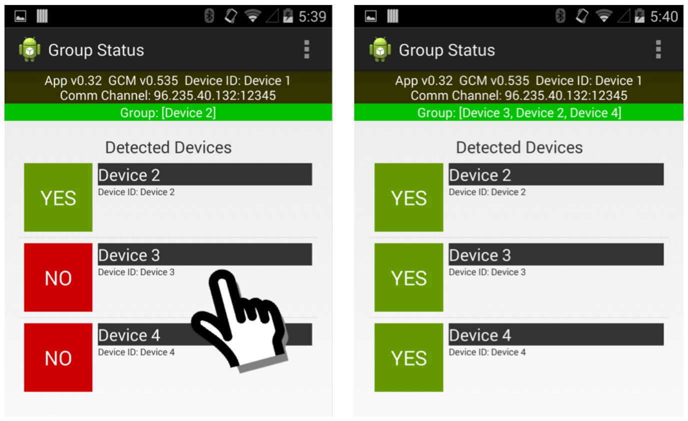


Figure 19. Manual grouping using Didja. In this example, a user selects a device that is known to be part of his or her current group (left). Didja then modifies its current rule set to include other devices that have similar characteristics (right).

- **Manual.** Some opportunistic groupings are so personalized that they simply cannot be described by rules beforehand. For these situations, Didja offers users the ability to build a custom rule set by manually selecting members, and using their sensor readings in order to derive reasonable group threshold values (Figure 19). Users can save these rules for later use, allowing them to create customized "grouping profiles" that complement their daily routine.

4.1.1.4. Scalability

One of the key challenges in designing Didja is finding an effective balance between functionality and scalability. While the current system is technically capable of allowing devices to share a wide range of contexts simultaneously, the overhead to process the results can be significant in an environment with dozens of potential group members.

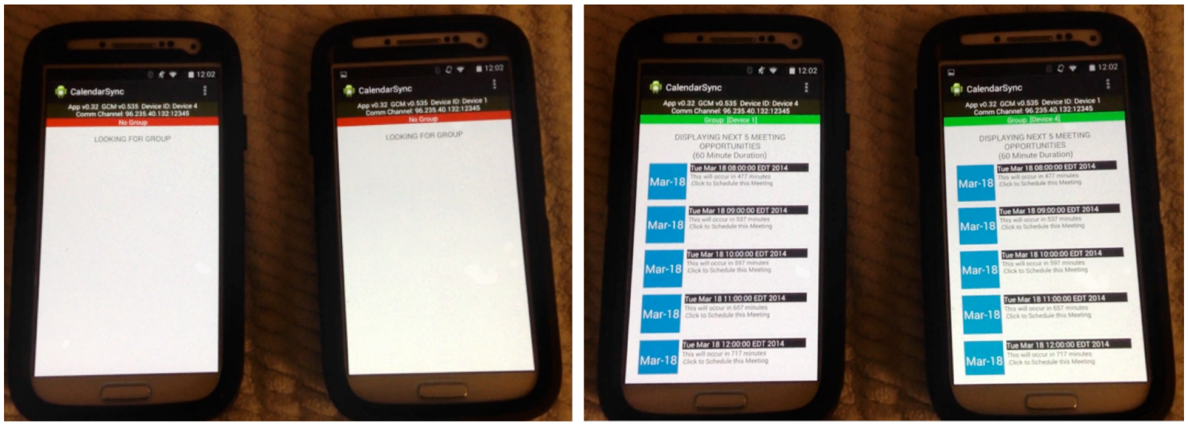
From a pure processing standpoint, there are limits to the number of simultaneous sensor feeds that can be analyzed by a single device. Didja provides two mechanisms to mitigate this problem. First, our system conducts periodic Bluetooth scans of the environment, and only requests context from devices that it detects. This optimization significantly reduces overhead traffic, as the number of devices in the same space is small compared to the total number of devices in a TCP or UDP broadcast domain. Secondly, Didja can be configured to ask for context in stages instead of all at once. In this mode, the toolkit will obtain a single context from each device and evaluate it for similarity. The toolkit then only asks for additional streams from those devices that have satisfied the previous rule. This approach assumes that developers can rank order context from least to most specific. Assuming that they can, however, this approach significantly reduces sensor data from devices that have a low probability of being part of the group.

4.1.2. Example Applications

In this section, we present two applications that take advantage of Didja's ability to detect precise groupings. Our first application, CalendarSync, forms an opportunistic group with users that are participating in the same conversation, and allows them to seamlessly share their schedules so that they identify the next best time to schedule a meeting. The second application, Light Reader, uses Didja to form a group with a smart environment, and allows users to control the lighting by reading a book. Together, these applications highlight our toolkit's versatility, and illustrate the breadth of user experiences that are possible once devices are able to autonomously group with each other.

4.1.2.1. CalendarSync: Forming Opportunistic Groups with Users for *ad hoc* Scheduling

CalendarSync is a scheduling tool that allows users to temporarily share calendar information with one another. When users first turn on the application, they only see their own schedule. When two or more CalendarSync users are near one another, however, the applications form an opportunistic group and automatically share calendar information. The application then presents users with a list of times when they are available to meet again (Figure 20).


CalendarSync
No Group
LOOKING FOR GROUP
DISPLAYING NEXT 5 MEETING OPPORTUNITIES (60 Minute Duration)
Mar-18

(a) Using Didja to Detect an Opportunistic Group (b) Sharing Calendar Schedules to Find Available Times to Meet

Figure 20. CalendarSync screenshots. Smartphones use Didja to compare sensor readings and detect potential group members (a). The resulting group can then trade user schedules, and display a left of the next best time(s) for everyone to meet (b).

CalendarSync demonstrates how opportunistic grouping can improve the usability of existing calendar systems. While there are numerous tools that allow users to share their schedules with one another, they require users to manually grant permissions to others before they can see it. Furthermore, these systems tend to share an *entire* calendar with users, which can lead to privacy issues. Because of these drawbacks, traditional calendar sharing is only used in cases where the need to share calendars is either so great and/or frequent that it outweighs the inconvenience. CalendarSync eliminates the need for access lists in many cases by granting users access to each other's calendars on a need-to-know basis. Users only see each other's calendars when they are grouped, and lose access when they are no longer grouped. Through this approach, CalendarSync provides a capability that is missing in comparable applications, and allows users to easily share schedules, even if they are meeting for the first time.

Figure 21 provides an architectural diagram of the CalendarSync system. For this application, we created a CALENDARPROVIDER (context type "CAL") that can examine a user's personal calendar, and return a list of times when she is available for a meeting. We then added this context provider, as well as the Didja arbiter to the CalendarSync app, and directed the application to request Calendar context from other users *via* the following GCF call:

```
gcm.sendRequest("CAL", 60000, DIDJA, new String[] {"startTime=0700", "endTime=1900"});
```

Each time the CalendarSync receives a context capability message for calendar data, Didja checks to see if the device that sent the message is a member of the current opportunistic group. If so, Didja directs the GCM to subscribe to that device's calendar context provider. The application then compares the incoming calendar information to the user's schedule, and derives a list of available times.

CalendarSync was designed to facilitate information sharing between two or more acquaintances when they meet in a common area (*e.g.,* a water cooler). To support this type of interaction, we configured Didja so that it only forms a

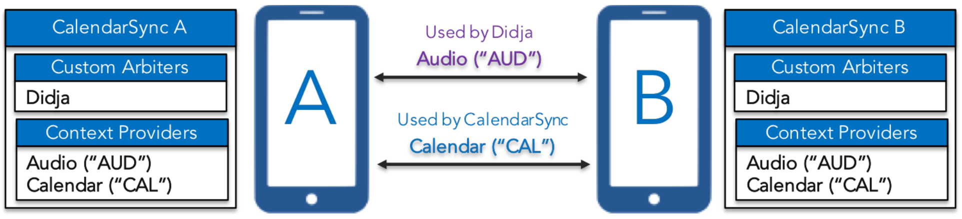


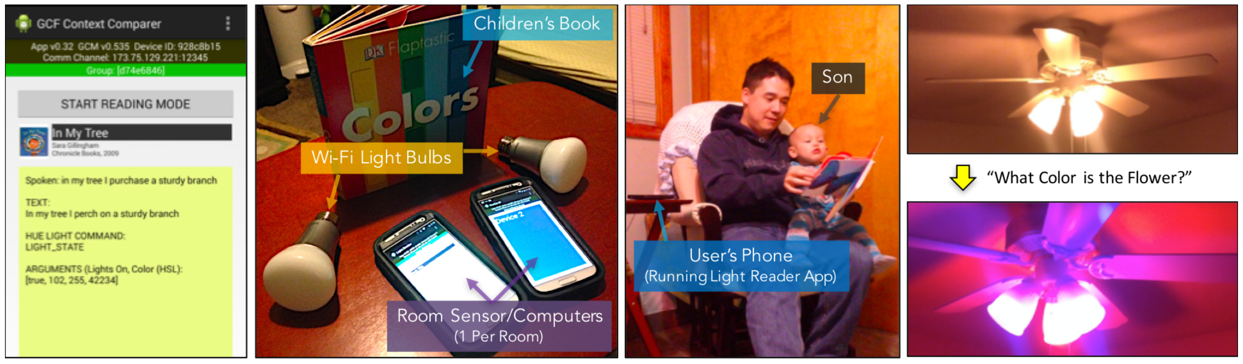
Figure 21. Architectural diagram of the CalendarSync system.

group with devices that are 1) in close proximity (as determined by Bluetooth), 2) displaying the user's calendar, and 3) hearing the same audio (correlation $\geq$ 0.5). Given the limited numbers of users that currently have access to CalendarSync, examining three types of context may seem counterintuitive when Bluetooth alone can technically suffice. However, this functionality is needed when multiple CalendarSync instances are in Bluetooth range, as it reduces our system's likelihood of producing false positives.

4.1.2.2. Light Reader: Forming Opportunistic Groups with Smart Environments for Interactive Entertainment Experiences

Whereas CalendarSync is designed to form groups with users, the Light Reader application shows how Didja can be used to form groups with an instrumented *environment*. Inspired by the authors' own experiences reading books to their children, the Light Reader system consists of three components (): 1) a set of Wi-Fi controllable light bulbs, 2) a series of sensors (*i.e.*, smartphones) that are installed in each room and control a different set of light bulbs, and 3) a mobile application, which is running Didja. When a user turns on the Light Reader app (Figure 6b), the application compares sensor readings from each room-based computer and forms an opportunistic group. Then, as the user reads out loud (Figure 6d), the application uses a speech-to-text converter to cross-reference what the user is saying against a database of known titles. When a match is found, the application retrieves the light configuration settings for that particular book/passage and transmits them to the room-based computer, altering the lights accordingly (Figure 6e).

Light Reader is similar to Disney's StoryLight [128] in that it allows users to dynamically adjust the lighting of a room based on the contents of a story. However, our system's ability to form opportunistic groups improves upon the former from a usability standpoint. While StoryLight only works with a dedicated light bulb in one room, our system's



App v0.32 GCM v0.535 Device ID: 928c8b15
Comm Channel: 173.75.129.221:12345
Group: [d74e6846]

In My Tree
Sara Gillingham
Chronicle Books, 2009

Spoken: in my tree I purchase a sturdy branch

TEXT:
In my tree I perch on a sturdy branch

HUE LIGHT COMMAND:
LIGHT_STATE

ARGUMENTS (Lights On, Color (HSL):
[true, 102, 255, 42234]

(a) Light Reader App (b) Physical Components (c) User Reads Story Out Loud (d) App Directs Room Computer to Alter Light Color/Intensity

Figure 22. Light Reader system. We created a custom mobile phone application (a) that interfaces with a series of Wi-Fi connected light bulbs and mobile sensors deployed throughout a house (b). When the user reads a book out loud (c), the app uses Didja to form an opportunistic group with the devices in his room. The app then modifies the color and intensity of the lights to match the contents of the story (d).

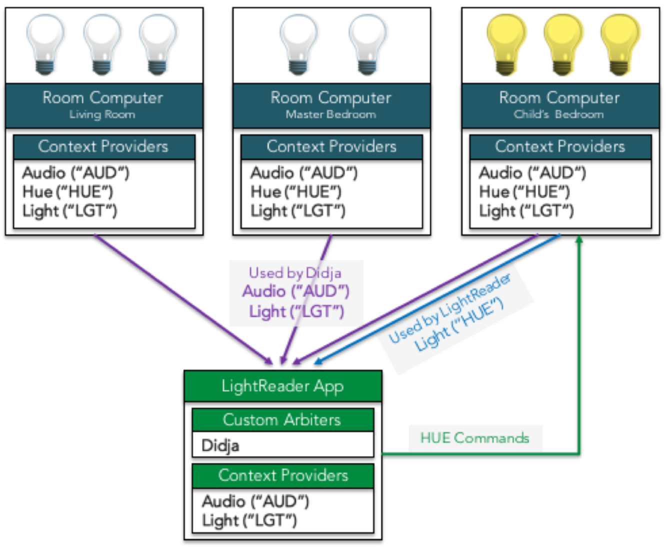


Figure 23. Light Reader System Architecture.

ability to form opportunistic groups allows it to control lighting of whatever room the user is location. In preliminary deployments, this feature was found to be especially valuable, as parents do not want to go to a specific part of the house in order to read a book to their children. Additionally, since our system is able to continuously monitor context and adjust group membership as needed, our system also allows parents to move about the house while reading a story. This allows them to start a story in one part of the house, and seamlessly carry that experience to another room. In order to take advantage of these features, our current implementation assumes that a house contains sensors and Wi-Fi controllable lights. However, such products are already commercially available, and should become more commonplace with time.

Figure 23 shows how we implemented Light Reader using GCF. For this system, we installed a series of Phillips Hue light bulbs [129] and computers (*i.e.,* smartphones) in each room of our test house, and created a custom context provider (*i.e.,* HUEPROVIDER; context type "HUE") that can return the URL(s) to each light bulb's REST API. We then registered this context provider with each room-based computer, and configured each one so that they only return the URL(s) for the lights in that particular room. For example, the context data message from the guest bedroom's HUEPROVIDER looks like the following:

```
{
 "deviceID": "GuestBedroom",
 "contextType": "HUE",
 "destination": ["Device A"],
 "payload":
 [
 "http://192.168.1.20/api/home/lights/4/state",
 "http://192.168.1.20/api/home/lights/5/state"
 ],
 "messageType":"D",
 "version":1
}
```

Figure 24. Context data message from a HUEPROVIDER deployed in our test home's guest bedroom. Each provider is configured for a single room in a smart environment, and provides the URLs to each light bulb's REST API (green; underlined). The Light Reader app can use these URLs to control the lights for that particular room.

When users start the Light Reader app, the software requests context from all HUEPROVIDERS in the house *via* the following GCF call:

```
gcm.sendRequest("HUE", 60000, DIDJA, new String[0]);
```

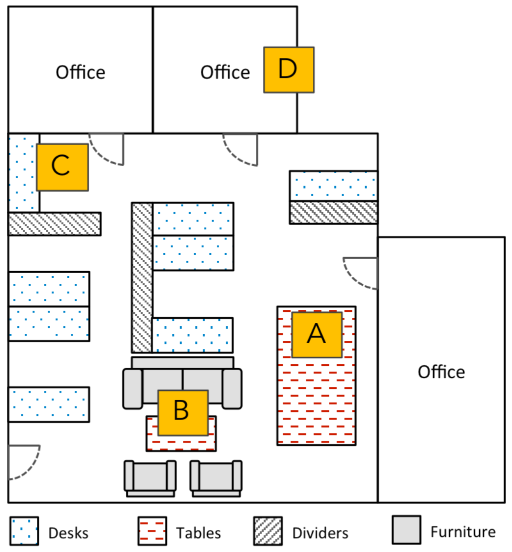
Didja ensures that the application only subscribes to the HUEPROVIDER that is associated with the room that they are currently in (*i.e.,* the current opportunistic group). The application then receives the URL(s) for that room's lights, and can issue the appropriate HTTP POST operations (in accordance with the Hue API [130]) to control their color and intensity.

Since Light Reader is designed to complement the traditional experience of reading a storybook, we have configured Didja so that it forms a group with devices that 1) are in proximity of one another, 2) hear the same audio (correlation $\geq$ 0.5), and 3) experience the same lighting conditions (light intensity $\geq$ 0.8). As was the case with CalendarSync, the use of multiple contexts allows for higher degrees of accuracy when determining which devices are part of the group. From our experience, the rate of false positives using either audio or light readings alone can be quite high, as bedrooms are typically grouped together, and can cause audio sensors to hear the same thing (especially if the doors are open), and the use of similar overhead lighting fixtures (*e.g.,* fans, lamps) can confuse light sensors. By using both, we significantly reduce the chance of error.

4.1.3. ACCURACY EVALUATION

In addition to the above prototypes, we conducted two experiments in our lab during normal business hours to evaluate Didja's accuracy in identifying opportunistic groups. While it may seem biased to use our work area as the subject of these experiments, our lab has several features that make it reasonable to study:

1. There are few walls, allowing sound to easily travel.
2. Students bring in their own lamps, creating unique lighting conditions throughout the room.
3. It contains two common meeting areas as well as a private office, allowing it to host multiple meetings/conversations simultaneously.



Experimental Rules

Sensor	Refresh Rate	Compare Method	Window	Threshold
Light	1000ms	Proportion	10	0.8
Temp	1000ms	Proportion	10	0.95
Accelerometer	500ms	Proportion	10	0.95
Microphone	250ms	Correlation	30	0.5

Figure 25. Setup used for our experimental evaluation, showing the positioning of devices in our test environment (left), and the rules we used to configure Didja (right).

Table 6. Comparison of grouping strategies from device 1's perspective (all devices stationary).

Strategy	Grouped w/Device 2	Grouped w/Device 3	Grouped w/Device 4	% False Negative	% False Positive
Match All Rules	75.6%	0%	0%	24.4%	0%
Match N Rules (N≥1)	100%	100%	100%	0%	100%
Match N Rules (N≥2)	100%	99.9%	100%	0%	100%
Match N Rules (N≥3)	99.6%	52.4%	23.3%	0.4%	57.1%
Find Best (Max N)	100%	8.9%	2.7%	0%	10.4%

Table 7. Comparison of grouping strategies from device 1's perspective (device 1 moves every 5 minutes).

Strategy	Found Exact Group	False Negative	False Positive
Match All Rules	70.4%	29.1%	1.7%
Match N Rules (N≥1)	0%	0.5%	100%
Match N Rules (N≥2)	0.6%	2.5%	99.3%
Match N Rules (N≥3)	55.5%	7.2%	39.4%
Find Best (Max N)	88.2%	3.8%	11.8%

Figure 25 (left) shows a bird's eye view of our lab. For the first test, two devices represented an opportunistic group, and were placed on a conference table (location A). A third device was placed nearby on a coffee table (location B), and a fourth was placed in an empty office (location D). We then instructed the toolkit to request and compare contexts according to the rules in Figure 25 (right), which we derived empirically.

Although our focus is on quick and spontaneous groupings of devices, we allowed Didja to analyze sensor data for one hour in order to see how well the toolkit performs under extended use. During this first experiment, devices were not touched or moved, and remained within Bluetooth range.

Over the course of the first evaluation, Didja performed 12,000 sensor comparisons. Table 6 provides a summary of Didja's grouping performance as seen from the perspective of device 1. We define a *correct group* as one that only includes device 2. Additionally, we define a *false negative* as a group that does not include device 2, and a *false positive* as a group that includes either device 3 or device 4.

Our results show that using a single sensor works with varying degrees of success. The light sensor was by far the best in terms of accuracy, matching with device 2 over 99% of the time, and with devices 3 and 4 less than 0.5% of the time. Meanwhile, thermometer and accelerometer readings were the least discerning, and matched with all three devices regardless of their location. When comparing audio data, device 1 frequently matched with device 2 (75.9%), and occasionally matched with devices 3 (52.2%) and 4 (23.1%). These results were not surprising, however. While we were recording, we observed several sidebar conversations that occurred either at the back of the room (between devices 1, 2, and 4) or at the opposite corner (between devices 1, 2, and 3). Additionally, there were several periods of extended silence, which can trick the audio sensor into believing that it should match with other devices when there is no associated group.

After each sensor comparison, we had Didja analyze its group membership using the *Match All Rules, Match N Rules*, and *Max N* strategies. As expected, Match All Rules proved to be the strictest, and allowed device 1 to only group with device 2. However, this was accomplished at the cost of increased false negatives, thus making this strategy useful in situations where the penalty of forming an incorrect grouping is significant. Match N Rules, on the other hand, minimizes the rate of false negatives, but at the cost of dramatically increasing the rate of false positives. This makes this strategy useful in applications where precision is useful, but not necessarily required. Interestingly, Max N offered a mix of accuracy and precision. By allowing N to fluctuate at each time step, device 1 is able to consistently group with device 2 while keeping false positives and false negatives low.

At first glance, these results appear to negate the need for Didja, as they suggest that highly accurate groups can be formed using a single sensor. Yet while the light sensor proved to be highly discerning, it is important to remember that this is only the case for *this particular arrangement* of devices. For our second evaluation, we positioned devices 2, 3, and 4 at locations A, B, and C, respectively. We then moved device 1 between these locations every five minutes for an hour. This setup allows us to test Didja's ability to form opportunistic groups, as device 1's group constantly changes as it moves between locations.

The results from our second experiment are provided in Table 7. Since device 1's group is constantly changing, we simply report the amount of time that it found the correct group, whether that is with device 2, 3, or 4. Similar to the previous experiment, we found that Didja is able to accurately and reliably form precise groupings. Although the toolkit does perform slightly worse than in our first trial, most of the inaccuracies occurred as the device was moving from one location to another (*i.e.,* when it was technically not part of any group). More importantly, we also found that using a single sensor to form groups was less effective in this scenario. While the light sensor again proves to be the best single sensor, its accuracy dips to 84% with a 15% false positive rate. Max N outperforms the sensor on both metrics, thus showing how our approach provides an accurate *and* consistent way to form groups.

4.1.4. Lessons Learned

Didja is the first significant context-aware system that we have created using the Group Context Framework. As such, the system has provided us with many critical insights regarding GCF's overall functionality and ease of use. In this section, we highlight two issues that we encountered while creating Didja, and show how they influenced our framework's final design.

The Need for Directed Context Requests. One of Didja's key features is the ability to ask for context in stages. For example, if there are numerous devices connected to a broadcast domain (*i.e.,* a subnet), Didja will first issue a multiple-source request from all devices for a single type of context (*e.g.,* temperature). The system will then identify the specific device(s) who are most likely to be part of the final opportunistic group, and only request the next type of context from them. By repeating this process for each context type, Didja minimizes the amount of unnecessary context that is received and processed, without reducing the system's overall accuracy.

Implementing this functionality using GCF proved to be difficult. As mentioned throughout this thesis, our main goal with GCF is to provide devices with a streamlined and *automated* way to detect form groups that minimizes the need for *a priori* information. Consequently, early versions of the framework did not let developers explicitly state which devices they would like to receive context from, even when these devices *were* known (or could be identified) in advance. As an initial workaround, we created a custom "whitelist arbiter" that would only form groups with devices whose GCF IDs were programmatically specified in advance. However, this solution became cumbersome to use, as it required us to register and manage multiple whitelists (one for each context type) in order to achieve the desired effect.

Through Didja, we realized that there are times when developers and/or applications need to be able to request context from a specific subset of devices in an environment. To support this, we modified GCF's context request message to include a *destination* field. We then modified the Group Context Manager's sendRequest() method so that users can populate this field when making their request, as shown in Figure 26.

Destination Field

```
gcm.sendRequest("TEMP", 5000, new String[] {"Device A", "Device C"}, SINGLE_SOURCE, 0.2, 0.2, 0.2, 0.2, 0.2, new String[0]);
```

Figure 26. A sample request for context using a modified version of GCF's sendRequest() method. By providing a list of device IDs (blue), developers can specify which devices are able to see and respond to any particular request.

When a device receives a context request message with a nonempty destination field, GCF examines it to see if the device's ID is included within. If so, the communications manager forwards the message to the Group Context Manager to be processed; otherwise, the message is discarded. Alternatively, if the message's destination field is empty or null, the communications manager assumes that the message is intended for everyone. This lets GCF continue to broadcast messages to all devices in communications range, and discover group members "on the fly."

This solution has two benefits. First, it lets developers specify the device(s) they want to group with on a *per context* basis. This allows GCF to support applications where users and/or developers already know which device (or range of devices) they want to exchange information with, as well as situations where these devices are not known *a priori*. Furthermore, since this solution only affects GCF's communication layer, developers can combine it with GCF's default arbiters to form more specialized groups (*e.g.,* "request location data from any of the following devices that I own"). This prevents developers from having to create their own arbiters in many situations, while still providing them with a robust interface for requesting context.

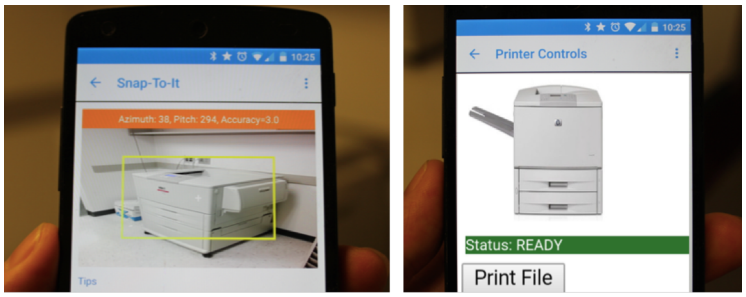
Using Bluetooth to Improve GCF's Situational Awareness. Another problem that we encountered while developing Didja was that we lacked a reliable way to tell which devices were physically nearby. Early versions of GCF relied solely on network broadcasts in order to determine which devices are in communication range. However, the range of a broadcast domain can largely vary depending upon the environment and network topology. In field trials, we found that this could cause Didja to request, receive, and analyze context from devices that are clearly not a part of the user's current group (*e.g.,* a device located on a different floor of the same building).

As mentioned in our system description, we overcame this problem by using Bluetooth to determine which devices were within 15-30 feet of the user. However, we still needed to devise a way to convert a device's Bluetooth name (*e.g.,* "Adrian's Phone") to its GCF ID (*e.g.,* "Device A") so that we could request context from it. While we initially considered having devices pair over Bluetooth to exchange their IDs, we eventually decided to let Didja programmatically set the device's Bluetooth name to contain its GCF ID. This eliminated the need for pairing altogether, while providing our system with the information that it needed to selectively request context.

Initially, this "hack" was implemented specifically for Didja, and was not intended to be incorporated into GCF. As we continued developing context-aware applications, however, we discovered that the ability to detect nearby devices is oftentimes needed to find and form opportunistic groups, especially in environments which do not have a local area network. This realization would eventually inspire us to create a dedicated add-on module for GCF that allows devices to easily discover and share information using their Bluetooth ID (refer to section 4.4 for more details).

4.2. Snap-To-It: Using GCF to Form Opportunistic Groups with Appliances, Objects, and the Internet-of-Things

Our second system, Snap-To-It, is a novel interaction tool that lets users easily select and interact with the ubiquitously distributed appliances in their environments [45]. In this system, users run a custom smartphone application that lets them take photographs of the devices (*e.g.,* printers, digital projectors) or objects that they would like to interact with. Our application then broadcasts this image (along with the user's location and device orientation) across a local area network so that appliances and/or software proxies can analyze it. When an appliance receives a photograph, it 1) determines that the user is nearby and pointed in the correct direction, and 2) uses image recognition algorithms [76] to analyze it. If a photo matches an appliance with a high degree of confidence, the app connects to it and renders a custom interface (Figure 27). Otherwise, the user receives a list of the most likely candidates based on their context, and is asked to select from them.



Taking a Photograph Getting an Interface

Figure 27. A user takes a photograph of the appliance that they want to control (left). Snap-To-It shares this image (along with the user's location and device orientation) with nearby appliances. The user's device connects to the appliance that best matches the image, and receives a custom user interface (right).

Snap-To-It contributes to our second research question by showing how the ability to form opportunistic groups can facilitate one time or spontaneous interactions with their environments. Currently, interacting with new or unfamiliar appliances is a well-known challenge in the Ubicomp community [6,54,78,83,96]. For example, if users want to use a networked printer for the first time, they have to know its IP address and install the correct drivers. Similarly, if a user wants to play music on a Bluetooth speaker, they must first know how to set the appliance to be discover, and *then* find and pair with it on their phone. While this level of effort is acceptable for appliances that we use regularly, it makes one-time or spontaneous use impractical. This discourages users from interacting with new or unfamiliar appliances, and can even force them to go out of their way to seek alternative solutions (*e.g.,* asking a friend or stranger who is already paired with a printer to print a document on their behalf).

To date, researchers have explored several solutions for users to control appliances from their mobile device [8,85,95,113,122]. Snap-To-It builds on this work in three important ways:

1. First, Snap-To-It broadens our understanding of the types of opportunistic interactions users would like to perform using their mobile devices. Prior to developing Snap-To-It, we conducted a probe with 28 participants to see how users would like to interact with the appliances in their surroundings. Through this probe, we identified six categories of general use cases. Snap-To-It's design is directly informed by these findings, providing a research platform that shows 1) what types of interactions are *possible* using a mobile device, and 2) what types of interactions are actually *desired* by end users.

2. Secondly, Snap-To-It offers a novel architecture that supports opportunistic grouping with a wide range of appliances. Rather than require the user to download a list of nearby appliances [54] or connect to a well-known server [14], our system uses multicasting to identify an appliance based on its photo. This makes it particularly useful when users are visiting a location for the first time, and need to use an appliance once or spontaneously. Additionally, while prior work has shown that mobile devices can be used to select and interact with hardware [90,105] and software [18], they have typically focused on one *or* the other. In contrast, Snap-To-It compares user-taken photographs against stock images and real-time screenshots. This lets it support both types of appliances using a single interaction technique. Finally, since Snap-To-It works with any photograph, our system can be used to make non-computation objects (*e.g.,* signs, maps) selectable. This dramatically increases the ranges of appliances, applications, and information that users can interact with without requiring each one to have an embedded computer.
3. Third, Snap-To-It provides a developer-friendly way to allow appliances to be selectable *via* camera. Our system's middleware (which uses GCF) automatically processes incoming photos, establishes connections with user devices, and delivers arbitrary HTML/JavaScript based user interfaces. This allows developers to incorporate Snap-To-It's functionality into appliances without having to implement their own photo recognition. Additionally, our middleware can act as a software wrapper for existing appliances and applications. This greatly expands the range of appliances that can utilize our system without forcing users or site administrators to install additional hardware.

In the following sections, we describe our probe, and show how user responses have helped us better understand the types of opportunistic interactions that users would like to form in their everyday lives. Afterwards, we describe Snap-To-It's architecture, and present four prototypes that highlight its capabilities. Through three studies, we show that Snap-To-It is sufficiently accurate to be deployed in real-world environments. Finally, we address issues such as usability, responsiveness, and hardware requirements, and discuss how our experiences creating Snap-To-It have helped us refine GCF's design.

4.2.1. Technology Probe and Derived Requirements

With Snap-To-It, we aim to let users quickly select and interact with any object in their physical environment. Yet while prior work has shown that this two-step interaction model supports many use cases, they focus on the challenges associated with connecting to devices and receiving and/or rendering custom interfaces [54,85]. Consequently, we still lack knowledge of 1) the *types* of devices that people would like to interact with in an opportunistic fashion, and 2) the *range* of interactions that they would like to perform, both of which are necessary to create a system that will actually be useful and usable to users.

To address this gap, we conducted a technology probe in which we asked users to provide us with a "wish list" of appliances that they would like to interact with using their mobile device(s). For this study, participants installed a custom Android app on their smartphone that allowed them to take photographs of devices and annotate how they would like to use them. They then spent a week taking pictures as they went about their normal routine.

For this probe, we recruited 28 participants (14 males, 14 females; 19 to 60 years old; consisting of a mix of novice and expert users), all of whom use a smartphone on a daily basis. Each participant was paid $20 regardless of how many photographs he/she submitted. In addition, we encouraged users to photograph as many devices/objects as they wanted, regardless if the technology or infrastructure needed to control them currently exists.

Through this process, we collected a total of 195 photographs. Each photo was categorized according to 1) the interaction being performed, and 2) the technologies needed to support them. This information was then directly used to identify Snap-To-It's general use cases and derive critical technical requirements. Although the number of

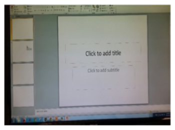
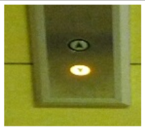
Use Case Category	Motivation	Sample Participant Responses and Appliances
Quick Control	Connect and use a nearby device.	"I want to print"  "Changing slides in a powerpoint presentation"
Long Distance Operation	Operate a device from another (remote) location.	"I would like to . . . heat my food before I reach home."  "Hold elevator when I'm in my apartment"
Transfer Content	Allow users to post and receive data.	"Extend my phone screen to a bigger display." "If I took a photo of a map, then a 3D map appears, it would be useful."
Secure Transactions	Be able to securely send information to a device.	"Access doors to let me in" "Enter choice thru my phone N pay through my phone"
Increase Intelligibility	Improves users' knowledge of how a device operates.	"I am not so sure what these button controls" "How does this work?"
Share Preferences	Allow settings/preferences to be shared across devices.	"Preset my treadmill" "Set my favorite TV channel."

Figure 28. General use cases for Snap-To-It, as derived from user responses from our technology probe.

photos submitted by participants varied, our analysis found that each identified roughly the same number of general use cases (mean=2.1, SD=1.1). This suggests that our findings were not overly influenced by any one participant.

4.2.1.1. Use Case Categories

Participants had a wide range of ideas as to how they wanted to interact with appliances using their mobile phone. Based on the results from our probe, we have identified six general use case categories.

Nineteen participants (68%) wanted to use their mobile phones to quickly interact with new or unfamiliar appliances (Figure 28, Row 1). The majority took photographs of office appliances such as printers, projectors, and multimedia controls. Others took photos of specialized equipment such as laser cutters and 3D printers. In each case, users wanted to use the appliance without installing software/drivers.

Another popular use case was to control appliances from afar (Row 2). Twenty participants (71%) took pictures of household/office appliances, vehicles and industrial equipment, and stated that they wanted to be able to control these devices from anywhere. In some cases, it was purely for added convenience (*e.g.,* "[I want to] turn off the lights when I'm in bed"). Others viewed their phone as a more hygienic way to interact with public objects (*e.g.,* lights, toilet handles) without having to physically touch them. Finally, several participants noted that being able to activate controls from a distance would be particularly helpful to disabled individuals. We found this to be an unexpected, but interesting use case that we wanted to support.

While we anticipated that participants would want to use their smartphones as a remote control, participants also identified several other ways to interact with appliances. Eight participants (29%) wanted to use their mobile device as a way to upload and download content (Row 3). Several users took pictures of public displays and televisions, and stated that they wanted to be able to send or receive content (*e.g.,* push a PowerPoint presentation, pull a closed-caption feed). Interestingly, the idea of pushing/pulling content was not limited to computer devices. Several participants took photos of disconnected objects such as campus maps and public museum displays, and stated that they wanted to extract the information represented by the object using their phone (*e.g.,* converting a picture of a map into a digital version). In these cases, participants viewed these objects as "physical hyperlinks", and were interested in being able to use their mobile phone as a way to access an object's digital representation.

Seven participants (25%) wanted to use their phone to perform secure transactions (Row 4). Several of them took pictures of locked doors and computers, and noted how it would save them time and effort if they could authenticate to these devices while approaching them. Others took photographs of payment systems (e.g., vending machines), and noted that it would be convenient if they could make purchases electronically.

A fifth use case identified by four participants (14%) was to learn more about a particular appliance (Row 5). One participant photographed a light switch, and wrote: "I am not so sure what these buttons control." Others took pictures of fire alarms, defibrillators, and household appliances, and asked for 1) what the appliance does, and 2) how to use it.

Finally, two participants (7%) wanted to use camera-based selection to easily transfer settings and preferences between appliances (Row 6). One participant took a photo of a treadmill, and said that he wanted to easily carry over his exercise preferences (*e.g.,* speed, incline) to another machine during his next workout session. The other took a photograph of a public television, and stated that he wanted to quickly find and tune to his favorite channels without having to search for them. In both cases, users wanted their phone to remember their past interactions. This would allow them to seamlessly transfer these preferences to new appliances.

Collectively, these responses highlight the importance of being able to form opportunistic groups with appliances. While only 18% of our participants stated that they use more than three devices daily, our probe reveals that 1) all of them wanted to interact with more devices than they currently do, and 2) that this desire extends beyond simple remote controls. This emphasizes the need for a more versatile way of interacting with appliances than is currently available.

4.2.1.2. Functional Requirements

Our probe shows that users seek a simple but versatile way of interacting with a wide range of objects/appliances. Based on their responses, we have identified the following functional requirements:

- **R1: Support Photo-Based Selection.** One of our main goals with Snap-To-It is to let users select appliances using a simple, well known interaction technique. Consequently, while other methods (*e.g.,* QR codes) may be necessary at times, camera-based selection should be used whenever possible.
- **R2: Interact with Software and Hardware.** As mentioned previously, current systems typically focus on interacting with hardware or software. Our technology probe, however, shows that users frequently want to

interact with both types of appliances. For this reason, tools like Snap-To-It need to support both hardware and software interactions through a single interface.

- **R3: Interact with Appliances from Afar.** As shown in Figure 28, Row 2, users want to interact with appliances that are out of sight. This points to the need to remember previously used appliances (*i.e.,* a "favorites list") so that remote operation is possible.
- **R4: Render Complex Interfaces.** Many of the use cases identified in Figure 28 call for complex user interfaces (*e.g.,* interactive maps, real-time closed-captioning feeds). These interfaces are more dynamic than the button/slider interfaces supported by existing remote control systems, and illustrate the need to be able to render any arbitrary UI at runtime.
- **R5: Support Expandability.** The final insight gained from our probe is that users use new technologies in unexpected ways. While we know that there are several obvious use cases that our system needs to support (*e.g.,* sending commands to remote appliances), our participants have also identified a number of use cases that are important to them, such as the ability to perform remote transactions and save/transfer settings. These findings highlight the real world challenges of creating a universal interaction tool, and that systems such as Snap-To-It need to be flexible enough to accommodate these use cases as they are discovered.

In the following section, we show how Snap-To-It's design is directly influenced by our technology probe. Our system satisfies all of the requirements described above, and provides a middleware that makes it easy to add our system's functionality to new or existing appliances. This allows it to support all six use case categories while still leaving room for future expansion.

4.2.2. System Design

The Snap-To-It system consists of two components (Figure 29):

1. A **mobile application** (left), which runs on the user's smartphone or tablet.
2. A series of **remote service providers** (RSP; right), each of which controls one or more appliances, and resides either on the appliance itself or a designated proxy (*i.e.,* a server).

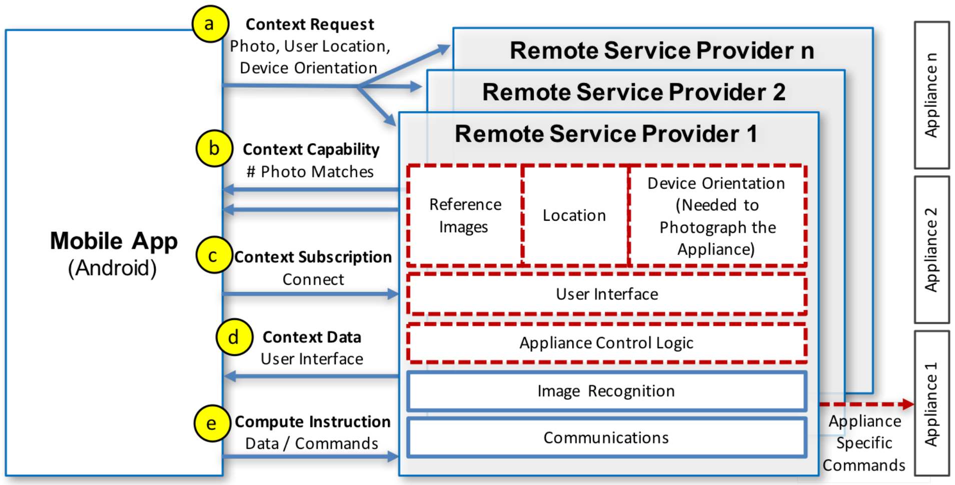
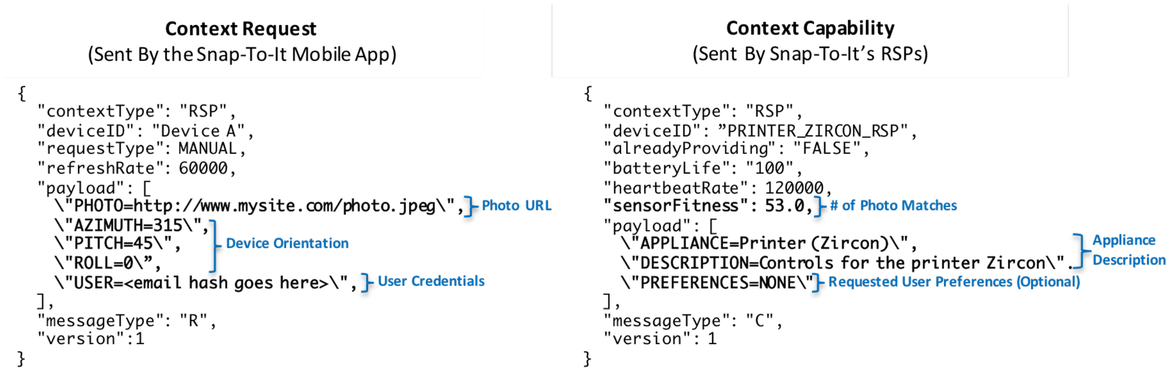


Figure 29. Snap-To-It high level architecture, and communications flow (a-e). Solid boxed components (blue) are provided and/or automatically handled by our middleware. Dotted components (red) are appliance specific, and are provided by developers or site administrators.


Context Request
(Sent By the Snap-To-It Mobile App)

```
{
  "contextType": "RSP",
  "deviceID": "Device A",
  "requestType": MANUAL,
  "refreshRate": 60000,
  "payload": [
    \"PHOTO=http://www.mysite.com/photo.jpeg\",
    \"AZIMUTH=315\",
    \"PITCH=45\",
    \"ROLL=0\",
    \"USER=<email hash goes here>\",
  ],
  "messageType": "R",
  "version":1
}
```

Context Capability
(Sent By Snap-To-It's RSPs)

```
{
  "contextType": "RSP",
  "deviceID": "PRINTER_ZIRCON_RSP",
  "alreadyProviding": "FALSE",
  "batteryLife": "100",
  "heartbeatRate": 120000,
  "sensorFitness": 53.0,
  "payload": [
    \"APPLIANCE=Printer (Zircon)\",
    \"DESCRIPTION=Controls for the printer Zircon\".
    \"PREFERENCES=NONE\"
  ],
  "messageType": "C",
  "version": 1
}
```

Figure 30. Example Snap-To-It context request (left) and context capability messages (right).

Both components were created using GCF. The RSP is a context provider (context type = "RSP") that can analyze photos, deliver interfaces, and transmit remote commands to the appliance(s) they control. The mobile app uses GCF to dynamically request and receive interfaces from these RSPs. When a user takes a photograph of an appliance, our app transmits a *context request* message that contains 1) a URL to the photo, 2) the user's current location, and 3) the smartphone's orientation (Figure 29a). RSPs on the same subnet can then use this information to determine if the user is looking at them, and respond with a *context capability* message that contains the number of photo matches, and a brief text description (Figure 29b). When the user or mobile app selects an appliance, GCF sends a *context subscription* message to the RSP (Figure 29c), and receives one or more *context data messages* that contains the user interface (Figure 29d). The user can then interact with the interface to send custom instructions to the RSP (Figure 29e), which are converted into appliance specific commands (*e.g.,* turn on/off, change channel).

This section describes Snap-To-It's important technical details. First, we show how our system recognizes appliances. Afterwards, we show how it renders interfaces, shares preferences, authenticates users, and supports extensibility.

4.2.2.1. Selecting an Appliance *via* the Mobile App

Snap-To-It offers two ways to select an appliance. The primary way is by using the mobile device's camera (**R1**). Each time a user takes a photo of an appliance, our app automatically uploads a 640 × 480 JPEG version of it to a cloud server (which simply hosts the photo for RSPs to access), and records the phone's location and orientation (azimuth, pitch, roll). It then transmits a context request message (Figure 30, left) containing the photo URL, location, and device orientation, and waits for a response. If an RSP's context capability message states that it matches a photo with a high degree of confidence (*i.e., sensorFitness* is greater than 50; Figure 30, right), our app automatically subscribes to it. If no high confidence match is found, the app provides users with the five highest matching appliances based on their reported context, and lets them choose which one they want to use. Our app always lets users return to the list of top matches for the most recently taken photo. This lets them gracefully recover in the event they (or the system) select the wrong appliance.

In support of **R3**, our mobile app also allows users to select appliances over long distances. Our system maintains a history of previously connected appliances, and lets users explicitly add appliances to a favorites list. The user can then use either list to reconnect to an appliance without having to be nearby.

4.2.2.2. Recognizing an Appliance

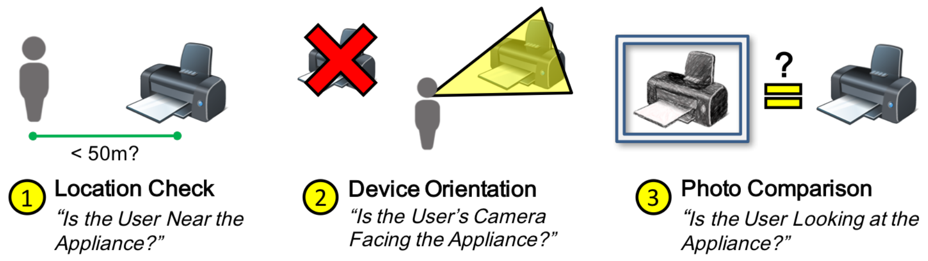

< 50m?
1 Location Check
"Is the User Near the Appliance?"
2 Device Orientation
"Is the User's Camera Facing the Appliance?"
?
3 Photo Comparison
"Is the User Looking at the Appliance?"

Figure 31. Snap-To-It's appliance recognition process.

Snap-To-It matches user requests to specific appliances in three stages (Figure 31). In the first stage, each RSP extracts the latitude/longitude from the request message and calculates the distance between itself and the user. The RSP only then proceeds to the next stage if the user is within a predefined distance (*e.g.,* 50 meters). Since indoor location tracking is imprecise, our system only checks to see if the user is in the general vicinity of the appliance (*i.e.,* the same building). This lets RSPs disregard requests from users that are clearly out of visual range, and lets our system work in networks where a subnet covers a large geographic area.

In the second stage, RSPs check to see if the user's camera is pointing towards the appliance they represent. Each RSP knows what direction a user needs to be facing in order to take a picture of it (specified a priori or obtained by using its onboard compass). RSPs can then check to see if the user's device is pointing towards it. Similar to before, this test cannot definitively tell if a user is looking at it. Instead, it only prevents appliances from comparing photos that are obviously taken from the wrong direction. This improves accuracy when appliances look similar, but face different directions.

The third stage examines the user's photograph. Here, each of the remaining RSPs under consideration downloads the user photo and extracts its salient features using the Scale Invariant Feature Transform (SIFT) algorithm [76]. These features are then compared to reference photos (*i.e.,* photos of the appliance that are either provided in advance, or taken programmatically), and the results (*i.e.,* the maximum number of matches) are sent back to the mobile app. We use SIFT because the features it identifies are resilient against changes in the orientation of the camera and distance from appliances. This lets it effectively compare two images, even when they are taken from slightly different angles and distances.

For the above process to work, each RSP needs to know 1) what it looks like, 2) where its appliance is located, and 3) what direction the user needs to face to photograph it. For appliances with a GPS, compass, and display, this information can be automatically obtained by directing the RSP to periodically poll its sensors and take screenshots of its display, respectively. Many appliances (*e.g.,* printers), however, lack this capability. To support them, developers and/or administrators can collect this information via our mobile app, and provide the photos (and their associated metadata) to the RSP. While cumbersome, we have empirically found that this approach yields good results with as few as 3 reference photos (one taken from the front, and two from opposite 45 degree angles). This makes it easy for developers to create RSPs for a wide range of hardware, software, and non-computational objects (satisfying **R2**).

Additionally, our process requires that every reference photo contain some information about an appliance's surroundings. To achieve this, our app provides users with a targeting reticule (*i.e.,* a box), and instructs them to keep the appliance in the center of the image (Figure 27, left). The resulting photos contain enough background scenery to give the RSP a sense of where it is located in relation to the environment, as well as what appliances are near it. This lets our system differentiate between similar looking appliances, even when they are physically near each other (an examination of Snap-To-It's accuracy is provided in the following sections).

4.2.2.3. Updating/Maintaining Reference Photos

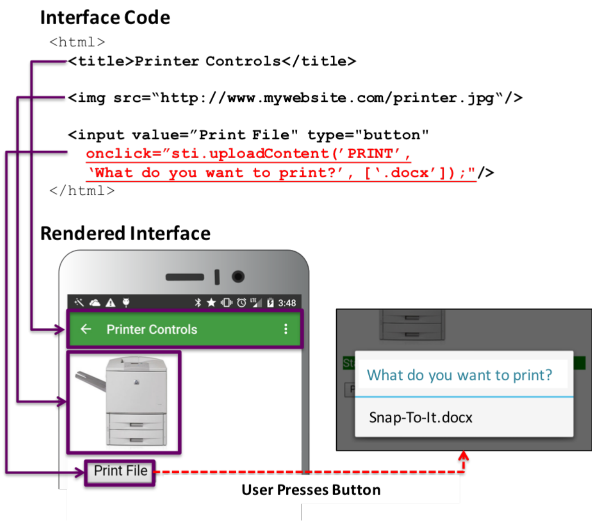
Snap-To-It's accuracy largely depends on the quality of its reference photos. When these photos closely match the appliance's actual appearance, our system is able to identify appliances with a high degree of confidence. However, reference photos become stale over time as the environment changes. Consequently, there needs to be a way for RSPs to update their reference images with minimal assistance.

Snap-To-It overcomes this problem by utilizing user photographs. When users connect to an RSP, the photo they took is compared to the RSP's reference images. If the photo was taken at a similar angle to an existing photo (*e.g.,* within 10 degrees azimuth/pitch/roll) and has a low number of SIFT matches, the RSP replaces the older reference image. Otherwise, the system adds the photo to its current library (up to a specified limit). By letting RSPs update their reference images when the SIFT accuracy drops below a threshold, our system is able to learn how an environment is changing over time. This keeps them up to date, while simultaneously allowing them to recognize appliances from a wider range of angles.

Although this provides RSPs with new photos, it also opens the possibility of RSPs receiving blurry or inaccurate images by mistake (*e.g.,* an image of another appliance). To overcome this, RSPs ask users to judge another user's photo before adding it to its library. By leveraging user responses, RSPs can weed out low quality photos from its library without having to explicitly judge image quality on its own. Moreover, since judging is only needed when a new photo is added, the impact on the end user experience is kept to a minimum.

4.2.2.4. Defining User Interfaces

From our technology probe, we know that there are a handful of common operations that users need to perform in order to effectively control an appliance, such as the ability to transmit remote commands. As a result, Snap-To-It provides a robust JavaScript API that supports these core functions. When developers specify their interface, they can insert calls to our API at key events (*e.g.,* when a button is pressed). These calls then direct the app to perform a pre-canned or customizable action (Figure 32).



```
<html>
  <title>Printer Controls</title>

  <img src="http://www.mywebsite.com/printer.jpg"/>

  <input value="Print File" type="button"
    onclick="sti.uploadContent('PRINT',
    'What do you want to print?', ['.docx']);"/>
</html>
```

Figure 32. Sample controls for a printer RSP. When users press the "Print File" button, it invokes our JavaScript API (red/underlined) to select and upload a file to the RSP (bottom right), and transmit the "PRINT" command to the RSP.

This approach lets Snap-To-It perform operations that are normally inaccessible (or difficult to perform) from a mobile web browser. Our system allows developers to customize these operations (*e.g.,* specify what command is transmitted), and "push" new UIs without requiring the user to refresh their display. This allows them to create responsive, arbitrarily complex UIs (supporting **R4**), and makes our system extensible to a wide range of appliances and use cases.

4.2.2.5. Sharing Preferences

RSPs can also store preferences (*e.g.,* favorite TV channels, exercise settings) on the user's device, similar to the way that websites store cookies on a client device. When an RSP needs to save a preference, it calls our API's SETPREFERENCE() method, and specifies the name of the preference and its value. These values are then stored in the mobile app, and are provided to the RSP in future sessions.

Moreover, preferences can be shared across appliances (supporting **R5**). Each time an RSP replies to a request, it specifies the preference(s) it needs (Figure 30, right) in its context capability message. The mobile app then delivers these preferences (if authorized by the user) to the RSP when it connects. Since RSPs can ask for any preference, our system lets appliances configure themselves (*e.g.,* transfer exercise settings) based on the user's interactions with similar and/or complementary appliances. In doing so, we reduce the need for manual configuration in many situations.

4.2.2.6. Identifying Users

There are many situations where an RSP needs to customize the services it offers on a per-user basis. For example, an RSP for a printer on a college campus might want to provide students and faculty members with an interface to print in color, while limiting visitors to black and white.

To support these situations, Snap-To-It allows appliances to distinguish between individual users. Each time a mobile app connects to the RSP, it transmits a set of identifying credentials to the RSP. The RSP can then use these values to determine what level of service to provide. For now, our credentials contain the device's Android ID and a hash of the user's email address (Figure 30, left), as this information is sufficient to identify a particular user (provided the email address or ID is known beforehand). In the future, however, a more robust authentication method (*i.e.,* digital signatures) could be used in its place. This would provide greater security, while still allowing our system to offer relevant and appropriate services (supporting **R5** as well).

4.2.2.7. Creating a New RSP

Snap-To-It makes it easy for both first and third party developers to create their own RSPs. Our middleware comes with a base REMOTESERVICEPROVIDER class that automatically handles all image processing and communications. This class extends GCF's CONTEXTPROVIDER base class, and implements its core methods in the following ways:

- In the constructor (Figure 2, line 4), we assign the RSP a context type of "RSP".
- In the SENDCAPABILITY() method (Figure 2, line 23), we extract the URL and device orientation from the context request message's payload, and analyze it using the three step process described in Figure 31. If the request satisfies all three conditions, we return TRUE; otherwise, we return FALSE.
- In the GETFITNESS() method (Figure 2, line 29), we compare the user's photograph to the RSP's current reference photo set. We return the number of SIFT matches from the best matching photograph.
- In the SENDCONTEXT() method (Figure 2, line 35), we deliver the appliance's user interface (specified by the developer, as described below) to all subscribed devices.

```
public class Printer_RSP extends RemoteServiceProvider
{
 public Printer_RSP () {
 // Step 1: Add Reference Photo(s)
 addPhoto("Printer1.jpeg)";
 }

 public String getInterface(User u) {
 // Step 2: Create and Deliver a User Interface
 return "<html>...</html>";
 }

 public void onRemoteCommand(User u, Command c) {
 // Step 3: Process Commands Sent by Mobile App
 if (c.getCommand().equals("PRINT_FILE"){
  print(c.getURL());
 }
 }
}
```

Figure 33. Sample implementation of a printer remote service provider.

To create a new RSP, developers create their own RSP class that inherits from REMOTESERVICEPROVIDER, and add their own functionality. To show how this is done, we provide a sample implementation of a printer RSP in Figure 33. This code is simplified for brevity, but highlights the three steps needed to create an RSP from scratch:

1. **Add Reference Photos.** In the first step, we tell the RSP what its appliance looks like. For this example, we only provide a single image (along with its location, azimuth, pitch, and roll). However, more photos will obviously improve its ability to recognize itself in a user-submitted image (at the cost of increased computation time).
2. **Specify the UI.** The second step is to return the user interface that the RSP will provide when SENDCONTEXT() is called. For this particular example, we return the HTML code that is described in Figure 32. Alternatively, developers can also the user with an HTTP/S link, or generate a custom UI for each subscribed user.
3. **Process Commands.** The final step is to process incoming commands from the mobile app. From Figure 32, we see that the app will automatically transmit a command (*e.g.,* "PRINT") to the RSP each time the user uploads a file. To detect this command, we check for this string in the ONCOMPUTEINSTRUCTIONRECEIVED() method. We then use the URL contained within the command to download and print the file.

4.2.3. *VALIDATION*

We validate Snap-To-It in two parts. First, we present four example applications that were created using our system. We then evaluate Snap-To-It's accuracy, both under controlled conditions and after a two-month deployment.

4.2.3.1. Example Applications

We have created four example applications (Figure 34 - Figure 37) using Snap-To-It. Each application is inspired by our general use cases, and demonstrates the range of opportunistic interactions (involving both hardware and software) our middleware supports.

Application #1: Game Controller

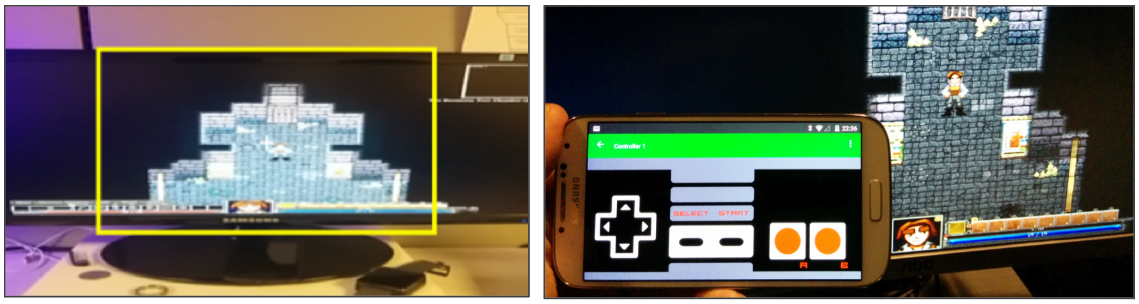

Photograph (Taken from the Snap-To-It App)
Gamepad Interface (Provided by RSP)
Controller 1
SELECT
START
A
B

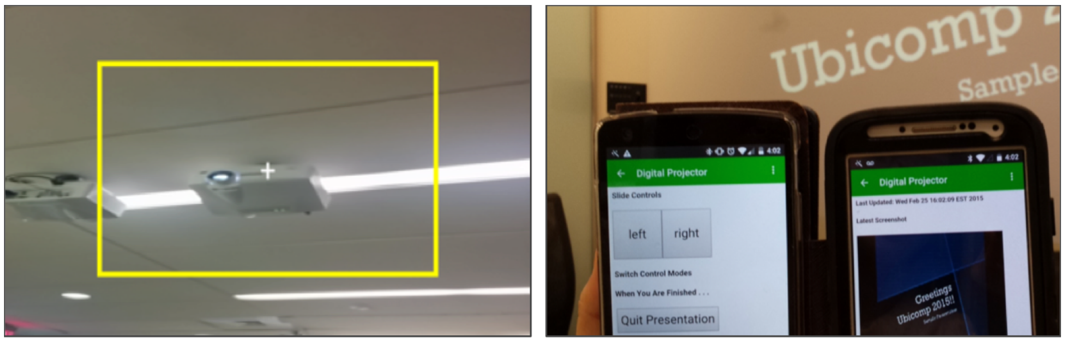
Figure 34. Video game application. By taking a photograph of a laptop screen (left), Snap-To-It returns a game controller interface (right). Users can then tap on the buttons to manipulate the on-screen character.

Our first application uses Snap-To-It to quickly control a video game (Figure 34). For this application, we deployed a laptop running both a commercial game and a *video game* RSP. When a user takes a photo of the screen, Snap-To-It sends a multicast datagram containing a link to the image, as well as the user's location and device orientation. Our RSP then verifies that the user's phone was pointed towards the laptop and looking at the screen, and sends back a response containing the number of visual feature matches and its IP address. When the user connects to the RSP, the mobile app receives an HTML interface for a gamepad and renders it on the screen. As the user presses buttons on the UI, the interface directs the mobile app to send commands to the RSP (*e.g.,* "LEFT"). These commands are then translated into keyboard presses that control the game.

This application demonstrates our system's ability to select appliances, render interfaces, and transmit commands without requiring either the mobile app or the RSP to know of each other *a priori*. Additionally, this application also shows how preferences can be shared through our system. In addition to the default controller shown in Figure 34, the application also lets users choose between multiple controller layouts. This preference is then stored on the user's device, and can recreate their control settings when playing on a different computer.

Note that we did not use a special API to control the game. Instead, our RSP merely translates the user's controller commands into keyboard presses to give the user the impression that they are controlling the game directly. The ability to wrap Snap-To-It around existing applications is inspired by prior work in software overloading [33], and we believe that this is a general technique that can be used to instrument a wide range of existing hardware and software appliances.

Application #2: Digital Projector
Our second application shows how Snap-To-It can be used to upload/download content. We created an RSP that is linked to a conference room's multimedia control system. When users take a photo of the room's digital projector, the RSP provides them with an interface that lets them upload a presentation. The RSP then downloads and projects the presentation, and provides the user with slide deck controls (Figure 35).

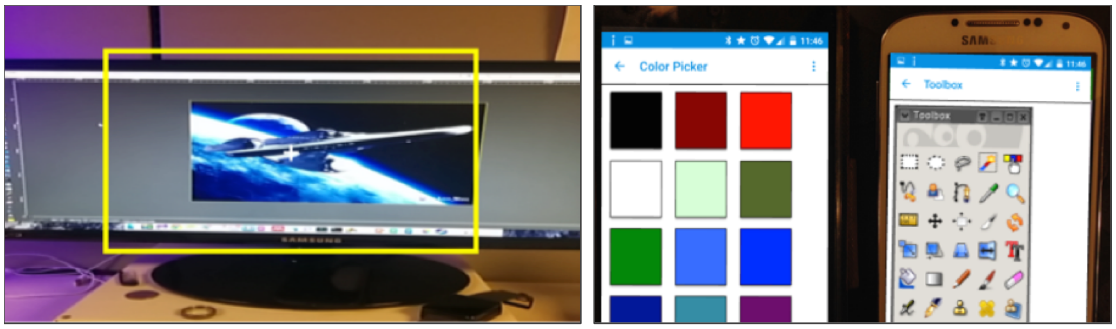


Photograph (Taken from the Snap-To-It App) Interface for Presenter (left) and Audience (right)

Figure 35. Digital projector application. By taking a photograph of a physical appliance (left), users receive a custom interface to either 1) upload and control a PowerPoint presentation, or 2) download the currently running slideshow (right).

This application also shows how Snap-To-It can support multiple users at once. Our RSP can differentiate between the user that uploaded the presentation (*i.e.,* the presenter) and users that connect afterwards (*i.e.,* audience members). The RSP can then provide audience members with a separate interface to 1) see the currently visible slide, and 2) download a copy of the presentation. Although simple in concept, this application shows how Snap-To-It can support multiple user types through a single RSP. This allows our system to support a wide range of collaborative and cooperative activities—a capability not explicitly supported in existing remote control systems.

Application #3: Paint Application



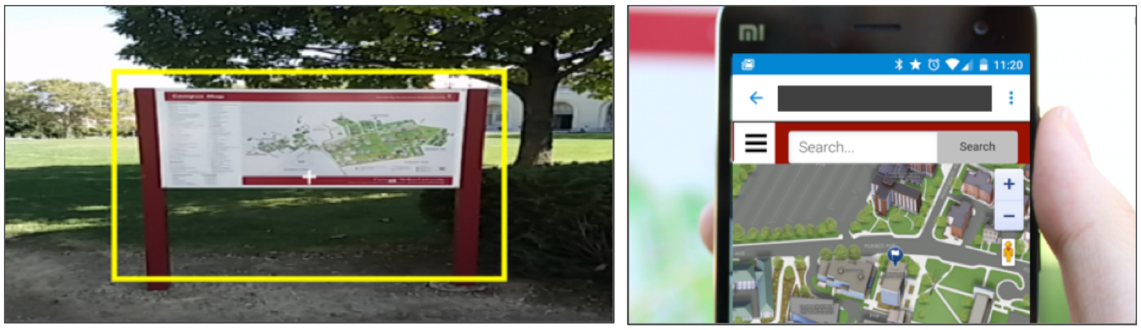
Photograph (Taken from the Snap-To-It App) Paint Controls

Figure 36. Paint application. By taking a photograph of a paint application (*e.g.,* GIMP, left), Snap-To-It lets users display toolbars on their tablets/phones. This frees up more space on the primary display for the image being edited.

Our third application uses Snap-To-It to enhance a traditional software UI. By taking a photo of a paint application, users are provided with a series of drawing tools on their mobile display. The user can spread these controls across multiple devices (Figure 36) so that they can access these controls without taking up room on the main display.

This application is heavily inspired by prior research in cross device interactions [53]. However, our system builds on this work by allowing devices to share interfaces without having to install the same software or pair in advance. This lets users utilize any nearby device as an extended control surface, while simultaneously making these types of interactions easy to develop and deploy.

Application #4: Campus Map



Photograph (Taken from the Snap-To-It App) Digital Map

Figure 37. Campus map demonstration. By taking a photograph of a physical, non-computational map (left), Snap-To-It delivers a digital version that the user can use to monitor his/her location.

Our final application uses Snap-To-It to interact with a non-computational object (*i.e.,* a campus map). When users take a photograph of the map, the RSP provides them with a digital version (Figure 37). They can then search for points of interest without having to remain near the physical object.

A key point of this example is that the sign we used was not instrumented in any way. Instead, we simply gave the RSP a photo of the sign, and linked it to our institution's online map. In doing so, we show how Snap-To-It could one day be used to help users extract deep knowledge directly from their environment.

4.2.3.2. Experimental Evaluation

To evaluate Snap-To-It, we performed three small studies. The first compares Snap-To-It to SIFT, and shows how our system's use of multiple contexts helps it identify appliances with higher accuracy. The second compares Snap-To-It to QR codes, and shows how our system supports a wider range of angles and ranges. The third was conducted after a two-month live deployment, and shows how our system can continue to accurately identify appliances after an extended period of time.

Snap-To-It vs. SIFT Photo Recognition Accuracy

For the first study, we took eleven pictures of every printer (13), copy machine (3), and fax machine (2) in our institution's building (18 appliances total). The first five photos served as reference images, and were taken from the front and sides. The following six images served as our test set, and were taken by two experimenters on different days. The test images were taken from multiple angles from separate phones, and were not filtered to remove blurry images. For our evaluation, we created 18 RSPs (1 for each appliance), and provided each with 1, 3, or 5 reference photographs. We then had the RSPs evaluate each photo using both our system and pure SIFT.

As expected, Snap-To-It's ability to recognize an appliance depends on the quality and variety of its reference photos (Table 8). When we only provided a single reference image (taken from the front), there was an 82% chance that the correct appliance appeared at the top of the list and an 89% chance it was in the top five. With three and five reference photos, however, the chance of the correct appliance appearing at the top increased to 87% and 89%, respectively. The chance of the device appearing in the top five then exceeded 95%.

There are two important takeaways from this study. First, it shows that our system does not need a large number of reference images. Table 8 shows that our system's accuracy starts to level off with three reference images. This means

Table 8. Comparison of Snap-To-It and SIFT's photo recognition accuracy. Percentages show the number of times the correct appliance (out of 18) appeared in a list of the top 1, 3 and 5 matches.

		SNAP-TO-IT			SIFT ONLY		
		1 Ref Photo	3 Ref Photos	5 Ref Photos	1 Ref Photo	3 Ref Photos	5 Ref Photos
List Position	Top 1	81.5%	87.0%	89.0%	67.6%	79.6%	85.2%
	Top 3	87.0%	92.5%	94.6%	75.9%	92.6%	90.7%
	Top 5	88.8%	95.3%	95.5%	83.3%	93.5%	92.5%

that on-site administrators and/or developers only have to provide RSPs with a handful of images to obtain good performance.

Second, these results show how our system outperforms standard image recognition in a real-world setting. Of the 18 printers, six were the same make/model and ten were either located next to each other (and hence appear in each others' photographs) or placed in similar looking office rooms. While these similarities confused SIFT (especially in situations when the photo contains only a small amount of background scenery), Snap-To-It was able to more accurately differentiate between appliances that looked the same but faced different directions. Moreover, while SIFT came closer to Snap-To-It's accuracy with more reference photos, the results also reveal that our system is more likely to have the correct appliance at the top of the list. This makes our system more user-friendly in an opportunistic setting, and lets users be more confident that the appliance at the top of the list is the correct one.

Snap-To-It vs. QR-Code-Based Device Selection

Our second study compared device selection using Snap-To-It and QR codes. We placed 3 and 8-inch tall QR codes on the front and sides of a printer used in the previous study. We then tried to scan the code with a popular mobile app [131] from a variety of angles, starting from the front of the appliance and moving around at 20 degree increments from distances of three and six feet. At each location, we tried three times to see if the app could recognize any of the codes visible on the appliance within five seconds (the amount of time it typically takes Snap-To-It to identify the appliance). We then used Snap-To-It three times from the same location to see how it would perform (using three reference photos).

The results (Figure 38) show that QR codes are sensitive to both angle and distance. When the user was directly in front of the QR code, the app had no trouble scanning it. However, when the user looked at the code from an angle (40-60 degrees), the app was often unable to scan it reliably. Even after placing the code on multiple sides, there were some angles where none of the codes could be successfully scanned. Additionally, while a 3-inch code was sufficient when the user was one meter away, an 8-inch code was required for the app to recognize it from two meters. This may be acceptable for larger appliances such as the printer we used, but it is likely too big for smaller or narrower devices (*e.g.,* projectors). Snap-To-It was able to identify the printer from all of the locations at both distances. While the system's confidence varied depending on the angle (with the best results when angle and distance were closest to the three supplied reference images), the printer was always the top match from the 18 possible appliances. Our system's ability to collect and process reference photos from additional angles lets it become even more accurate, and suggests that Snap-To-It can be more reliable than using multiple QR codes.

Long Term Feasibility

While our first two studies show that Snap-To-It is accurate, we were also interested in seeing how well our system performs over time. To evaluate this, we deployed 24 RSPs in our environment for common appliances such as printers

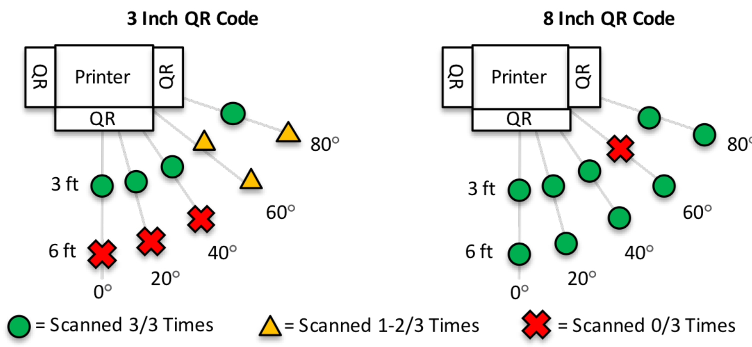


Figure 38. QR code scanning accuracy for a single appliance from multiple angles (codes placed on the front and sides) using varying sizes. Snap-To-It was able to correctly identify the appliance from each angle.

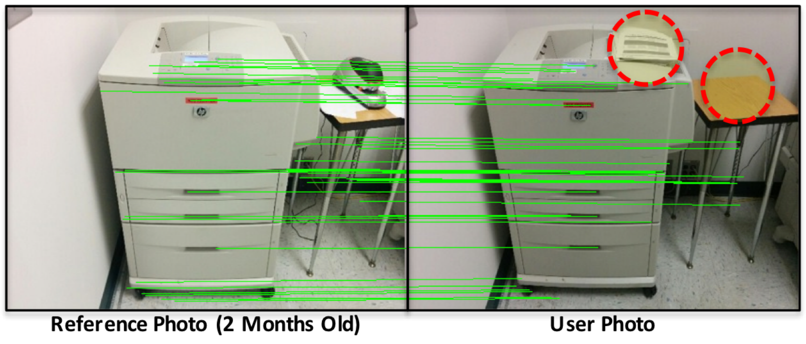


Figure 39. Sample photo analysis from our long term feasibility study. Despite noticeable differences (circled), Snap-To-It still identified the correct appliance using two-month old reference photos (SIFT matches shown as lines).

(18), computer displays (2), and digital projectors (4). We then published the Snap-To-It app in the Google Play store, and placed advertisements throughout our building letting users know that the app was available. During that time, a total of nine different users interacted with one or more appliances, and provided our RSPs with 25 new photographs (using the updating logic described earlier).

After running Snap-To-It for 59 days, we asked 16 new participants to interact with an appliance using our system. Our participants consisted of first-year Ph.D. students and visitors, as both groups were new to our environment and not affiliated with our work. For this study, we emailed them a document or a PowerPoint presentation, and asked them to use Snap-To-It to either print or project them on an appliance that they have never used before, respectively. We then observed them as they performed the task on their phones. Even though nine participants interacted with appliances with two-month old reference photographs, our system was still able to identify the correct appliance in all 16 cases (Figure 39). Additionally, in all but one case, the correct appliance was at the top of the list; in the other case, the correct appliance was second. While additional studies are needed to verify that Snap-To-It works over longer stretches of time, these results show that it works despite small but frequent changes to an environment (*e.g.,* papers placed on a printer). This suggests that our system is robust enough to be used outside of the lab.

4.2.4. Discussion

In this section, we examine Snap-To-It from the following perspectives: usability, responsiveness, and hardware requirements.

4.2.4.1. Usability

There are two factors that affect Snap-To-It's usability:

Which Appliances Are Compatible with Our System? One challenge with deploying a system like Snap-To-It is letting users know which appliances are selectable. This is a well-known problem for Ubicomp systems [9]. Although icons/markers can help, tagging every hardware/software appliance goes against the simplicity our system offers. At the same time, however, we know that users will stop using a system if it does not work consistently [25]. Consequently, it is important to provide some guidance so that users never feel like they are guessing.

We expect that users' uncertainty with Snap-To-It will decrease as more appliances become compatible. To expedite this process, however, we used multiple strategies. First, we posted a series of advertisements throughout our institution to let users know when they are entering/leaving a Snap-To-It enabled area. In addition, we are also experimenting with letting the user know when they are in Bluetooth range of a Snap-To-It compatible appliance, either by displaying a notification on the user's mobile device, or by allowing appliances to display an icon and/or beep. No approach works for all appliances. Collectively, however, they reduce the chances of users taking a photo when no services are available.

Security. Although Snap-To-It provides basic mechanisms for user authentication, it assumes that the appliances in an environment are trustworthy. This creates two security concerns. If a malicious RSP states that it strongly matches every user photograph, it can prevent users from connecting to other appliances. Additionally, since Snap-To-It uses JavaScript for its interfaces, rogue developers can potentially use it to execute malicious code on a user's device. While concerning, it is important to remember that establishing trust is an issue in every networked system. One way to overcome this is to use blacklists to block rogue RSPs. A more robust solution, however is to mandate the use of digital certificates. The infrastructure to support this already exists, and would allow our mobile app to ignore advertisements from an RSP that has not been vetted by a trusted authority.

4.2.4.2. Responsiveness

On average, Snap-To-It takes 4 seconds to find an appliance from a photograph. This includes the time needed for our app to upload an image (1360ms) and for the RSP (running on a Macbook) to download it (1043ms), calculate its features (780ms), and compare it against one or more reference images (267ms each). Once the app has connected, an additional delay is required to download the interface. This delay varies depending on internet connectivity and complexity of the UI, but ranged between 1-3 seconds in our tests.

Although 5-7 seconds may seem long, the participants in our final study found it to be fast compared to the time it takes for a user to pair with an appliance or install a driver/app. This time could be reduced even further by including the photograph in the request message, as opposed to using URLs. Furthermore, we have also improved the responsiveness of our mobile app so that it displays results as they arrive, rather than forcing users to wait for all RSPs to respond. This, combined with our system's sub-100ms latency for sending/receiving commands, makes Snap-To-It fast enough for many interactive use cases.

4.2.4.3. Hardware Requirements

Snap-To-It's architecture assumes that each appliance runs its own RSP. However, many appliances currently lack the computational power to analyze photos on their own. This is especially the case when it comes to calculating SIFT features, as the process can take upwards of 28 seconds per image on low-powered hardware (*e.g.,* a Raspberry Pi).

Fortunately, Snap-To-It can take advantage of existing infrastructure to give users the impression that they are directly interacting with an appliance. All of the RSPs described thus far were hosted on laptop/desktop computers; servers can provide similar functionality. Additionally, our approach does not require appliances to process images on their own. As an additional enhancement, we have modified the mobile app so that it calculates the SIFT features of its photos via a web service prior to sending a request. This approach increases the time needed for the app to upload a photo (from 1360ms to 3564ms), making it slightly slower for RSPs that can already calculate SIFT features quickly. However, this strategy only requires low-powered RSPs to compare SIFT features (which takes 2.4s per image on a Raspberry Pi), thereby making our goal of running RSPs on appliances technically feasible.

4.2.5. *Lessons Learned*

In contrast with Didja, which forms groups in the background, Snap-To-It is the first GCF application that lets *users* specify which device(s) they want to form an opportunistic group with. Consequently, creating this system has furthered our understanding of how GCF can be used to facilitate one-time, user-initiated interactions. In this section, we discuss two challenges that we encountered when using GCF to create Snap-To-It, and describe the specific modifications we made to the framework in order to overcome them.

Providing Explicit Support for Manual Grouping. When we originally conceived of Snap-To-It, our goal was to let GCF *automatically* group with the correct RSP(s) based on the user's photograph. During initial tests, however, we found that there are rare occasions when the system is unable identify a specific appliance from a photo, even when that image is supplemented with additional user context (*e.g.,* location, device orientation). To address this, we decided to give users a list of the top matching devices, and let them choose which one(s) they wanted to interact with; this would let Snap-To-It work in situations when no obvious match is found. While implementing this functionality, though, we quickly discovered that GCF's architecture is so optimized for forming groups on its own that it lacks mechanisms to let users (or applications) specify which devices they would like to group with. This makes it difficult to use the framework in situations where user choice is necessary.

From this experience, we realized that GCF needs to include *some* architectural support for manual grouping. To fill this need, we have created a new *manual arbiter* for GCF. This arbiter does not form groups on its own, but instead keeps track of all of the context capability messages that it has received for each context request, and provides these messages to an application when asked. The Snap-To-It mobile app uses the manual arbiter to collect context capability messages from RSPs, and uses this information to generate a list of the five most likely matches. When the user selects an RSP from the list, the mobile application adds the RSP's device ID to the arbiter's whitelist. The arbiter will then direct the framework to form a group with that device.

By including this arbiter in GCF's core library, our framework now explicitly supports applications that require manual grouping. Our solution still requires developers to provide their own user interfaces before users can easily select which device(s) they want to group with. Through this approach, however, our solution gives developers the flexibility to incorporate user choice in their applications, without mandating what that choice looks like at the UI or system level. This maximizes developer's freedom, while still providing them with a robust mechanism for discovering and grouping with devices at runtime.

Enabling Remote Commands. To date, all of the applications that we have built using GCF have only required information to flow from the context provider to the context requester. While developing Snap-To-It, however, we realized that there are times when two-way communications is required. For example, when the mobile app subscribes to a printer RSP, it is not enough to receive an interface. Instead, there needs to be a way for the mobile app to send commands to the RSP so that it can tell the printer what file to print, and when.

To address this, we have created a new *compute instruction* message in GCF. This message type allows applications to issue remote commands that are asynchronously transmitted and received by one or more subscribed context providers. To send a compute instruction, developers call the group context manager's SENDCOMPUTEINSTRUCTION() method, and pass it:

- The context type of the context provider that this message is intended for
- The name of the command
- (Optional) One or more arguments (*e.g.,* the URL to the file to print)

When a compute instruction is received, GCF routes the message to the corresponding context provider's ONCOMPUTEINSTRUCTIONRECEIVED() method. The context provider can then examine the message contents and perform the requested action.

Through the inclusion of the compute instruction message type, GCF can be used to create a wider range of interactive applications. Our framework now provides applications with a simple way to deliver arbitrary messages to each other at runtime, creating new opportunities for devices to share information and collaborate in near real time. In the following section, we introduce the Impromptu system, which extends Snap-To-It's functionality to support opportunistic interactions with applications and services in addition to appliances. In doing so, we demonstrate the usefulness of compute instructions, and show how this capability can be applied to a wide range of use cases.

4.3. Impromptu: Using GCF to Increase Users' Access to "Just in Time" Applications, Information, and Services

Up until now, our exploration of opportunistic groups has primarily focused on groupings of *tangible* entities. Didja, for example, uses GCF to identify precise groupings of users/devices when they are co-located in the same physical space. Similarly, Snap-To-It forms opportunistic groups with the appliances, devices, and physical objects that exist in a user's immediate surroundings (and can be photographed).

As we have shown, these systems already allow us to explore a wide range of context-aware applications. In this thesis, however, we are also interested in seeing how GCF can help users form opportunistic groups with *intangible* entities such as applications and/or services. To explore this idea, we have created Impromptu, a context-aware software platform that provides users with just-in-time access to relevant apps. In our system, users run a smartphone app that *continuously* monitors their context (*e.g.,* identity, activity, location) and shares it over a trusted communications

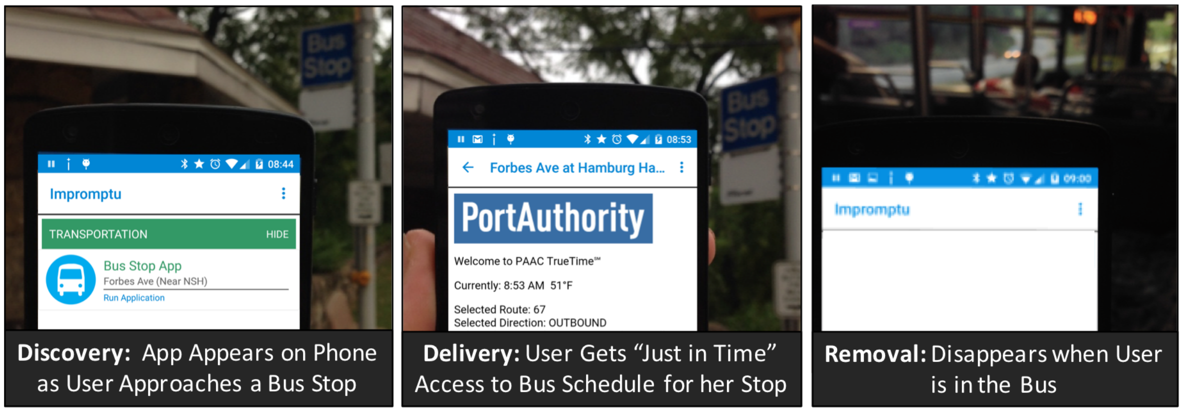

Bus Stop
08:44
Impromptu
TRANSPORTATION
HIDE
Bus Stop App
Forbes Ave (Near NSH)
Run Application
08:53
Forbes Ave at Hamburg Ha...
PortAuthority
Welcome to PAAC TrueTime™
Currently: 8:53 AM 51°F
Selected Route: 67
Selected Direction: OUTBOUND
09:00
Impromptu
Discovery: App Appears on Phone as User Approaches a Bus Stop
Delivery: User Gets "Just in Time" Access to Bus Schedule for her Stop
Removal: Disappears when User is in the Bus

Figure 40. Impromptu lets users share context with apps (not installed on their phones). Apps then appears when they are relevant (left), provide just in time access (center), and disappear when no longer needed (right).

channel. Apps hosted on our platform can then: 1) analyze this context, 2) appear on the user's device when they are contextually relevant, and 3) remove themselves when no longer needed (Figure 40).

Our work with Impromptu is motivated by the realization that there are many types of applications that are only useful within a specific context. An app that helps tourists use a city's public bus system, for example, is only needed when the user is visiting that location. Similarly, an app that helps visitors print from a public printer is only needed the (one) time they need to use the appliance. In both cases, there is a limited window of time in which the apps described above are relevant and useful. Consequently, there needs to be a way to make these apps instantly accessible so that the cost of obtaining them (*i.e.,* time, effort, bandwidth) does not outweigh the benefits.

To address this problem, Impromptu introduces the concept of "opportunistic" apps—apps that can autonomously determine if and when they might be useful to users, and make their services available. In support of this vision, Impromptu offers three contributions:

First, Impromptu presents a novel architecture for discovering and distributing opportunistic apps. Our system uses GCF to let users share any arbitrary amount of context with our platform. Individual apps (not installed on the user's phone) can then use some or all of this information to determine if they are contextually relevant, giving them precise control over when they should appear on the user's device. While prior work has tried to address this problem by *recommending* useful apps [10,23,48,123,147], these systems still require users to manually install and manage apps on their phones. In contrast, Impromptu's architecture supports each stage in an app's "life cycle." Our system not only lets users *discover* apps, but also provides a way for them to be *delivered* to the user's device, and even *removed* when no longer needed. This lets it offer a wide range of opportunistic information and services without burdening the user with an ever increasing library of once useful apps.

Second, Impromptu provides a technical solution to deploy apps that only need to be run once or sparingly. Building on our work from Snap-To-It, our platform lets developers render any HTML/JavaScript interface from within the Impromptu host application. In addition to being lightweight compared to a full-sized app, these interfaces can be dynamically generated in near real-time, thereby allowing them to be run on users' devices without having to be installed. At the same time, Impromptu can also deliver traditional Android applications. This increases the range of apps that can be delivered via Impromptu, without requiring developers to recompile them or redesign them from scratch.

Third, Impromptu increases the range of possible interactions by allowing apps to dynamically share information and services with each other. Our platform lets apps advertise what information and services they can consume and provide. Our host application can then dynamically detect compatible apps at runtime, and generate UI elements to let users invoke their combined services. Although prior work has shown how recombining information services can be advantageous on the desktop and web platforms [27,35,65,92], they have not dealt with the engineering issues of supporting this for modern mobile apps. Meanwhile, Google Now supports limited interactions with third party apps, but 1) requires the apps to already be installed, and 2) does not allow apps to communicate with each other. Impromptu builds on this work and investigates how the ability to automatically share information and services can increase users' abilities to take advantage of functional links between mobile apps. This is particularly important in an opportunistic setting, as users may be so focused on learning how to use an app for the first time that they may not form these connections on their own.

Impromptu addresses our second research question by showing how opportunistic groups can help users take advantage of the full range of apps and services available through their mobile devices. In the following sections, we describe Impromptu's architecture, and show how the system uses GCF to 1) obtain and analyze user context, 2) render interfaces, and 3) advertise their services. We examine Impromptu's ability to support opportunistic applications based on six deployments—three in the lab and three in the field. Finally, we identify limitations of our current prototype, and discuss how the need for an "always on" version of GCF influenced our framework's design.

4.3.1. *System Architecture*

Impromptu provides an end-to-end architecture that supports the discovery, delivery, and removal of contextually relevant applications. The platform consists of:

1. A series of **app providers** (created by developers), which each represents a single app. App providers are GCF context providers that can request, receive and analyze user context. They then advertise their services to the user when they consider themselves contextually relevant, and deliver interfaces to the host application when the user connects. App providers can also be used as a wrapper for existing (*i.e.,* "legacy") Android apps.
2. A **host application** (provided by Impromptu) which runs on the user's smartphone and/or tablet, and is responsible for 1) monitoring the user's context, 2) sharing it with app providers, and 3) running apps.
3. An **application directory** (provided by Impromptu), which relays messages between the host application and app providers. The directory delivers user context to all connected app providers, and sends their responses back to the host application. The directory also collects feedback from host applications to learn which apps users are actively ignoring. This lets it ignore apps that are recommending themselves at inappropriate times.
4. (Optional) A series of **app beacons** (deployed by developers), which are installed throughout an environment. Beacons contain one or more app providers, and can scan for nearby devices, analyze context, and deliver apps to users as they come within range.

Figure 41 presents an overview of Impromptu's architecture. In our system, host applications request *apps and/or services* from the application directory by transmitting context request messages that contain the user's current context in its payload (Figure 41a). At the same time, app providers request *user context* from the application directory by transmitting their own context request message (Figure 41b). When an app provider analyzes a user's context (Figure 41c) and determines that it is relevant, it transmits a *compute instruction* to the application directory that contains all of the information needed by the host application to generate an icon and to it connect at a later time (*e.g.,* IP address, port, context type; Figure 41d). This information is then sent back to the correct host application *via* a context data message (Figure 41e), where it is used to generate an icon.

In the following sections, we provide a more comprehensive look at Impromptu's inner workings. First, we describe each component in our platform. We then show how these components interact with each other to provide users with contextually relevant apps.

4.3.1.1. Host Application

The host application is an Android app that periodically assesses the user's context and openly shares it with the application directory. It then collects advertisements from relevant app providers, and uses this information to create an up-to-date list of contextually relevant apps. The host application runs continuously on the user's phone as a background service. This lets it share context and notify users when relevant apps are available (using Android's notification API), regardless of whether or not the user is looking at his/her screen. In this section, we highlight the main functions of the host application. First, we look at the specific types of context that it collects and shares. We then show how it can run apps without having to install software on the user's device, as well as provide (limited) support for legacy apps.

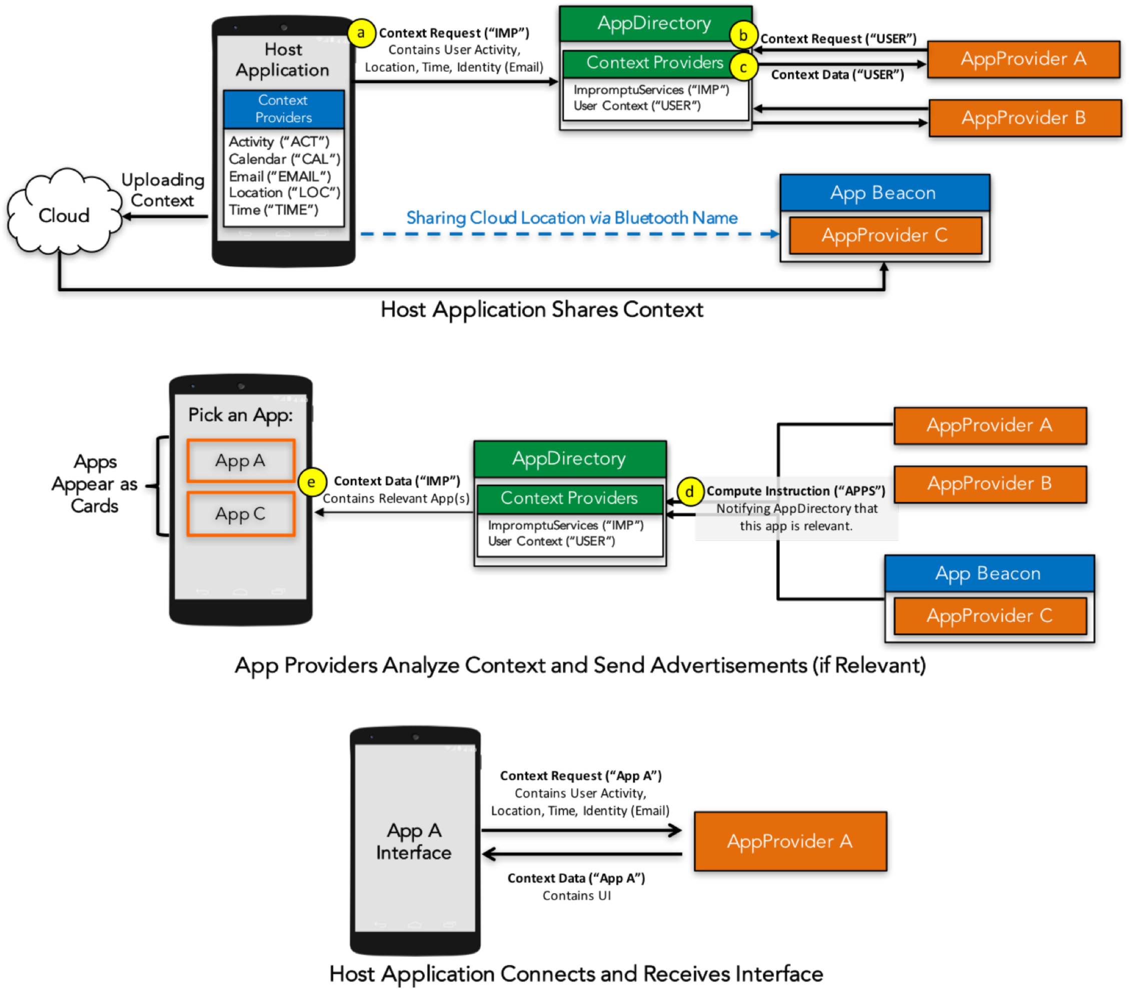


Figure 41. A high level view of Impromptu's architecture, detailing the process by which the host application shares context (top), receives advertisements from relevant apps (middle), and downloads interfaces (bottom).

Collecting User Context

The host application is responsible for monitoring the user's context. Our app considers multiple types of context along four categories: location, identity, activity, and time. These categories were identified by Dey and Abowd as being the most informative types of context [30], and provide applications with a wealth of information to determine if and/or when they are relevant.

We use a combination of prebuilt and custom context providers in order to infer each context category. To obtain location data, we use GCF's LOCATIONPROVIDER (context type = "LOC") which uses a combination of GPS sensors and network triangulation (*i.e.,* the same approach used by services like Google Now) to determine the user's geographic coordinates. Similarly, we use a LOCALTIMEPROVIDER (context type = "TIME") to query the system clock and return the user's time zone. To represent the user's identity, we developed a custom EMAILADDRESSPROVIDER (context type = "EMAIL") that provides a SHA1 hash of the domain name (*e.g.,* "gmail.com") and entire address. This lets apps that already know a user's full email address to verify their identity. Finally, to determine the user's activity, we use GCF's ACTIVITYPROVIDER (context type = "ACT"), which is built on top of Google's Activity Recognition Toolkit [132]. This context provider fuses readings from multiple sensors in order to infer the user's physical activity (*e.g.,* "ON_FOOT", "IN_VEHICLE"). For each context type, we issue a LOCAL_ONLY request using the group context manager's SENDREQUEST() method. GCF then ensures that the application receives an update each time each context type changes.

These contexts were empirically selected to strike a balance between allowing an app to determine if it is contextually relevant, and preserving users' privacy. It is important to note, however, that this list is only a starting point. Our app also comes with context providers that can provide information about the user's contact lists, calendar schedules, and hardware sensors. For now, we have configured the app to only provide this information to apps when they request it and are given permission by the user. If needed, however, we can easily reconfigure the host application to produce and share these additional types of context by default.

Sharing User Context

Impromptu uses two different techniques to share the user's context (Figure 41, top). The first way is through a traditional network connection. Each host application is connected to the application directory *via* a dedicated MQTT channel. At predefined intervals (*e.g.,* once per minute), the host application checks to see if the user's context has changed. If so, the host application transmits a context request message containing the updated context to the application directory. This context is then delivered to all subscribed app providers.

The second way of sharing context, is *via* short range radio IDs. Similar to work done by [4,24], we have devised a technique that allows Impromptu to share information by altering a device's Bluetooth name. Each time the host application transmits a new set of context to the application directory, it also uploads a file to a dedicated web server containing the same information. It then modifies the Bluetooth name of the device to contain a URL to this file, and sets the device to be discoverable. As the user moves throughout an environment, app beacons are able to detect the user's device via Bluetooth discovery. These beacons can extract the URL from the name and download the file, thereby allowing them analyze the user's context without having to pair with the device.

These two techniques cover a wide range of use cases. Our first technique works well for apps that require users to be outdoors (and thus, have access to GPS positioning data), or only need a general sense of where the user is located (*e.g.,* in a department store) in order to determine if they are contextually relevant. Our Bluetooth-based technique, on the other hand, is better suited for apps that require fine-grained indoor location (*e.g.,* an app that only lets users control a printer when they are near it). By supporting both use cases, Impromptu eliminates the need to instrument the environment in many situations, while still letting developers do so when it is necessary.

Running Apps

The host application displays contextually relevant apps as a list of cards, sorted by category (Figure 42, left). By selecting a card, users can view additional information about a specific app, such as the specific information (*i.e.,* sensors, context) it needs to run. The user can then run the app, which causes the host application to connect to the corresponding app provider and download the specified user interface (Figure 42, right).

One of our primary goals with Impromptu is to let apps be instantly accessible once they are deemed relevant. To achieve this, our interfaces are created using web-based technologies (similar to work done in [1,56,96]). When the host application connects to an app provider, it receives one or more interface messages that specifies the UI(s) (*i.e.,* HTML/JavaScript) that it needs to render. By leveraging existing web-application Figure 3. Impromptu apps appear as cards, sorted by category (left). Users can select a card to run the app (right). By leveraging modern web UI toolkits (*e.g.,* Enyo [133], Phaser [134]), Impromptu is able to display a wide range of static and dynamic interfaces. This gives users the experience of a native app without requiring them to install and update it.

Although the host application is optimized to run apps created specifically for our system, it can also run traditional Android apps. By specifying an application's package name (*e.g.,* "com.example.myapp") instead of an HTML interface,

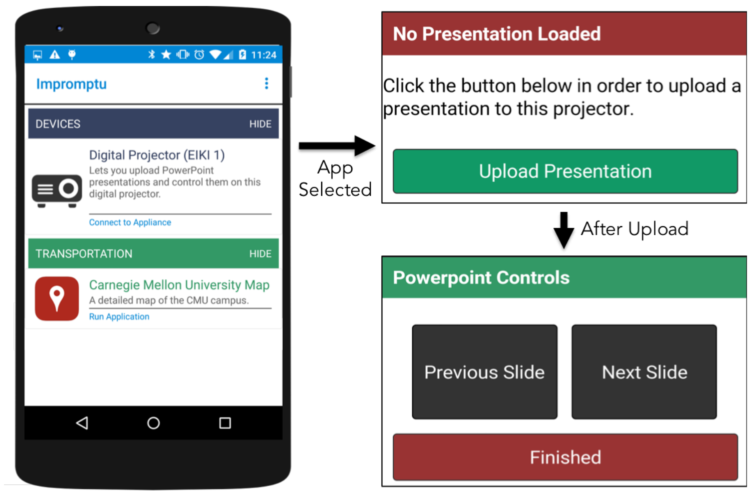

Impromptu
DEVICES
HIDE
Digital Projector (EIKI 1)
Lets you upload PowerPoint presentations and control them on this digital projector.
Connect to Appliance
TRANSPORTATION
HIDE
Carnegie Mellon University Map
A detailed map of the CMU campus.
Run Application
App Selected
No Presentation Loaded
Click the button below in order to upload a presentation to this projector.
Upload Presentation
After Upload
Powerpoint Controls
Previous Slide
Next Slide
Finished

Figure 42. Impromptu apps appear as cards, sorted by category (left). Users can select a card to run an app and receive an interface (right).

the system can look up and install the app directly from an existing app store (*e.g.,* Google Play). It can then automatically run the app and uninstall it once the app is no longer relevant. For security reasons enforced by the Android operating system, our app must explicitly prompt the user before installing and uninstalling legacy apps. Despite this, this capability increases the types of services that can be delivered through our platform, and give developers a new way to distribute their apps to their target audience.

4.3.1.2. App Providers

App providers are context providers that deliver contextually relevant services and information to the host application. Each app provider is responsible for a single Impromptu app, and can be run on either an Internet connected device (*e.g.,* a laptop or server), or on a Bluetooth and network enabled beacon (*e.g.,* a smartphone).

The Impromptu SDK provides a base AppProvider class that automatically 1) connects to the application directory to receive user context, 2) sends advertisements to specific users through the directory, and 3) delivers interfaces to the host application when it connects. Developers can then create their own Impromptu app by performing the following steps:

Step 1: Decide on a Context Source. The first step in creating an app provider is to decide how it will obtain user context. By default, app providers automatically connect to the application directory, and receive regular context updates. Alternatively, Impromptu lets developers install the provider on an app beacon. The provider will then only receive context from users in Bluetooth range.

Step 2: Analyze Incoming User Context. Once context has been obtained, it must be analyzed to see if the app is relevant to the user. How this is done depends on the app. An app that displays a bus schedule, for example, may need to look at a user's activity and location in order to determine if she is standing near a bus stop. Alternatively, an app that lets college students use an on-campus printer may only need to know that the user is in proximity of the printer and is a registered student. To accommodate these diverse use cases, each app provider has an abstract Java method (isRelevant()) that takes the user's context as input, and returns a Boolean value stating if the app is relevant. The app provider automatically calls this method whenever it receives context from a user, and uses the result to determine if it should advertise its services.

The decision to have app providers analyze user context is motivated from both a practical and research standpoint. Our long term goal is to have apps formally specify their relevance criteria (using ordered tuples or a formalized context language, as suggested in [11]) so that we can offload this computation to our application directory. We have discovered, however, that even "simple" applications have complex relevance criteria. For example, an app that is intended for conference attendees has to compare each users' identity against a white list to determine if it is relevant. Similarly, an app that notifies users of when a friend or colleague is in town needs to know 1) where every user is located, and 2) which users are friends. In both examples, the information required to determine if an app is relevant is both dynamic and domain specific. As a result, it is oftentimes more intuitive for developers to perform the check programmatically than it is to specify it using a formalized language.

For now, we have chosen to deal with this problem by focusing on the most popular use cases. Since many opportunistic apps are location specific, Impromptu now lets app providers specify a geofenced region representing the location(s) where it is relevant. Our application directory will then automatically send an advertisement on behalf of the app when the user is in these regions. As the number of Impromptu apps increase, we expect to be able to identify popular context criteria and offer further optimizations. Yet by always giving apps the option to analyze context, we ensure that our platform supports the widest conceivable range of apps.

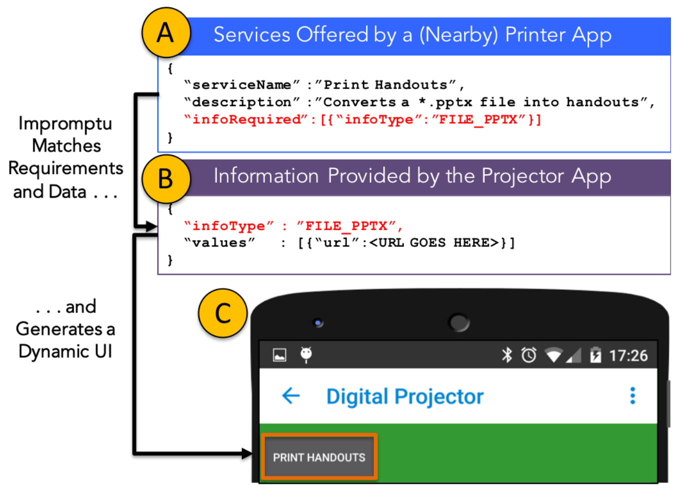
Step 3: Deliver App Content. The third step in creating an app provider is specifying what information/interface to send to the user's host application. This step occurs in two parts. When developers first instantiate an app provider, they need to provide the following:

- The name of the app (and the URL to its icon)
- A brief text description
- A "lease time" that states how long the app's services are relevant to a particular user (*e.g.,* 60 seconds). As long as an app remains relevant (*i.e.,* isRelevant() returns TRUE), the lease is automatically renewed. However, once the user's context changes and the app is no longer relevant, the lease expires, and the app is removed from the host application.
- (Optional) A list of contexts (beyond that shared by default) that the app needs from the user to operate.

Each time an app provider analyzes a user's context and decides that it is contextually relevant, it returns an advertisement message containing the above information, as well as the IP address and Port where the provider is listening for new connections. The host application uses this information to create and/or update the card for this app, and saves the connection details for when the user decides to run it. Impromptu advertisements are intentionally small (*i.e.,* less than one kilobyte per app). This lets us offer users a wide range of applications without consuming excessive bandwidth.

The second set of content is delivered when a host application directly connects to the app provider. When this occurs, the app provider transmits an interface message that specifies the UI to be rendered on the host application. An app provider can deliver a wide range of interfaces based on:

1. HTML/JavaScript
2. A package name (specifying the unique name of an application on the Google Play Store)
3. A URL to an existing website


A
Services Offered by a (Nearby) Printer App
B
Information Provided by the Projector App
C
Impromptu Matches Requirements and Data . . .
. . . and Generates a Dynamic UI
17:26
Digital Projector
PRINT HANDOUTS

```
{
  "serviceName" :"Print Handouts",
  "description" :"Converts a *.pptx file into handouts",
  "infoRequired":[{"infoType":"FILE_PPTX"}]
}
```

```
{
  "infoType" : "FILE_PPTX",
  "values"   : [{"url":<URL GOES HERE>}]
}
```

Figure 43. Impromptu lets apps share services (a) and information (b) with each other. The host application can generate UI elements to invoke these services (c) so that the user can utilize these features from the currently running app.

In order to specify which technique is being used, each app provider has a method that takes the user's context as input, and outputs a string value representing the corresponding UI (*e.g.,* "HTML=..."). Similar to the previous section, this method must be implemented for each app provider. Yet by supporting these options, developers can create a wide range of interfaces, and maximize opportunities for code reuse.

(Optional): Sharing Information and/or Services. The final step in creating an app provider is to decide what information and/or services (if any) can be shared between apps. Each time an app provider sends an advertisement, it includes a list of services that it is capable of providing to other apps on the same phone. These services are specified by the developer, and are represented as a JSON string (Figure 43a) that describes the unique name of the service, its description, and the type(s) of information it needs. Additionally, each time an app provider sends an interface message, it includes a list of information that it is willing to share. This information is also represented 5 as JSON (Figure 43b), and specifies the type of the information and its value. When the host application receives both types of messages, it checks if the information provided by one app can be consumed by another app. If so, the host application generates a custom interface to let users remotely invoke the service (Figure 43c).

By allowing apps to share information and services, we allow users to take full advantage of the apps on their device. This feature is currently optional. To encourage use, however, any files or messages that are sent or received by the host application are automatically shared with the other apps on the same phone. This ensures that some information will be available, thereby encouraging developers to use it.

4.3.1.3. Application Directory

The application directory is responsible for mediating communications between the host application and app providers. It contains two context providers:

1. An IMPROMPTUSERVICESPROVIDER (context type = "IMP"), which provides contextually relevant services to each host application.
2. A USERCONTEXTPROVIDER (context type = "USER"), which collects user context and provides it to app providers.

To receive apps, each host application periodically transmits a context request message to the IMPROMPTUSERVICEPROVIDER, and passes the user's context in its payload. Meanwhile, to receive user context, each app provider sends a context request message to the USERCONTEXTPROVIDER (Figure 41, top). When the host application shares a user's context, the directory automatically forwards it to all connected app providers. It then collects their responses and forwards them to the user's device as a single, consolidated response (Figure 41, middle).

The application directory serves two purposes. First, it provides the host application with an additional layer of security. By having all app advertisements come from the directory (whom we assume is trusted), we prevent devices from having to open up a dedicated port. This prevents rogue or malicious app providers from being able to directly connect to the user's device, while simultaneously preventing the device from becoming overwhelmed if too many app providers try to communicate with it simultaneously.

Second, the application directory provides a centralized way to detect app providers that recommend themselves at inappropriate times. In addition to letting users view and run apps, our host application also lets users discard an app if it is not relevant to them. Each time the user does so, a message (containing the app's unique ID) is sent to the application directory. By compiling these messages across multiple users, the directory can learn which apps are most often rejected, and block their advertisements as punishment for a period of time. This demerit-based system is still in the process of being evaluated with actual users. Early results, however, show that this method can incentivize developers to only recommend their apps when they are actually relevant.

Currently, all users and app providers are connected to a single directory that is hosted at our institution. As Impromptu gains more apps and users, it will need to account for this additional load. One way to address this is to have a hierarchy of application directories, with each directory being responsible for a particular geographic region. This capability is already supported by our system, and allows Impromptu to scale up to any arbitrary size.

4.3.1.4. App Beacons

Applications beacons provide an alternative way to deliver apps to users. In contrast with Physical Web beacons [135], which share the same service (*i.e.,* a web URL) with every user, app beacons selectively deliver apps to nearby users based on their context. Each beacon performs Bluetooth scans to detect nearby users. If a device running the host application (in the foreground or background) is detected, the beacon automatically parses its Bluetooth name and obtains the URL to the file containing the user's context. This file is then downloaded and sent to its app provider(s) for analysis. If the app reports that it is relevant, the beacon automatically forms a connection with the user's application directory, and delivers the advertisement.

App beacons let developers deploy opportunistic apps that are only accessible from a specific location (*e.g.,* an app to control a nearby printer or digital projector). Our middleware includes an Android app that continuously performs Bluetooth scans and forwards context to designated app provider(s) for analysis. This allows developers to turn any Bluetooth-equipped and network-connected device (*e.g.,* user smartphones, tablets placed in an environment) into an app beacon. Additionally, since beacons can be connected to multiple app providers, developers can repurpose the same beacons to host multiple apps. This lets them deploy a wide range of apps in an environment without the need for additional hardware.

4.3.2. EXAMPLE APPLICATIONS

Over the past year, we have developed and deployed a total of 56 Impromptu apps, ranging from simple location based apps, to apps that are only intended for a specific population of users (*e.g.,* all of the attendees at a music concert). In this section, we focus on six notable examples. The first three are lab prototypes, and highlight Impromptu's core capabilities. The final three are use case-driven, and demonstrate how our system has thus far been utilized in support of real-world tasks. Collectively, these examples demonstrate how the ability to opportunistically discover, deliver, and remove apps is useful and increases the range and types of interactions that users can opportunistically take part in.

4.3.2.1. Lab Demonstrations

The first set of examples were created to test Impromptu's functionality, and were deployed within our lab. While being proofs-of-concept, these examples showcase our platform's core capabilities, *i.e.,* the ability to: 1) leverage multiple types of context in order to determine if and when an app is relevant, 2) support new and existing apps, and 3) share information and services across applications.

Bus Stop App

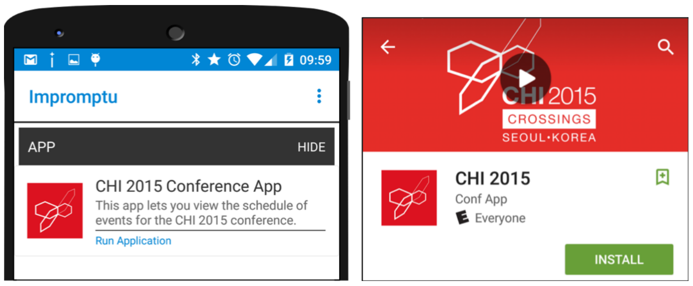
Our first example uses Impromptu to keep users apprised of the bus schedule for their specific stop (Figure 40). We created an app provider that is linked to the Pittsburgh Port Authority website. This provider is connected to the Impromptu application directory, and is relevant to users that are 1) within 10 meters of a bus stop (the locations of which are known by the app provider), and 2) standing (activity = "STILL").

When a user running Impromptu (either in the foreground or as a background process) stands near a bus stop, her phone sends her context to the application directory, which in turn shares it with our app provider. Once the app provider has analyzed this context and determined that it is contextually relevant, it sends an advertisement message (*via* the app directory) to the host application, which causes it to generate a card and generate a system notification. When the user selects either the card or the notification, the host application connects to the app provider, and starts receiving the real-time bus schedule.

This example demonstrates how using multiple contexts can significantly improve an app's ability to determine if and when it is contextually relevant. Here, location and activity are both needed to allow the app provider to distinguish between users that are actually waiting for a bus, and those that are just passing by. The use of multiple contexts also allows the app to effectively determine when it is no longer needed. Once the user boards the bus, her activity will change from "STILL" to "ON_VEHICLE." The app provider can sense this change and stop recommending itself, thereby allowing the app to disappear from the phone once the app's lease expires.

In addition, this example also shows how Impromptu maximizes code reuse. Instead of creating our own interfaces and bus tracking infrastructure from scratch, our app provider determines which bus stop the user is close to, and provides her with a URL to the corresponding Port Authority web page for that stop. Through this approach, we show how third party developers can use Impromptu as a wrapper for existing services. This is an important benefit of our approach, and significantly increases the number and types of apps that can be deployed on our platform without requiring first party "buy in."

Conference App



CHI 2015 Conference App
This app lets you view the schedule of events for the CHI 2015 conference.

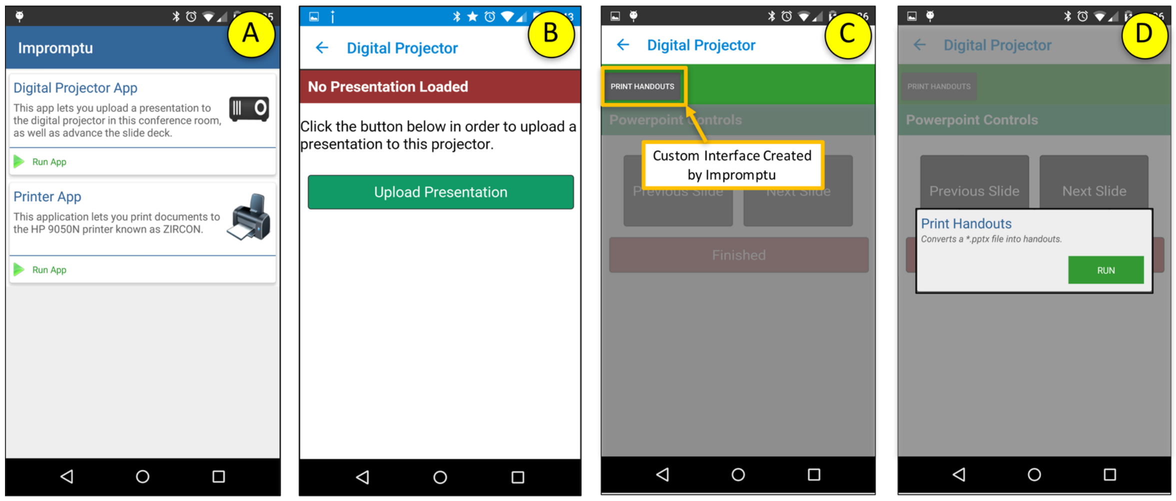
CHI 2015

Figure 44. Conference app demo. Impromptu can deliver legacy applications (left), and either run them (if installed) or redirect users to an app store to download it (right).

Our second example is an app that is relevant to attendees at an academic conference. Here, we developed an app provider that can compare the user's email address (using the hashed email addresses shared through our platform) against a list of known attendee addresses. When a match is found, the app provider checks the date to see if the conference is in progress, and provides the user with an app that lets him add sessions to his schedule, take notes, and vote for his favorite talk.

In addition to demonstrating how multiple contexts (in this case, identity and time) can be used to determine if and when an app is contextually relevant, this example also shows how Impromptu can work with legacy apps. For this demonstration, our app provider delivers the Android app used for the CHI 2015 conference. When users select the app, the host application checks if the app is already installed, and if not, redirects them to the Google Play store (Figure 44). Once the conference has ended, Impromptu automatically prompts the user to uninstall the app. By allowing legacy apps to expire, Impromptu is able to help users remove apps that are no longer relevant. This prevents users' phones from becoming filled with apps that were useful at one point, but are no longer needed.

Combining Services at Runtime



Digital Projector App
This app lets you upload a presentation to the digital projector in this conference room, as well as advance the slide deck.

Printer App
This application lets you print documents to the HP 9050N printer known as ZIRCON.

No Presentation Loaded
Click the button below in order to upload a presentation to this projector.

Print Handouts
Converts a *.pptx file into handouts.

Figure 45. Impromptu lets applications share services and information across application. When the user selects a digital projector app (a) and uploads a presentation (b), Impromptu is able to determine that the same data can be used by the printer application to print handouts. Impromptu then generates a custom UI element within the projector app (c) so that the user can invoke the service without having to switch between apps (d).

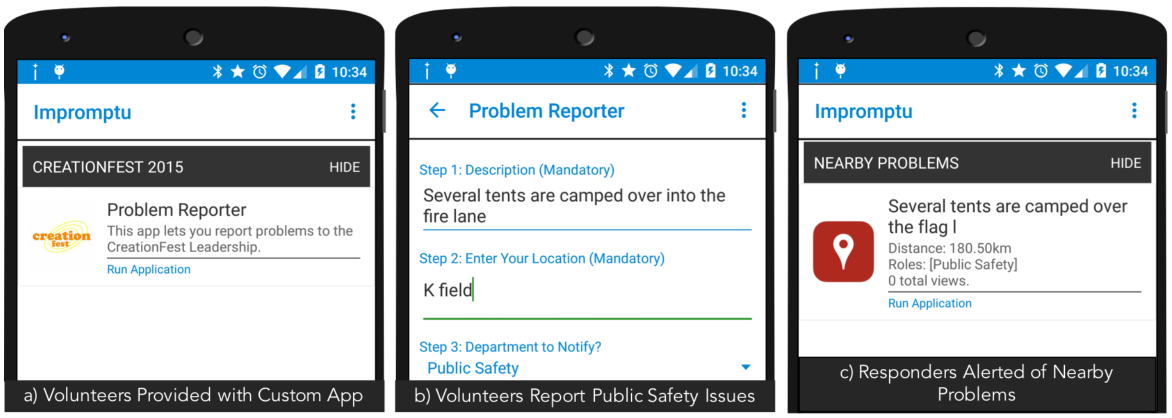
Our third example shows how Impromptu supports sharing services and information across applications. For this example, we created two app providers (hosted on an app beacon) that are both relevant when the user enters a classroom (Figure 45a). The first controls a digital projector, and lets users upload a PowerPoint presentation and advance the slides (Figure 45b-c). The second app provider is controls a printer (located near the projector), and provides 1) an interface to allow users to upload and print a file, and 2) a service that will print handouts if given a PowerPoint file. When the user runs the projector app and uploads a presentation, the host application automatically determines that same data can be used by the printer app's service. The host application then generates and inserts a button to allow the user to invoke the printer service from within the projector app (Figure 45c). This lets the user print handouts without having to switch between apps (Figure 45d).

This application provides a simple demonstration of how Impromptu increases users' access to relevant services. In this example, the user may be so focused on setting up the presentation that she may not consider using other apps to assist in this task. By letting apps share data and services, however, Impromptu can form functional linkages on the user's behalf, and allow her to make use of them with minimal added effort.

4.3.2.2. Field Trials

In addition to the prototypes described above, we have also provided Impromptu to over 60 users in support of real-world tasks. In this section, we describe three sets of applications, and show how our system's ability to provide just-in-time apps is useful in real-world situations.

Reporting Public Safety Hazards to First Responders


Impromptu
CREATIONFEST 2015 HIDE
Problem Reporter
This app lets you report problems to the CreationFest Leadership.
Run Application

Problem Reporter
Step 1: Description (Mandatory)
Several tents are camped over into the fire lane
Step 2: Enter Your Location (Mandatory)
K field
Step 3: Department to Notify?
Public Safety

Impromptu
NEARBY PROBLEMS HIDE
Several tents are camped over the flag l
Distance: 180.50km
Roles: [Public Safety]
0 total views.
Run Application

a) Volunteers Provided with Custom App
b) Volunteers Report Public Safety Issues
c) Responders Alerted of Nearby Problems

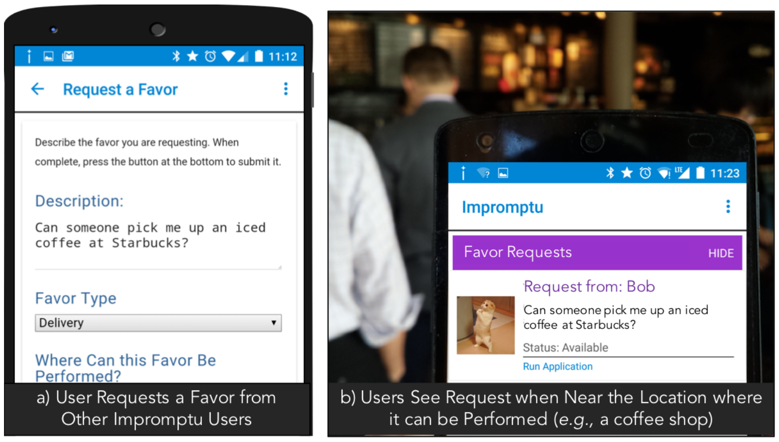
Figure 46. App provided to volunteers at the 2015 CreationFest Music Festival. Users report public safety issues (a-b), which alerts nearby first responders (c).

Our first wide scale deployment of Impromptu was conducted at the 2015 CreationFest Music Festival. For this event, we created an Impromptu app that appeared on festival workers (*i.e.,* civilian volunteers) phones, and allowed them to take photographs of potential public safety hazards (*e.g.,* leaking pipes, blocked emergency lanes; Figure 46a-b). These photos (along with the user's instantaneous location and contact information) were then forwarded to the festival's emergency management agency (EMA). A second app provider (Figure 46c) monitored first responders (*i.e.,* police, medical personnel, whose email addresses were provided to us by the EMA staff) locations, and alerted them when they were near a reported problem. The responder could then investigate the problem and mark the issue as "resolved."

This example shows how Impromptu can support one-time events. In follow up interviews, we learned that the EMA had considered making a public safety app, but did not believe that volunteers or first responders (many of which belong to functionally separate agencies) would download an app that was only useful for four days. Through Impromptu, however, we provided a simple way of distributing this app on the days that it is needed, without requiring users to find and install it on their own. Over the course of the festival, a total of 28 volunteers and first responders downloaded Impromptu. Together, these users reported 18 public safety issues, far exceeding those reported by telephone or social networking combined. This led to increased interest in our system by both the music festival staff and law enforcement agencies: "This would really be helpful in a disaster or emergency" (P2).

This example also demonstrates how Impromptu can be used to quickly deploy new apps. On the second day of the festival, we received a request from the head of the EMA to allow supervisors to see all reported problems (active and resolved) on their smartphones. In the past, such a request would have required us to 1) create and deploy a new app on the app store (and ask users to download it), or 2) update an existing app (and ask users to check for the update after a few hours). Using Impromptu, however, we were able to quickly create a new app provider that displayed every reported problem as a list, and configured it so that only a select set of users could see it on their phone. We then ran the app provider on our platform and had it instantaneously appear on the correct users' phones. In all, it took us less than an hour from the original request to create and deploy a new app on Impromptu. This highlights our system's flexibility, and shows how our vision of opportunistic apps can be useful in highly fluid situations.

Asking for Personal Favors



Describe the favor you are requesting. When complete, press the button at the bottom to submit it.

Description:

Can someone pick me up an iced coffee at Starbucks?

Where Can this Favor Be Performed?

Request from: Bob

Can someone pick me up an iced coffee at Starbucks?

Status: Available

a) User Requests a Favor from Other Impromptu Users

b) Users See Request when Near the Location where it can be Performed (*e.g.,* a coffee shop)

Figure 47. Impromptu Favors application. Users use our app to request a favor (a). Other users then see the request from Impromptu when they are near the location where it can be performed (b).

Our second real-world example was created to help users collaborate and ask for favors from each other (Figure 47). We have developed an app that allows users to post requests for simple favors (*e.g.,* "Pick up a gallon of milk from the supermarket and bring it to my house"). When a user approaches a location where a favor can be performed, Impromptu presents her with an app that displays the request with instructions on how to complete it. She can then decide if she would like to perform the favor on the requestor's behalf.

The idea of allowing users to exchange services is not new, and has been leveraged by a number of commercial services [136,137,149]. However, existing services require users to explicitly opt into a community before they can see and respond to requests. In contrast, our example uses Impromptu to support opportunistic collaborations. A user might not be willing to exert a great deal of effort to help another individual. However, through Impromptu, these users are able to see requests for favors as they go about their normal routine. This lets users reach out to a wider audience base than they could otherwise, and increases the chance of a request being fulfilled.

Table 9. Survey responses obtained from 11 participants using a 5-point Likert scale (1="Strongly Disagree", 5="Strongly Agree")

Question	Mean	Median
"Overall, I would find a system like Impromptu to be useful."	4.33	4
"Having [Impromptu] provide me with information and services was useful."	3.78	4
"The system did not overwhelm me with unnecessary information."	3.88	4

Over a 30-day trial, a total of 19 favors (across 26 users) were requested through our system. These requests ranged from simple requests (*e.g.,* "Can someone bring a pen to my office?") to specialized tasks (*e.g.,* "Can someone give me a haircut"). During this time, we observed occasions where the individual performing a favor only did so because she was at the right place at the right time. This shows how the ability to deliver opportunistically information can increase users' ability to serendipitously work together.

Using Impromptu in Everyday Lives
Our third, and most extensive use of Impromptu to date is as an everyday assistive tool. Over the past three months, we have released Impromptu to over 25 users in the Pittsburgh area, and have created a total of 28 apps to provide them with useful information and services as they go about their normal routine. Some of the apps that we have deployed include:

- **Shopping apps (12):** Provides users with the weekly advertisement for the (popular) stores that they visit.
- **Tourist apps (5):** Displays information about a location (e.g., maps, schedules) when users visit it for the first time.
- **Remote control apps (5):** Allows (authorized) users to control nearby appliances (e.g., printers, digital projectors, lighting systems) both at their workplace and in select homes.
- **Navigation apps (3):** Provides users with the schedule for a particular bus or shuttle stop, or maps of college campuses.
- **Restaurant apps (3):** Provide users with the correct menu for the restaurant they are eating at (*e.g.,* breakfast, dinner).

After a ten-week deployment, we deployed a survey application across our entire platform, and received responses from 11 users (42%). Our survey results (Table 9) show that users generally liked the possibility of having apps opportunistically delivered to them. User opinion varied, however, about the specific apps currently being offered through our system. While some users were pleasantly surprised when Impromptu provided them with an app (e.g., an app for the store they were currently visiting), they were also displeased when the system did not provide an app when they felt it should. This caused several of them to specifically ask us for "more apps."

Responses such as these are to be expected. Given our limited time and resources, we could not single-handedly produce apps for every situation. The fact that users want more apps delivered this way, however, demonstrates the potential of our system, and the need for further study

4.3.3. Discussion
Our work is inspired by the vision of a future where users have opportunistic access to mobile apps. Yet, while Impromptu shows how such a future might be possible, it also reveals the technical and social challenges in bringing it to fruition. In this section, we highlight these challenges and their implications.

Table 10. Results of battery consumption experiment. Devices were allowed to run uninterrupted for 8 hours.

Condition	Avg Power Drain (%)
Idle	10
Impromptu (Running in Background)	12

Table 11. Impromptu idle bandwidth usage (based on telemetry from 48 users), showing the amount of data consumed by Impromptu per minute, and per day when not running any apps.

	Bytes (Per Min)	MB (Per Day)
Min	539	0.757
Max	1400	2.016
Avg	819 ± 172	1.152

4.3.3.1. Battery and Bandwidth Consumption

Impromptu requires devices to continually monitor and share their context so that apps can determine if and when they are contextually relevant. Our host application requires devices to monitor the user's location and activity via sensors, keep their Bluetooth ID permanently discoverable, and maintain a continuous socket connection with the application directory in order to send context and receive apps. Thus, while we try to minimize the amount of data that is sent and received, we acknowledge that Impromptu's impact on battery and bandwidth consumption can be problematic if not kept in check.

To better understand these costs, we ran two experiments. For the first experiment, we performed a series of battery drain tests using a Nexus 5 smartphone. As a baseline, we first ran the phone for 8 hours without Impromptu. We then installed the app on the phone, and let it collect and share context for the same period of time. Each condition was tested 5 times.

For our second experiment, we monitored Impromptu's average bandwidth use across 48 different users. Here, we modified our application directory to track the amount of data that is transmitted by each device per minute. We then used this information to extrapolate the amount of data used per day. This value is not a true measure of Impromptu's total bandwidth usage, as it does not take into account the data consumed when users run apps. Nevertheless, it does show how much data our system uses when left on its own, which is an important consideration for many users when they install a new app.

The results from both experiments are presented in Table 10 and Table 11, respectively. Our results show that Impromptu's current resource requirements are minimal. Impromptu only consumes 20% more power than idle, meaning that it can run all day without significantly depleting the battery. Similarly, our system's low bandwidth usage (less than 1.2MB per day) means that our system can run on both unlimited and metered data plans, regardless of Wi-Fi availability. Together, these results suggest that Impromptu is practical to deploy as is.

4.3.3.2. User Privacy

A second concern that users of Impromptu might have is privacy. When designing Impromptu, we paid particular attention to minimize any potential privacy issues. For example, instead of directly sharing the user's email address, we chose to obfuscate it. Similarly, while we do share many contexts in plaintext, such as activity and location, this information is usually not enough to identify someone from a crowd. Through this approach, we strived to balance precision and detail of context so that it is not leaking personally identifiable information yet is still accurate and useful enough for apps to determine which information and services might be relevant to users. While we have tried to preserve user privacy, we also know that users still need to have explicit control over what contextual information they would like to share. To support this, Impromptu allows users to indicate which contexts they want to publicly share with apps (*e.g.,* identity, location). In addition, if a particular app needs more contextual information than what the user agreed to share, it prompts the user to obtain access. It is difficult to generally say how much contextual information users need to give before they start to benefit from the system. Clearly, there is a tradeoff, but this is one for users to make, as they risk getting worse or fewer suggestions of contextually relevant apps if they refuse to share.

4.3.3.3. Potential Misuse by Developers

The idea of letting apps decide when they are relevant has clear potential for misuse. A simple example would be an advertising app that always states that it is relevant without actually considering the provided context. While misuse could be intentional, it might also happen accidentally. For example, if a developer makes a bus app available to any user within 20 meters of a bus stop, it will accidentally appear to users that happen to work or live within that distance.

Allowing users to reject such apps and using a demerit-based system to block them in the future is a first measure for protecting users from potential misuse by developers. The data that Impromptu gathers about rejected apps and the contexts in which they are rejected by users is useful both for the system and also developers. Sharing this data with developers would enable them to address accidental misuse. Additionally, we are looking at developing Impromptu-specific acceptance tests that check how often an app recommends itself in response to user contexts. This would help identify intentional misuse and help the app directory decide which app providers to blacklist.

4.3.4. Lessons Learned

Impromptu is the first GCF application that is designed to run perpetually on users' smartphones. As a result, building this system has increased our understanding of how GCF can be used in an "always on" capacity. In this section, we document the major challenges that we encountered while trying to use GCF to create Impromptu, and show how we have adapted our framework in order to let it be utilized in a more persistent manner.

Allowing GCF to Run as a Background Service. As mentioned at the beginning of this thesis, one of the challenges in forming opportunistic groups is that it is difficult to predict when and where they will occur. With Impromptu, we gained firsthand experience with this problem. Although our architecture lets users share their context and receive contextually relevant applications and services, early versions of GCF only worked when the user kept the host application running on their mobile device(s). In early trials, we found that users had to know *a priori* when they needed to turn on Impromptu in order for apps to detect the user and make their services available. This defeats the point of having a system like Impromptu, and prevents it from being helpful in truly opportunistic situations.

To overcome this limitation, we realized that we needed a way to let GCF continuously send requests and receive context in the background, regardless if the user has started the app or is actively using it. To address this, we created a GCFService module that contains a group context manager, and is configured to automatically restart itself whenever 1) the phone is powered on, or 2) when the service is turned off by the user or operating system. To use the service, Impromptu calls Android's startService() method, and passes it the name of our class, as shown below:

```java
// Creates Service for GCF
Intent i = new Intent(applicationContext, GCFService.class);
this.startService(i);
```

Once the service is running, it broadcasts an intent (ACTION_GCF_STARTED). The application can then use the service's group context manager to begin requesting and receiving context.

By allowing GCF to be run as a background service, our framework provides a simple way to create mobile applications that are always on and ready to request and receive context. Yet while this solution increases the opportunities to devices to find and form opportunistic groups, it is not meant to be an all encompassing solution. Our work with Didja and Snap-To-It, for example, shows that there are many opportunistic groups that only need to be formed when the user is running an app, and thus only need GCF to run for a short period of time. Meanwhile, our analysis of Impromptu also shows that running a persistent service can be potentially costly from both an energy consumption and bandwidth standpoint. It is up to application developers to consider if their application needs this capability when designing it. But by offering this capability as an *option*, we increase the types of use cases that we can practically support.

Using Bluetooth to Improve GCF's Situational Awareness (Again). Another challenge that we encountered when developing Impromptu was finding a way to provide users with contextually relevant applications when they are in a specific location (*e.g.,* inside of a classroom). As mentioned in our system description, our solution to this problem was to once again rely on Bluetooth discovery. However, instead of simply modifying the name to contain the GCF ID (as was done in Didja), Impromptu programmatically manipulates the name to contain a URL to a *cloud file* that contains the user's context. This lets users share any arbitrary amount of information with nearby devices without requiring them to pair, and allows Impromptu's beacons to only offer their services when the user is within a predefined distance.

As we have seen twice now, the ability to use Bluetooth as an out-of-bands communications channel provides GCF with the (sorely needed) capability to identify and capitalize on nearby groupings of devices. In the following section, we introduce the Bluewave system, and show how we formally incorporate this functionality within GCF. Bluewave takes the lessons learned from Didja and Impromptu, and provides a standardized way to broadcast and listen for a broad range context using Bluetooth radio IDs. This provides a developer friendly way to share context over a short geographic radius, and allows GCF to work in environments where common connectivity is not guaranteed.

4.4. Bluewave: Using Bluetooth Names to Form Opportunistic Groups in the Real World

In **CHAPTER 1**, we motivated the need for opportunistic groups of devices (and by extension GCF) through two motivational scenarios. In the first scenario, we described a situation in which a smartphone was able to form a group with a bus' GPS sensor when its user (Jack) stepped on board. This allowed the phone to track the user's location without having to run its own sensor, thereby extending its battery life. In the second scenario, we described a situation in which three smartphones were able to share calendar information with each other while their users were engaged in an unplanned office conversation. This let the users temporarily compare schedules without having to manually grant each other access or pair.

The work described in this thesis takes significant strides towards supporting these diverse use cases. In our discussion of Didja, for example, we introduced the CalendarSync application (Figure 20), and showed how it can already address the scheduling problem described in our second scenario. Supporting scenario one, however, is more problematic. Since most buses do not have a local area network, it is highly unlikely that Jack's phone will be on the same broadcast domain as the bus' sensor. This prevents GCF from forming an opportunistic group, as there is no direct way for the phone's context request message to reach the bus.

In order to support use cases like the one described above, we need to provide GCF with an alternative way to detect nearby devices and share context—one that works regardless of where the devices are, or if they have access to common network connectivity. To satisfy this requirement, we developed Bluewave [46], a novel context sharing technique that uses Bluetooth radio IDs to convey information. With Bluewave, devices upload context to a trusted cloud server, and update their Bluetooth name to contain both a URL and set of temporary credentials. Nearby devices can then collect this information *via* Bluetooth discovery, and use it to request and receive information (Figure 48).

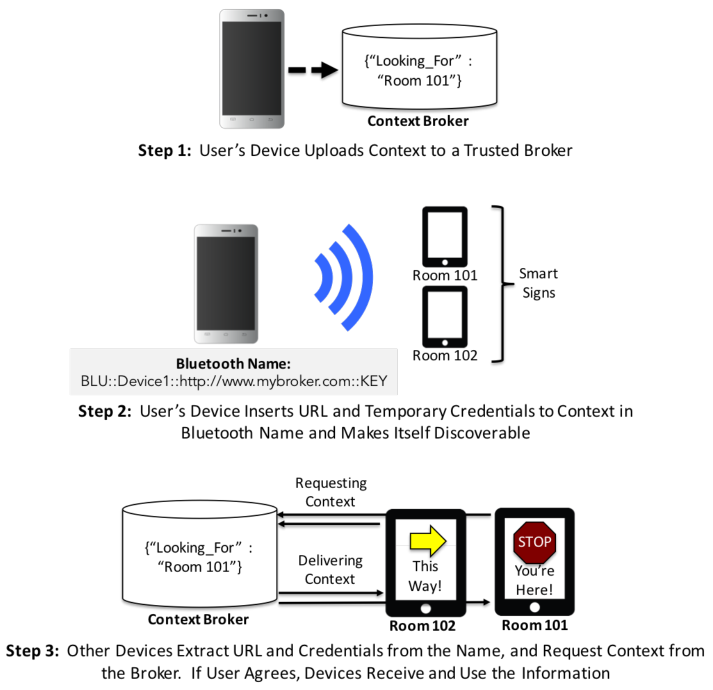

{"Looking_For" : "Room 101"}
Context Broker
Step 1: User's Device Uploads Context to a Trusted Broker
Room 101
Room 102
Smart Signs
Bluetooth Name:
BLU::Device1::http://www.mybroker.com::KEY
Step 2: User's Device Inserts URL and Temporary Credentials to Context in Bluetooth Name and Makes Itself Discoverable
Requesting Context
Delivering Context
{"Looking_For" : "Room 101"}
Context Broker
This Way!
STOP
You're Here!
Room 102
Room 101
Step 3: Other Devices Extract URL and Credentials from the Name, and Request Context from the Broker. If User Agrees, Devices Receive and Use the Information

Figure 48. In Bluewave, devices upload their context to a trusted web server (top) and modify their Bluetooth name to contain a URL and temporary credentials (middle). Other devices can scan for this information and download/use the context (bottom).

Our work with Bluewave is directly inspired by our experiences creating Didja and Impromptu. In both systems, we discovered that programmatically inserting information inside a device's Bluetooth name gave devices a simple but effective way to 1) detect when they are in close proximity of one another, and 2) share context over a small geographic area, respectively. Bluewave builds on this work by offering a *generalizable* way to share context over short distances. This gives GCF an alternatively way of detecting and forming groups that supports quick, one-way exchanges of information, while still providing applications with the option to form longer lasting groups (using GCF's subscribe/unsubscribe mechanism) when real-time, or more dynamic information is needed.

Through Bluewave, we offer the following contributions:

1. First, our work provides a truly opportunistic way to share context over short distances. By using a device's Bluetooth name as an out-of-bounds communications medium, Bluewave is able to reliably advertise and share context in both indoor and outdoor environments, regardless of whether or not they are connected to the same network. Furthermore, since all of the information needed to retrieve context is included in the device's Bluetooth name, *no actual pairing is required*. This lets Bluewave work even when the devices sharing context have never met, making it useful in situations where devices need to form groups and share information once or spontaneously.
2. Secondly, Bluewave increases our understanding of how we can responsibly share user context in an opportunistic manner. As part of our design process, we conducted an exploratory study with 15 participants, and learned that users were willing to share a wide range of information with nearby devices (including personally identifiable information) so long as they have explicit control over what context is shared and with whom. Informed by these findings, we have integrated an "opt-in" permissions model into Bluewave that lets users share context openly or on a per-application basis. By giving users the ability to specify when and where context is shared, we increase the types of information that they are willing to share. This

differentiates our system from traditional beacon technologies [135,138] (which share the same information with every device), and increases the types of context-aware applications that our system can enable.

3. Third, Bluewave helps us better understand how to make opportunistic context sharing accessible from a developer standpoint. Our system lets developers share any JSON-encoded context through their applications. Additionally, we also provide tools to let developers see 1) what context is being shared, and 2) how it is formatted. This lets developers take advantage of commonly shared contexts in their applications, and lets our system work with both established [50,93,115] and new and evolving ontologies [139].

Bluewave addresses our second research question by expanding the range of environments and situations under which GCF can find and form opportunistic groups. In doing so, we increase the types of context-aware applications that we can practically create, deploy, and explore. In the following sections, we present Bluewave's architecture, and show how our system uploads, shares and retrieves context. Afterwards, we describe our user probe, and show how responses from participants caused us to integrate privacy controls within our system. Through nine prototype applications split across two domains (public displays and the Internet of Things), we show how Bluewave can be used by developers to realize both established and novel context-aware applications. Finally, we provide experimental data to show that our system is both power-efficient and responsive, and discuss how our experiences creating Bluewave have influenced GCF's final design.

4.4.1. System Architecture

Bluewave was designed as an add-on module for GCF that lets devices broadcast and receive context without having to directly communicate with each other (Figure 49). The system consists of:

- A **client service** (Figure 49, left), which runs on individual devices, and is responsible for uploading context to the cloud, discovering nearby devices, and requesting and receiving context.
- A set of one or more **context brokers** (Figure 49, right) which are hosted on dedicated web servers, and are responsible for storing and sharing context with authorized (*i.e.,* nearby) devices.

When applications need to use Bluewave to *share* context, they access the client service, and tell it what information they want to share (*e.g.,* language settings, user's first name). The service will then automatically upload this information to its designated context broker, and modify the device's Bluetooth name. When applications need to use Bluewave to *receive* context, they tell the service what information they are interested in. The client service will then scan for nearby devices, parse their Bluetooth names, and request these elements from their respective brokers.

Although conceptually simple, a number of technical challenges had to be overcome to make Bluewave efficient and easy to use. In this section, we describe each component of our architecture. Afterwards, we present findings from an exploratory user study, and show how users' concerns about openly sharing context guided us in incorporating

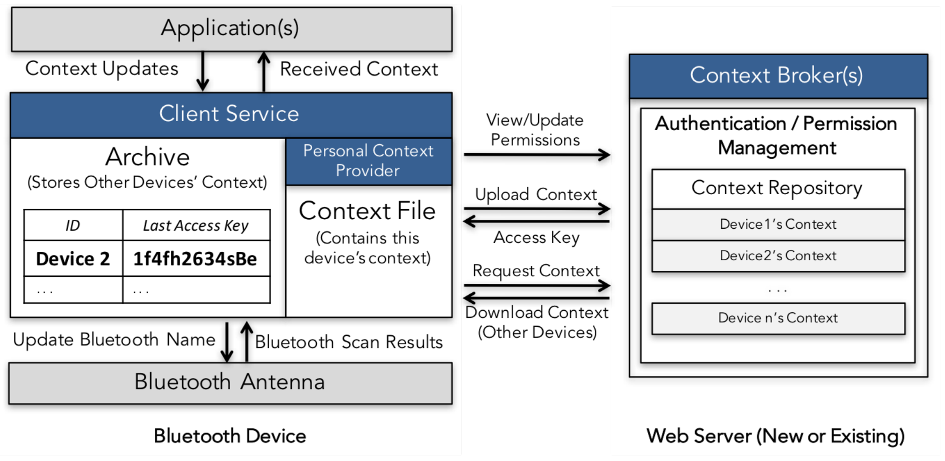


Figure 49. Bluewave's high level architecture.

privacy controls into our system. Finally, we describe Bluewave's developer tools, and show how developers can use our system to opportunistically request and receive context.

4.4.1.1. Client Service

The client service is a background process that runs on the user's phone. It is responsible for managing the user's context, advertising context to nearby devices *via* modifying the device's Bluetooth name, and obtaining context from nearby devices (as directed to by the application).

We now discuss each of these tasks in detail:

Task 1: Upload Context. The client service's primary responsibility is to upload context to the context broker. Each client service has a *personal context provider* (context type = "PCP") that contains all of the information that the application is willing to share to share. When an application wants to start/stop sharing information, they provide this information to the client service *via* the SETCONTEXT() and REMOVECONTEXT() methods, respectively. The context provider then encodes this context as JSON, and publishes the latest version to the cloud. For added efficiency, the client service only uploads the JSON elements that have actually changed to the broker. This reduces upload times, and makes sharing large amounts of context practical.

Task 2: Advertise Context. The client service's second responsibility is to advertise a device's context to its immediate neighbors. Each time the service uploads a new set of context, it modifies the device's Bluetooth name to include a URL and set of temporary credentials. Other devices can then obtain this information *via* Bluetooth discovery, and download the updated context using standard HTTP (or HTTPS) requests. A sample Bluewave name is provided below. It contains four fields, separated by colon delimiters ("::"):

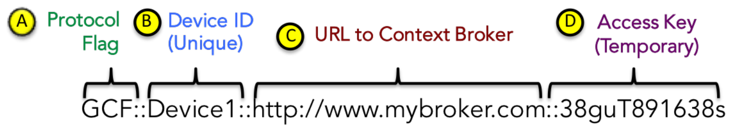


Figure 50. A sample Bluewave device name, containing a fixed length flag (A), the device's unique identifier (B), a URL to the device's context broker (C), and a temporary access key (D).

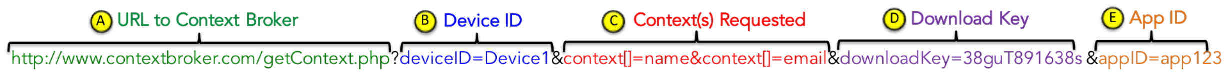

A URL to Context Broker
B Device ID
C Context(s) Requested
D Download Key
E App ID
http://www.contextbroker.com/getContext.php?deviceID=Device1&context[]=name&context[]=email&downloadKey=38guT891638s &appID=app123

Figure 51. A sample HTTP request for context, containing the URL of the device's context broker (A), its unique ID (B), the context(s) requested (C), the access key (extracted from the Bluetooth name; D), and the unique identified of the app requesting this information (E).

- **Protocol Flag.** All Bluewave device names start with a standardized string ("GCF"). This is used to determine if a newly discovered device is Bluewave compatible.
- **Device ID.** The second field contains a device specific identifier. By default, this field contains a device's Android ID. However, any unique value (i.e., user specified names) can also be used. 3
- **Context URL.** The third field contains a URL to the device's context broker. This is the link that other devices will use to request this device's context.
- **Access Key.** The last field contains a 12-character alphanumeric key. When other devices want to download context, they include the key in their request. The broker then validates the key before returning data. The access key is regenerated after every context update. This temporary credential prevents devices from accessing each other's context once they move out of range.

Task 3: Obtaining Context from Other Devices. The final responsibility of the client service is to obtain context from other devices. When directed to by an application, the client service performs periodic Bluetooth scans. As Bluewave devices are found, the service extracts the URL and access key, contacts the appropriate context broker, and forwards the information to the application for processing.

There are two technical hurdles to collecting context. The first is managing repeat discoveries. Due to the nature of Bluetooth discovery, client devices will repeatedly detect the same devices if they perform consecutive scans. To prevent devices from downloading the same context twice, each device maintains an archive of devices that it has encountered in the past. Whenever a Bluewave ID is detected, the service checks the access key to see if it has been altered. This allows the service to only download context when a new version is posted.

The second challenge in downloading context is minimizing the amount of extraneous data that is downloaded. Many context-aware applications only need a small amount of context in order to function properly. A public display, for example, may need to know the user's language preferences in order to make sure that its contents are properly translated. To prevent devices from downloading a device's entire context, Bluewave requires applications to specify the JSON elements that they need in their HTTP/S request (Figure 51c). The client service will then only receive these elements from the broker.

By packaging all three capabilities into a single module, the client service provides a simple interface to request and receive context. Since the client service is always running as a background process, developers can use it to share context, regardless of whether their application is running. More importantly, because the client service's functions are modularized, developers can turn on/off features as needed. For example, they could share context, but not scan for it. This minimizes our system's overhead (an examination of Bluewave's battery consumption is provided in our validation).

4.4.1.2. Context Broker

Context brokers are cloud servers that are responsible for storing and retrieving context. They serve as trusted middlemen, and allow devices to share context without having to directly connect with each other.

Context brokers have two main responsibilities. First, they provide a standardized interface for uploading and downloading context. Each context broker contains a set of PHP files that allow client services to invoke its services. To upload context, for example, the client service performs an HTTP post to *uploadContext.php*. Similarly, to download context, the client service performs an HTTP request to *getContext.php* (Figure 51). By leveraging open web technologies, our design allows any web server to act as context brokers. This lets Bluewave run on existing hardware, and allows users to stand up their own context brokers if they want explicit control over where their data is stored.

Secondly, context brokers provide authentication services. When client services connect to the broker for the first time, they receive a set of authentication credentials. The client service must then provide these credentials back to the broker each time it uploads new context. Combined with secure transmission protocols such as HTTPS, this approach prevents devices from being able to modify each other's context at will. This allows devices to be confident that the context they download has not been tampered with.

While Bluewave relies on context brokers to share context, it is important to note that our system neither assumes nor requires that devices use the same broker. Instead, by including a self-contained URL to their broker in the Bluetooth name, Bluewave supports both centralized and decentralized configurations.

4.4.1.3. Supporting User Privacy

The idea of using beacons to share context has significant implications from a user privacy standpoint. While our work assumes that users would be willing to share some context (*e.g.,* shopping lists, language preferences) with their environment, prior research has shown that preserving user privacy is an important consideration when designing Ubicomp systems [70]. At the same time, however, prior work has also shown that users are willing to share information with nearby users and devices, especially when they are near each other [121]. Consequently, we were interested in learning 1) how users would react to the idea of broadcasting context, and 2) what types of safeguards needed to be added to Bluewave to make it socially acceptable and useful.

To answer these questions, we conducted a user study with 15 participants (8 males, 7 females; 21-73 years old). For this study, we created three probes (described below) using an early version of Bluewave that lacked privacy controls. We then had participants try these applications, and conducted a series of semi-structured interviews in order to elicit feedback and rate their overall comfort levels.

Preliminary Comfort with Sharing Context

To establish a baseline before the probes were introduced, participants first filled out a survey that asked them to rate how comfortable they are with openly sharing various types of context through their mobile device. This survey consisted of 12 items, 8 of which are widely accepted as being personally identifiable [80], and 4 of our own design. Each item was rated using a 5-point Likert scale (1 = "very uncomfortable"; 5 = "very comfortable"). Additionally, participants were instructed to consider each type of information separately, and to assume that sharing is limited to users/devices within a 30-foot radius.

The results of this survey are reported in Figure 52. Similar to prior work [121], our results show that there are a number of contexts that users are unwilling to share over any distance (*e.g.,* social security numbers, home addresses). Interestingly, though, our results also show that there are many types of information, *including personally identifiable information,* that users *were* willing to share over a short distance. For example, many participants did not mind sharing their gender, age or location, because they reasoned that this information could be determined just by looking at them. Other types of information, such as shopping lists, exercise habits, and dietary restrictions, were considered more personal, but okay to share since they could not be used to identify participants in a crowd. Finally, information

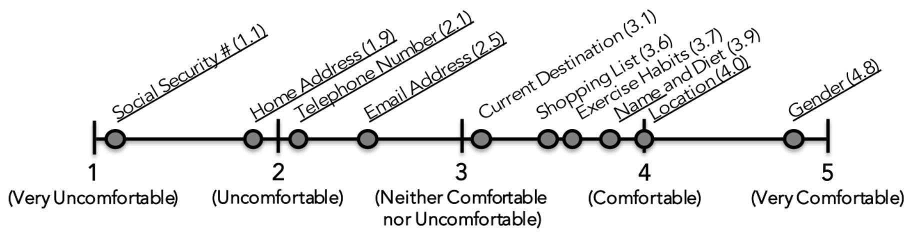


Figure 52. Participants' average comfort levels with sharing various types of contextual information. Underlined items indicate personally identifiable information.

such as email addresses, telephone numbers, and navigation destinations were more ambiguous, with only half of the participants willing to share them openly.

Overall, these results suggest that users would be willing to share context using a broadcast based system like Bluetooth. While participants were unwilling to broadcast all 12 types of information, all of them *were* open to the idea of sharing a subset over short distances. Admittedly, the results from this survey cannot be used to definitively say what contexts can and cannot be shared opportunistically. However, they do show that this distinction exists, and that there is a wider range of information that users might be willing to share than what one might assume.

Probe Applications

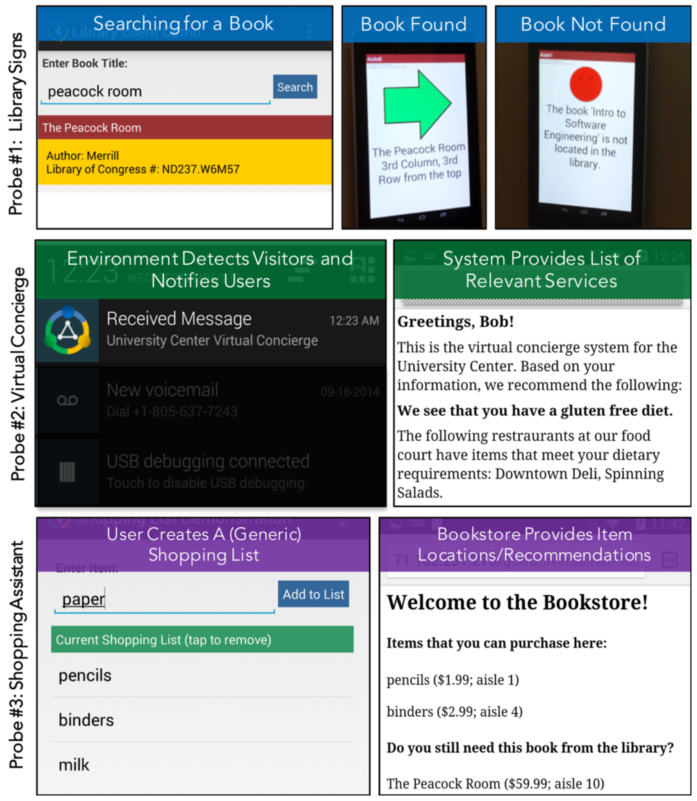
After the survey, we introduced participants to our probes. The criteria we used to select the three probes were they had to be typical applications from the literature, and that using standard development approaches, required significant *a priori* configuration in order to work. For probe 1, we created a series of intelligent signs for our institution's library. Using a custom Android application (Figure 53, top left), participants could search through the library catalog and select a book to check out. The title and call number of the book was then shared with the signs via Bluewave, allowing them to either guide the user to the correct row/column, or inform the user that the book is not available (Figure 53, top right).

Probe 2 was a "virtual concierge" for public spaces. For this example, we deployed a series of Bluewave-equipped sensors at the entrances of our institution's University Center. By having users share their email address, dietary restrictions (*e.g.,* lactose intolerant), and exercise habits through Bluewave, our system could determine 1) if the user has been to the building before, 2) which restaurants he/she can order food from, and 3) what exercise facilities he/she might be willing to use. The system could then send a notification to users' phones, which welcomed them, and provided a list of services that they might find interesting (Figure 53, middle).

Finally, Probe 3 was a shopping assistant to help users find items in our institution's bookstore. Here, participants used a custom app to share their shopping list via Bluewave. As participants entered the bookstore, specially placed sensors scanned the user's shopping list and determined which items were available for purchase. This information was then presented to the user (Figure 53, bottom). Additionally, our shopping assistant also uses the context shared by our library application (Probe 1) to see if the title they were looking for is offered in the store. This provided users with an alternative way to acquire a book of interest—one they might not have considered when they first entered the store.

Post Study Comfort Levels

After using our probes, we let participants adjust their self-reported comfort levels from the first part of the study to see if our probes had any influence on their willingness to share context. *In all 15 cases*, we found that participants'


Probe #1: Library Signs
Searching for a Book
Book Found
Book Not Found
Probe #2: Virtual Concierge
Environment Detects Visitors and Notifies Users
System Provides List of Relevant Services
Probe #3: Shopping Assistant
User Creates A (Generic) Shopping List
Bookstore Provides Item Locations/Recommendations

Enter Book Title:
peacock room
Search
The Peacock Room
Author: Merrill
Library of Congress #: ND237.W6M57

The Peacock Room 3rd Column, 3rd Row from the top

The book 'Intro to Software Engineering' is not located in the library.

Received Message 12:23 AM
University Center Virtual Concierge
New voicemail 09-16-2014
Dial +1-805-637-7243
USB debugging connected
Touch to disable USB debugging.

Greetings, Bob!
This is the virtual concierge system for the University Center. Based on your information, we recommend the following:
We see that you have a gluten free diet.
The following restraurants at our food court have items that meet your dietary requirements: Downtown Deli, Spinning Salads.

Enter Item:
paper
Add to List
Current Shopping List (tap to remove)
pencils
binders
milk

Welcome to the Bookstore!
Items that you can purchase here:
pencils ($1.99; aisle 1)
binders ($2.99; aisle 4)
Do you still need this book from the library?
The Peacock Room ($59.99; aisle 10)

Figure 53. Probing applications developed for our user study.

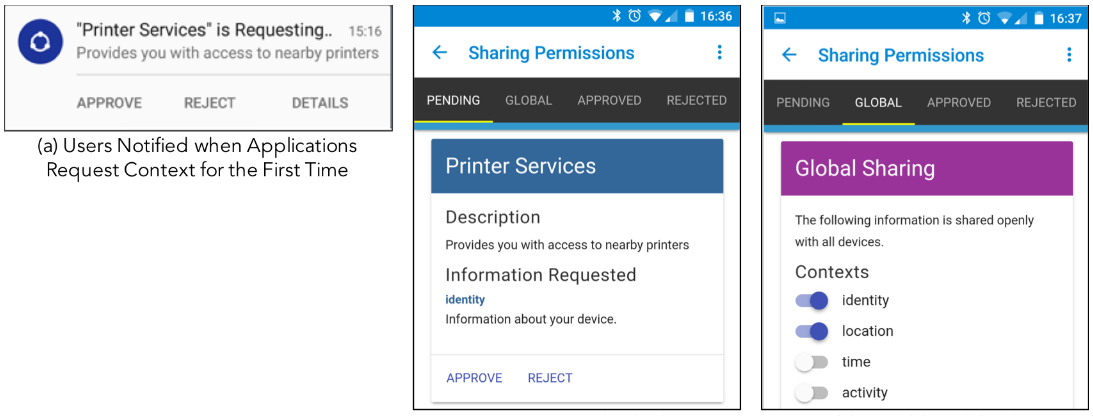
willingness to share information increased. While participants' opinions did not change for items that they had ranked high or low, they did increase their scores for middle ranked items (rating 2-4) by one point, with the most popular being shopping lists (9 out of 15 participants), exercise habits (6 out of 15), and diet (5 out of 15).

In follow-up interviews, we asked participants to explain the rationale behind their increased scores. In each case, participants told us that they were initially hesitant to share their context: "I don't want to give others [my information] and get nothing in return" (P9). Yet after seeing our prototypes, many stated that they had a better idea of how sharing context might benefit them. This made them more willing to have this information be openly available.

Designing Privacy Centric Features Based on Participant Responses

While our participants warmed up to the idea of sharing context, the issue of user privacy was brought up on numerous occasions. Many participants liked the services provided by our probes, but noted that they did not want others to have easy access to their information. Furthermore, while participants admitted that most of the information used by our probes was innocuous (*e.g.,* "I eat junk food-anybody can know that"—P13), they also noted that they would would be more willing to use Bluewave if they had explicit control over what information is being shared: "I would prefer the system to ask me before taking my information" (P6).

In light of these findings, we have added several privacy features to Bluewave. Instead of sharing context with everyone, Bluewave now requires developers to register their applications on our framework's website and receive an app ID. This ID must be provided to context brokers when requesting context (Figure 51e). Similar to existing app stores, this approach does not prevent rogue developers from registering a seemingly safe application. It does, however, prevent anonymous access, and provides a simple way to blacklist malicious apps once they are found.


"Printer Services" is Requesting.. 15:16
Provides you with access to nearby printers
APPROVE REJECT DETAILS

16:36
Sharing Permissions
PENDING GLOBAL APPROVED REJECTED
Printer Services
Description
Provides you with access to nearby printers
Information Requested
identity
Information about your device.
APPROVE REJECT

16:37
Sharing Permissions
PENDING GLOBAL APPROVED REJECTED
Global Sharing
The following information is shared openly with all devices.
Contexts
identity
location
time
activity

(a) Users Notified when Applications Request Context for the First Time

(b) Approve/Reject Individual Requests

(c) Enable/Disable Public Sharing

Figure 54. Bluewave's client service notifies users when applications request context for the first time(a). Users can then use our interface to share context with individual apps (b) or openly (c).

Additionally, context brokers are now able to track individual applications in order to see what types of information they ask for. Users will then receive a notification on their devices (*via* the client service) when applications request their context for the first time (Figure 54a), and be provided with an interface to grant or deny access (Figure 54b).

Finally, in light of our discovery that there are some contexts that users are comfortable sharing at all times, we also allow users to specify which contexts (*e.g.,* location, first name) they would like to share openly (Figure 54c). This prevents users from having to approve every request for context, while still allowing them to be notified in outstanding cases.

One downside of using an opt-in model is that users must now grant applications permission before they can receive context-aware services. Our probe suggests, however, that such safeguards are needed for now to give users peace of mind. Thus, while we expect users to become more comfortable with sharing context over time, our design recognizes that the decision to share is highly personal, and must be left to them.

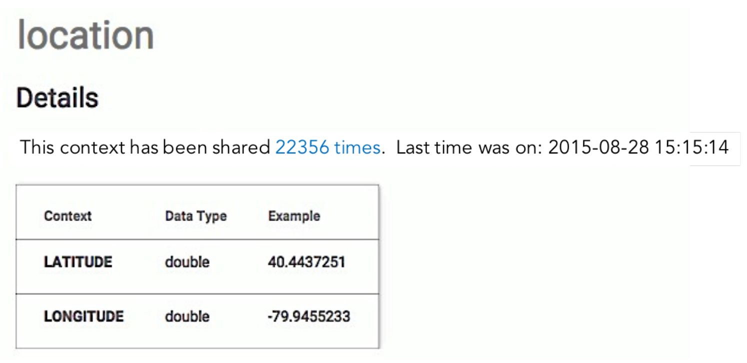
4.4.1.4. Supporting Developers

Bluewave supports developers in two important ways. The first way is by making it easy for developers to broadcast and listen for context. To use Bluewave, developers need to import GCF into their application, and create a group context manager. They can then access Bluewave's functionality by performing the following steps:

Step 1: Specify Context(s) to Upload. To share context, developers call the group context manager's UPLOADCONTEXT() method. They then pass the specific JSON object(s) that they would like to upload to the context broker, as shown below:

When the developer is done sharing context, they call the PUBLISH() method. This directs the client service to upload the information to the broker.

Step 2: Specify Context(s) to Receive. To receive context, developers must first register their application with our website. They can then use the RECEIVECONTEXT() method to provide Bluewave their unique app ID, as well as the specific JSON elements they are looking for, to the client service. The service will then automatically scan for devices and download the requested information.


location

Details

This context has been shared 22356 times. Last time was on: 2015-08-28 15:15:14

Context	Data Type	Example
LATITUDE	double	40.4437251
LONGITUDE	double	-79.9455233

Figure 55. Sample context description from Bluewave's web dashboard. This page is curated with live usage data, and allows developers to see which contexts are being shared, and how they are formatted.

Step 3: Process Incoming Context. Each time the client service downloads a new or updated context file, it broadcasts this information (along with standard Bluetooth discovery data, such as the MAC address and RSSI value) as an Android Intent ("ACTION_OTHER_USER_CONTEXT_RECEIVED"). Developers can then listen for this Intent in their application, and use this information.

The second way that Bluewave supports developers is by allowing them to see how context is being shared across our system. Each time a context broker receives a new type of context (*i.e.,* a JSON element with a novel tag), it uploads a sample of that information to our centralized database. This information is then used to create a web dashboard that documents 1) which contexts are available, 2) how they are defined, and 3) how often they are openly shared (Figure 55). Through our dashboard, developers can see which contexts are most commonly used across the entire Bluewave platform. This allows developers to utilize the same contexts in their application without having to explicitly coordinate.

The inclusion of "living documentation" gives Bluewave a significant advantage over other context-sharing systems. To date, other systems have assumed that developers will share context using a standard ontology. In contrast, Bluewave can share any JSON encoded context, making it compatible with a wide range of existing [50,93,115], as well as emerging standards [139,140]. In the following sections, we present nine applications built with Bluewave. We use them to validate the utility of our tools, and show how our system offers a simple but effective way to share context.

4.4.2. Validation

We validate Bluewave in two parts. In the next section, we show how the ability to openly broadcast and listen for context is useful in a wide range of applications. Afterwards, we evaluate Bluewave's battery consumption and latency, and show that our system is practical along both dimensions.

4.4.2.1. Example Applications

As an initial proof of concept, we describe nine prototype applications that were created using Bluewave. These applications are split across two domains, public displays and the Internet of Things (the first of which has been of long-standing interest in research and the second of which has recently attracted a lot of attention in industry), to demonstrate the types of opportunistic applications that are enabled through our system.

Additionally, our applications also showcase Bluewave's ability to reuse context. Rather than create all nine examples ourselves, we gave Bluewave (and our web dashboard) to 3 different developers, and had them create the applications

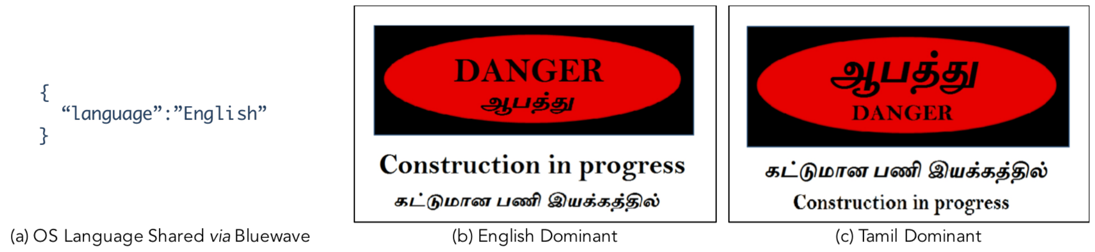

```
{
  "language":"English"
}
```

Figure 56. Self-translating sign demonstration. Users share their device's operating system language *via* Bluewave (a), and signs can automatically adjust their contents based on the number of nearby users that speak each language (b-c).

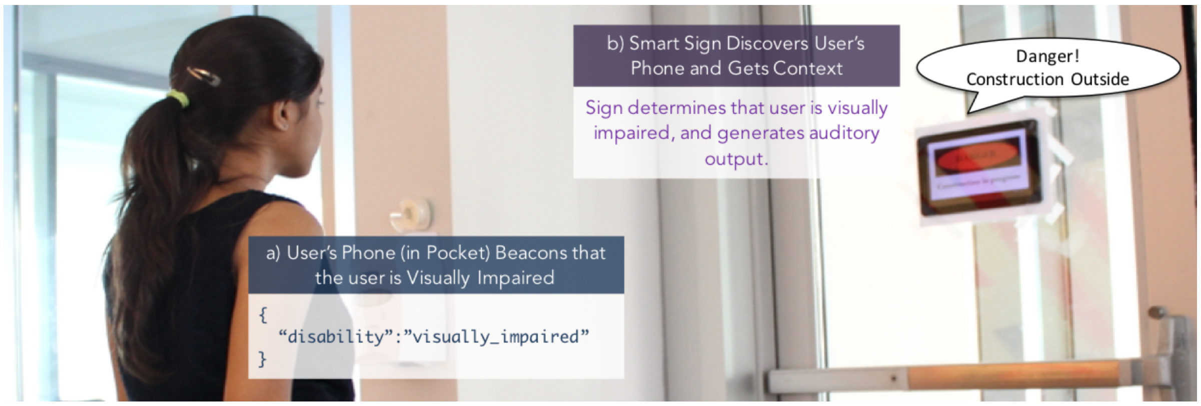

b) Smart Sign Discovers User's Phone and Gets Context
Sign determines that user is visually impaired, and generates auditory output.
Danger! Construction Outside
a) User's Phone (in Pocket) Beacons that the user is Visually Impaired
```
{
  "disability":"visually_impaired"
}
```

Figure 57. Smart sign demonstration. Users share their disability using Bluewave (a), and smart signs can modify the way they output their contents (b).

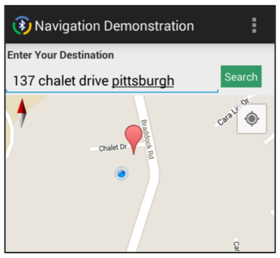
using our system. In doing so, we not only show that Bluewave is easy to understand and use, but also demonstrate how our system lets developers use the same context(s) in multiple ways.

Public Displays

A key challenge in creating public displays is scalability. While prior research has shown that we can intelligently modify the contents of signs based on users' context, the resulting systems require users to manually connect to the sign [5,84], or register with a central service [17,49,57]. This works when the number of public displays is small, but becomes impractical if we want to expand this functionality beyond a single environment or to a large number of users.

Bluewave provides a simple but elegant solution to this problem. Instead of requiring users to explicitly connect to every sign, our system lets signs scan and collect context from users. The signs can then adjust their contents based on users' collective information needs. To demonstrate this, one of our developers created a series of self-translating signs using our system (Figure 56). For this demonstration, users share their operating system language settings *via* Bluewave. The signs then detect the language and translate their contents to accommodate the most common language(s) for the users nearby. In later work, the same developer augmented the signs to support visually impaired users. By having users share their disability via Bluewave and using Bluetooth RSSI to estimate range, the signs are able to detect when a user with special needs is nearby, and assist them by speaking out loud (Figure 57).

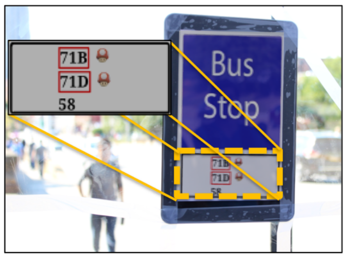
Bluewave was also used to create navigation aids. We created a simplified version of Google Maps (Figure 58a) that shares users' navigation destination (latitude/longitude coordinates) through Bluewave. A developer then created a smart bus stop sign that can listen for this information, and tell the user which bus routes will travel near this location



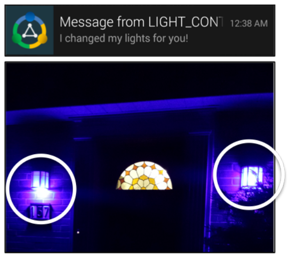
a) User inputs navigation destination into a map app on his/her phone

```
{
  "destination": {
    "latitude":40.434090,
    "longitude":-79.853565
  }
  "icon": "mushroom.jpeg"
}
```

b) App shares destination coordinates using Bluewave



c) Smart bus sign recommends busses to take to go to destination (and places user specified icon)

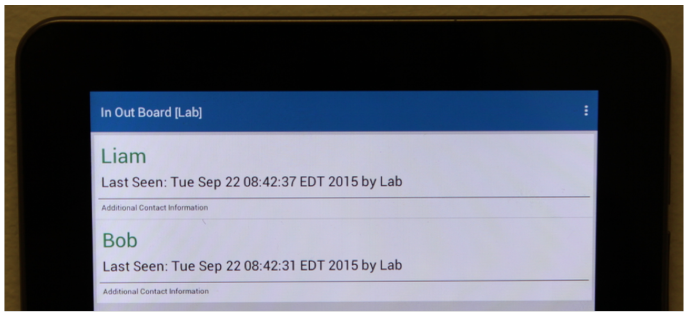


d) House notifies user when near destination (top), and alters porch lights (bottom)

Figure 58. Navigation aide demonstration. By inputting their destination address into a Bluewave compatible map app (a) and sharing it (b), bus signs can tell users which routes to take (c), and the destination can notify users when they are nearby (d)

```
{
  "name": {
    "first":"Liam",
    "last":"de Freitas"
  }
}
```

a) User shares name using Bluewave


In Out Board [Lab]

Liam
Last Seen: Tue Sep 22 08:42:37 EDT 2015 by Lab
Additional Contact Information

Bob
Last Seen: Tue Sep 22 08:42:31 EDT 2015 by Lab
Additional Contact Information

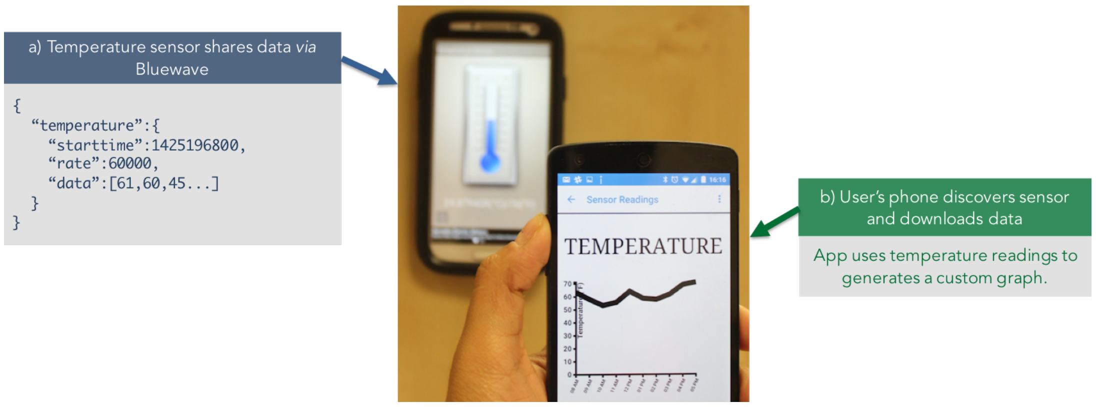
b) Sign performs Bluetooth scans, and displays nearby users.

Figure 59. In Out Board demonstration. Users share their name using Bluewave (a), and the sign uses this information to generate a list of who is nearby (b).

(either by drawing a red box around the route name, or displaying a user-specified icon; Figure 58c). Another example, created by a different developer, uses the same context to control a house's porch lights (Figure 58d). Here, a Bluewave-equipped sensor placed at a house searches for users that are navigating towards it. When found, the sensors alter the color of the porch lights and sends the user a message (e.g., "Look for the purple lights").

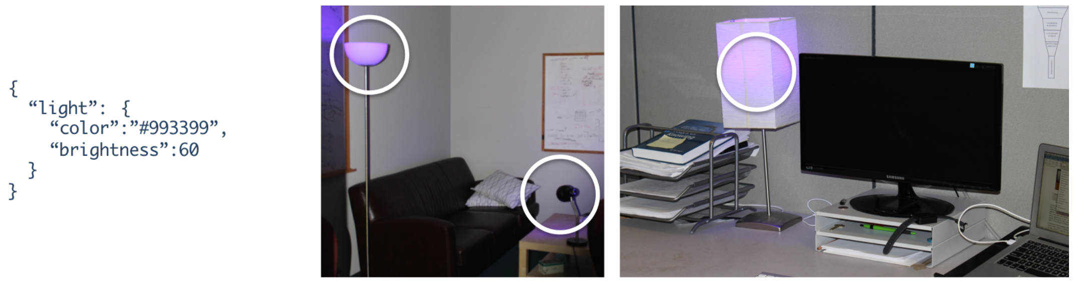
We also used Bluewave to recreate classic context-aware signs, such as the In-Out Board [100]. Our "sign" is a modified tablet that has been placed at the entrance of our lab (Figure 59). To use our system, users share their name via Bluewave. The sign can then display this name, as well as the time they were detected so that other users can know who is in the office. Unlike the original In-Out Board, which requires users to place a dedicated hardware device (*i.e.,* an "iButton") on the sign to check in/out, our system uses Bluewave to continually determine who is nearby. This eliminates the need for users to directly interact with our sign, and allows it to seamlessly work with visitors and changing lab membership—a feature that the original system was not designed to handle.

In each case, Bluewave improves upon the state of the art by allowing public displays to provide users with customized services without having to know of them in advance. Clearly, we kept these prototypes simple to illustrate the main benefits of using Bluewave, and did not address usability issues such as how to create a self-translating sign that can handle large numbers of users. Yet even in their current form, these examples show how sharing a small amount of information via Bluewave substantially increases users' access to timely and relevant information.


a) Temperature sensor shares data *via* Bluewave
b) User's phone discovers sensor and downloads data
App uses temperature readings to generates a custom graph.
Sensor Readings
TEMPERATURE

```
{
  "temperature":{
    "starttime":1425196800,
    "rate":60000,
    "data":[61,60,45...]
  }
}
```

Figure 60. IoT sensor demonstration. Environmental sensors share temperature values *via* Bluewave, and nearby devices are able to download and view this information without having to directly connect to the sensor.



```
{
  "light": {
    "color":"#993399",
    "brightness":60
  }
}
```

a) User shares preferred light color and brightness settings *via* Bluewave

b) Using lighting preferences at home

c) Using lighting preferences at work

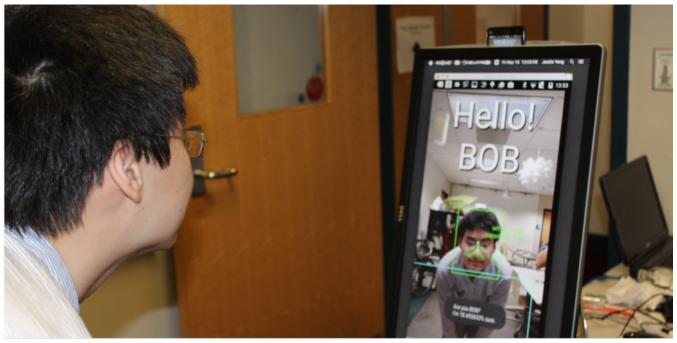
Figure 61. Migrating preferences demonstration. Users share their preferred lighting settings *via* Bluewave (a), and these settings follow them from their home (b) to their office (c).

Internet of Things

We also explored how Bluewave can be used to support the Internet of Things (IoT). As an initial demonstration, we have created a series of remote sensors (*i.e.,* tablets/phones) that can monitor and share their readings using Bluewave. Users can then collect and view this information from their mobile phones (using a scanning app created by another developer) without having to directly connect with each sensor (Figure 60). Since Bluetooth scans can take between 15-30 seconds, our early prototypes focused on sensors whose readings do not quickly change (temperature, barometric pressure, humidity). Since then, however, we have created sensors that can share TCP connection settings (*e.g.,* IP address/port) in addition to sensor data. This lets devices connect to individual sensor streams, and shows how Bluewave can support occasions where real-time context (*e.g.,* microphone data) is needed.

Another use for Bluewave in IoT is to transfer user preferences between smart environments (Figure 61). Inspired by systems such as Aura [107], which first introduced the idea of cross-environment syncing, we created a mobile app that allows users to adjust the color and brightness of Wi-Fi connected light bulbs in their room. These settings are then shared via Bluewave, and used to configure the lights when the user enters a new location (*e.g.,* an office and/or hotel). While this prototype focuses on light settings, we are also looking at how the same technique can be used to migrate other settings (*e.g.,* preferred temperature). This would allow environments to intelligently configure themselves without requiring the user to perform manual configuration.

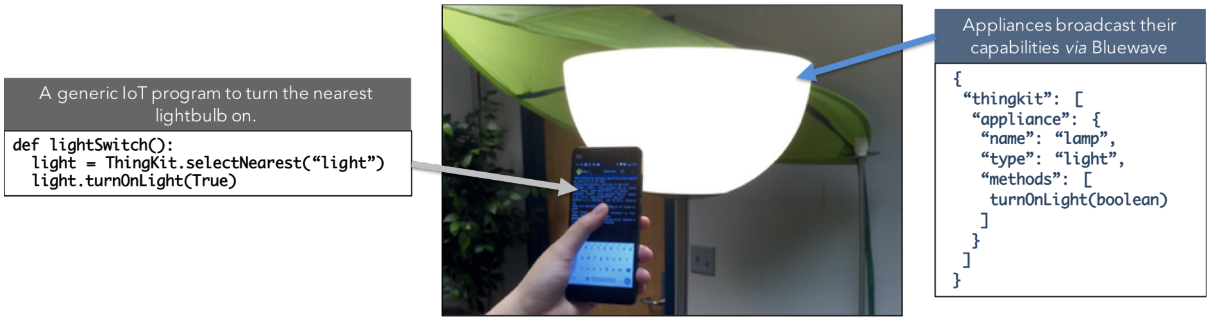
```
{
  “name”: {
    “first”:”Bob”,
    “last”:”Nobody”
  },
  “face”: [
    “http://…/bob1.jpeg,
    “http://…/bob2.jpeg
  ]
}
```



a) User shares name and photos of his face *via* Bluewave

b) Other devices can use this information to train a classifier, and identify the user when they meet for the first time

Figure 62. Facial recognition demonstration. Users share their name and headshot photos *via* Bluewave (a). This lets camera equipped devices identify users in arbitrary environments (b).


A generic IoT program to turn the nearest lightbulb on.
Appliances broadcast their capabilities *via* Bluewave

```
def lightSwitch():
  light = ThingKit.selectNearest(“light”)
  light.turnOnLight(True)
```

```
{
 “thingkit”: [
  “appliance”: {
   “name”: “lamp”,
   “type”: “light”,
   “methods”: [
    turnOnLight(boolean)
   ]
  }
 ]
}
```

Running the application in an IoT environment.

Figure 63. ThingKit demonstration. Appliances use Bluewave to broadcast their capabilities (right). Developers can then create generic IoT applications to control these appliances when they are opportunistically discovered (left, center).

A third use for Bluewave in IoT is to support *ad hoc* facial recognition (Figure 62). For this example, one of our developers created an app that lets users upload photos of themselves to a web server. The URLs to these photos, along with the user’s name (using the same format as our In-Out Board example), is shared *via* Bluewave. Camera-equipped devices can then obtain this context, and use it to identify specific users in the environment. This allows devices to tell if a user is looking at them (rather than assume that this is the case when they are in Bluetooth range) without requiring 1) developers to create a centralized face database, or 2) users to manually register their face with every environment.

Our fourth, and most ambitious use of Bluewave to date is as a tool to create portable IoT applications. As part of our ongoing work, we have deployed a series of smart appliances in our lab that can share their generic type (*e.g.,* lights, speakers), capabilities (*e.g.,* turnOnLight(), playSound()), and network connection details (*e.g.,* IP Address, port) *via* Bluewave (using smartphones as proxies, as suggested in [82]). We then created ThingKit, a custom Python environment that can dynamically discover these appliances and convert them into in-code objects. When developers use ThingKit, they can programmatically control a single appliance (*e.g.,* “light1.turnOnLights(true)”), or all appliances of the same type (*e.g.,* “lights.turnOnLights(true)”). This allows the user to run a single application (*e.g.,* “turn on the nearest light when I get an email”) that works in multiple locations (Figure 63).

Collectively, these examples demonstrate how the ability to openly share context is relevant from an IoT perspective. Currently, research in IoT has largely focused on the technological challenges associated with instrumenting a single

Table 12. Results from our battery drain tests, showing the average power consumed running Bulewave in various configurations. Each test was run for eight hours, and repeated 5 times.

Configuration	Avg Battery Drain	Std Dev
Control (Bluetooth Not Discoverable)	2.8%	1.3%
Configuration 1: (Discoverable Only)	4.6%	0.55%
Configuration 2: (Discoverable/Scanning)	12.4%	0.89%

Table 13. Results from our latency tests, showing the average time needed to detect and download 10 devices' context (1KB each) using various techniques. Each configuration was tested 30 times.

Technique	Avg Download Time	Std Dev
Bluewave	11.6s	5.5s
Bluetooth (Direct Connection)	55.8s	13.5s
Physical Web + Download (BLE)	9.3s	2.9s

environment, such as a home or office. Our work complements this growing body of research by showing how our technology supports services that can span multiple smart environments. This increases users' access to services, while simultaneously minimizing the need for configuration and setup.

4.4.2.2. Battery Consumption

In order for Bluewave to be useful for long-term context sharing, it needs to be power efficient. To verify this, we conducted a series of power drain tests using identically configured Samsung Galaxy S IV smartphones. To establish a baseline, we first measured the power drain on the devices when idle (*i.e.,* no applications or Bluetooth running). We then had each device share a 1KB JSON file (the collective amount of context shared by all of our prototypes) using Bluewave, and tested two configurations for eight hours each (each test was run 5 times):

1. **Configuration 1:** The device is only broadcasting context (*i.e.,* Bluetooth name is set to continuously discoverable).
2. **Configuration 2:** The device is both sharing context and listening for context from others (*i.e.,* Bluetooth discovery is restarted every 30 seconds, and Bluetooth name is set to be continuously discoverable).

As expected, using Bluewave leads to reduced battery life (Table 12). However, the exact amount varies depending on which configuration is used. When Bluewave is used to share context (Configuration 1), our system only consumed an additional 1.8% of battery power after eight hours. When the device is both sharing and listening for context (Configuration 2), Bluewave's energy usage was about four times as much as the baseline condition, but still relatively low at 12%.

These results are promising for two reasons. First, they show that it is feasible to use Bluewave for extended periods of time. While we do not expect user devices to have to constantly scan their environments and share context, our results show that their devices can do so for a long period of time without depleting their battery.

Second, our results also show that, in the vast majority of use cases, Bluewave's impact on battery life is minimal. Of the 12 prototypes presented in this section, only 2 (Figure 60 and Figure 63) required the user's device to scan the environment; the rest had users share their context for the environment to scan. As Bluewave is intended to allow users to share context with the environment, we expect that users will only need to have their Bluetooth set to discoverable to benefit from our system. This lets Bluewave run in its low power configuration for most use cases, while still giving users the option to scan for context when needed.

4.4.2.3. Latency

In addition to power efficiency, we also wanted to see how Bluewave's latency compared to other beacon based technologies (*e.g.,* Google's Physical Web). To investigate this, we positioned 10 Bluewave devices throughout an office area, and timed how long it took for an 11th device (a Samsung Galaxy S IV) to scan and download each one's context (*i.e.,* the same 1KB JSON file used in the previous test). We then compared this to the time needed to download the same information by 1) connecting to each device via Bluetooth (assuming no PIN is required), and 2) using Physical Web beacons (*i.e.,* using Bluetooth LE (BLE) to discover each beacon, extract the URL, and download the file). Each method was tested 30 times.

Our results (Table 13) show that Bluewave's latency outperforms, or is at least comparable to, industry standard techniques. Our system discovers and obtains context nearly five times faster than is possible using direct Bluetooth pairing. This is because Bluewave is able to discover devices and download context in parallel, while Bluetooth must individually pair with each device before downloading its context. Our system *is* slower than using Physical Web beacons to share context (Table 13, row 3) due to Bluetooth's longer advertisement interval. Our results, however, show that the difference is, on average, 2.3 seconds. Practically speaking, this means that applications that can tolerate BLE's latency should also work with Bluewave.

These results show that Bluewave offers a good compromise between speed and flexibility. While our system is not the fastest, it offers similar performance to state-of-the-art technologies. This, combined with Bluetooth's pervasiveness and our system's integrated privacy tools, makes Bluewave a compelling alternative to current context-sharing techniques.

4.4.3. Discussion

Bluewave demonstrates how the ability to manipulate Bluetooth names significantly increases devices' abilities to opportunistically share context. At the same time, however, it also raises several interesting usability and design issues. In this section, we discuss how manipulating Bluetooth names and making devices permanently discoverable can negatively impact the end-user experience, and offer mitigating solutions. We then examine our prototypes' reliance on specialized applications to share context, and show how this dependency should decrease as our technique becomes more widely used. Finally, we reexamine our decision to use Bluetooth as opposed to Bluetooth Low Energy, and show how our technique can be generalized to support other technologies.

Tradeoffs for Increased Usability

Our work has identified two key challenges with using Bluetooth names to share context:

Less User-Friendly Bluetooth Names. Bluewave programmatically modifies a device's Bluetooth name to encode a flag, device ID, URL, and temporary key. While the resulting strings (Figure 50) are easily read by a machine, our modifications may make it harder for users to find the device(s) they want to pair with.

However, our approach fully preserves the user friendly device name, and we added a helper method to our middleware to extract the Device ID from a Bluewave name. Applications can use this method to only show the user friendly name (*e.g.,* "Bob's Phone") as opposed to its system generated name (*e.g.,* "GCF::Bob's Phone::...") when users perform a Bluetooth scan. In future work, we are also looking at incorporating this preprocessing step at the OS level. This approach is more difficult to implement, but would completely hide Bluewave's naming scheme from users.

Increased Opportunities to Track Users. Bluewave requires devices to be continuously discoverable so that they can advertise updates to their context. This eliminates the need for a priori coordination, but also creates new opportunities for applications and/or users to track users by their (static) device IDs as they move throughout an environment.

However, this is a potential risk with all context-aware systems and not unique to Bluewave. Many popular technologies, such as websites, credit cards, and personal digital assistants (*e.g.,* Google Now) already monitor users' locations and/or activities to offer personalized services, and can be used to track users without their knowledge or express permission. Compared to these technologies, Bluewave's ability to track the user is highly localized. Since Bluetooth's range is effectively capped at 15-30 feet, developers would need to deploy a large number of Bluetooth scanners to track a user over longer distances. Importantly, since our system only works when Bluetooth is active, users can easily turn off Bluetooth discovery if they do not want to be tracked. This gives users absolute control over when their context is shared, and allows them to use our system only if and when they feel comfortable.

The Need for Dedicated Context-Sharing Applications
Many of our prototypes rely on custom apps to publish and share user information. For example, before users can receive recommendations from our intelligent bus displays, they need to specify their destination using our map application. Similarly, before our facial recognition system can identify a person, the user must first use our app to collect and share her photograph and name.

The need for dedicated apps imposes a requirement on our opportunistic context-sharing technique. In the absence of better context-sharing apps, our own apps were required to show how Bluewave might be used today. However, it is conceivable that the contexts used by our examples could be collected and shared directly by the operating system, or by OS-standard apps (*e.g.,* a map app). This would reduce the need for specialized apps for many commonly used contexts, while still providing systems with the information they need.

It is also important to note that the apps used in our examples are not system-specific. For instance, our map application (Figure 58a) does not just share the user's destination with bus signs (Figure 58c), but rather with any system that needs this information and is given permission by the user (such as the porch light system depicted in Figure 58d). Similarly, our facial recognition app (Figure 62) does not just share user photographs with a single environment, but with any environment that requests it, if approved by the user. This lets camera-equipped devices recognize nearby users when they meet for the first time. This makes our context-sharing technique more general than most application-specific solutions, and is useful for sharing contexts between apps. The ability to share context between apps provides opportunities for new context-aware systems that work together in symphony and enable richer applications.

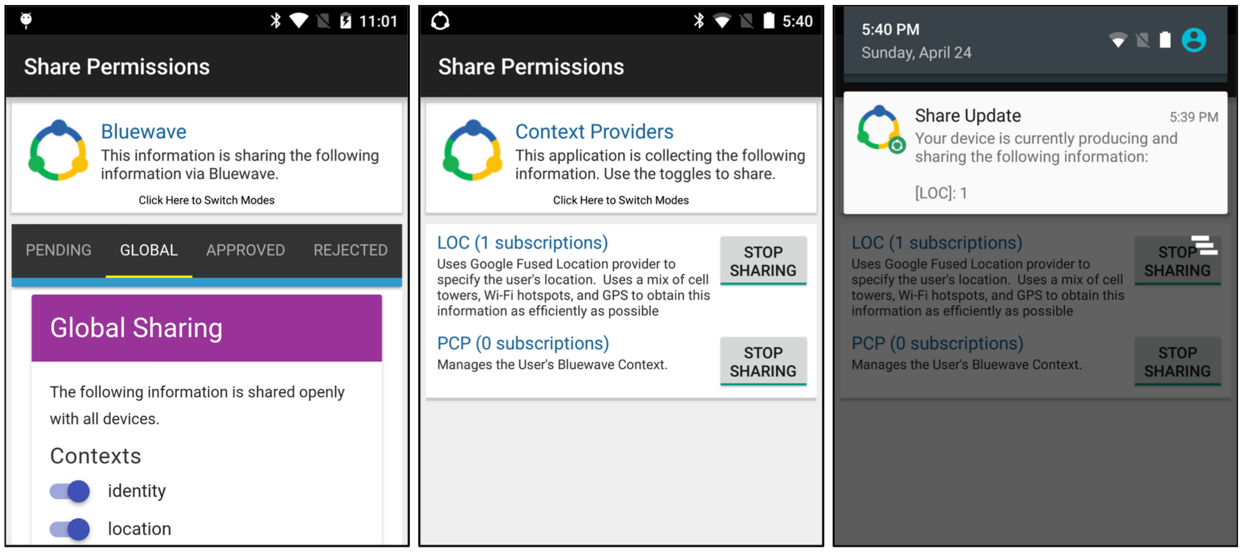
4.4.3.1. Bluetooth vs. Bluetooth Low Energy

Bluewave currently relies on "classic" Bluetooth technologies in order to share context. However, Bluetooth 4.0 (*i.e.,* Bluetooth Low Energy) has been ratified since 2010, and is available on commercial hardware. While our experimental results show that it is feasible for devices to both scan for devices and be discoverable using Bluetooth for a single day, BLE devices are known to be even more efficient, and can operate for *years* on a single charge [62].

Our decision to base Bluewave on Bluetooth was made mostly for compatibility reasons. Bluetooth is an established standard and adoption of new technologies such as BLE is much slower than one might think. To date, only 60% of Android devices are able to scan for BLE devices [141]. Furthermore, out of the 4,600+ Android models currently available to consumers, only the Nexus 6 and 9 (as of this writing) have the ability to serve as BLE beacons [142]. In contrast, classic Bluetooth is available on nearly every mobile device, and allows devices to both scan for devices and be discoverable at the same time. We have successfully used Bluewave on devices running Android Gingerbread (a 6-year-old operating system as of this writing). Thus, while BLE is technically more efficient and may one day replace Bluetooth, Bluetooth's ability to run on almost any hardware, combined with its acceptable energy use and latency, makes it the preferred solution for now.

Fortunately, Bluewave is not inherently tied to a single technology. The technique that we present, as well as the backend architecture used for storing context and managing user privacy, is generalizable to any technology that can broadcast a customizable link. Google's Physical Web project [135], for example, shows that it is already possible to share URLs *via* BLE beacons. Consequently, future versions of Bluewave can easily be created to support BLE and other emerging technologies as they become more widely available.

4.4.4. Lessons Learned



This information is sharing the following information via Bluewave.

The following information is shared openly with all devices.

This application is collecting the following information. Use the toggles to share.

Uses Google Fused Location provider to specify the user's location. Uses a mix of cell towers, Wi-Fi hotspots, and GPS to obtain this information as efficiently as possible

Manages the User's Bluewave Context.

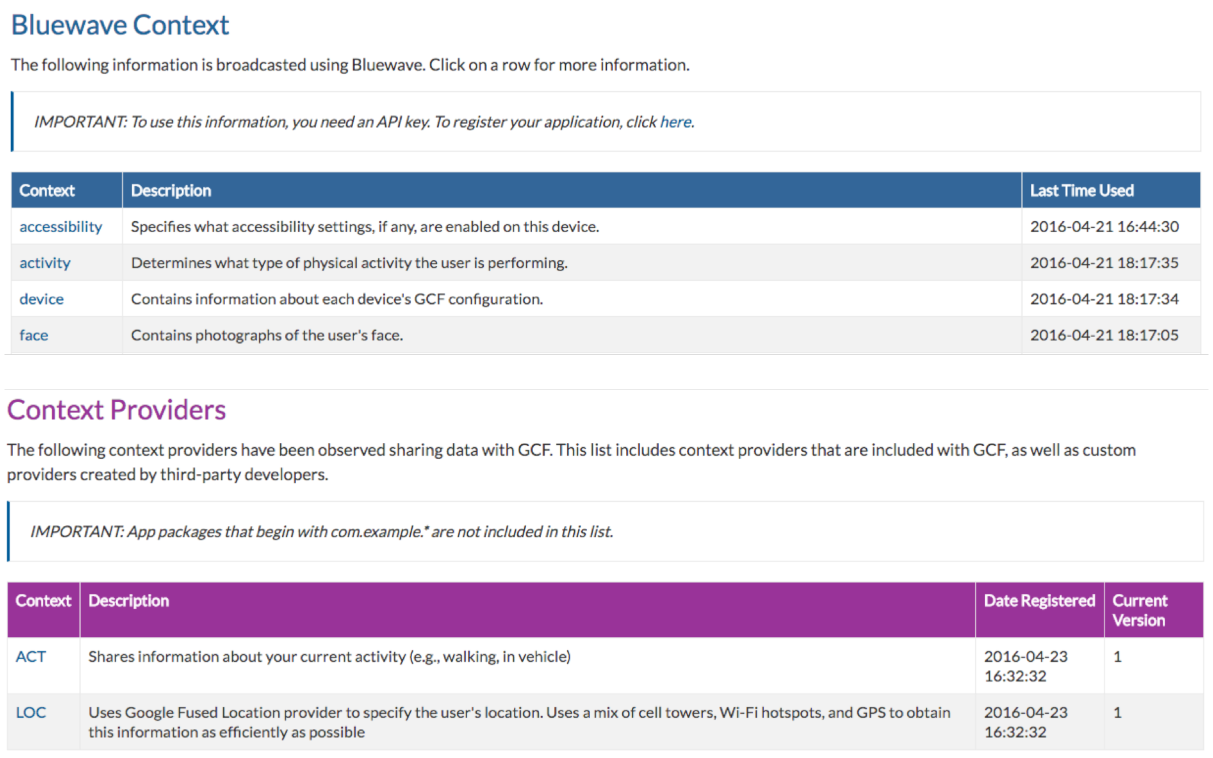
Your device is currently producing and sharing the following information:

a) Bluewave Permissions b) Active Context Providers c) Notifications when Another Device Subscribes to its Context Providers

Figure 64. GCF's updated permission screen allows users to control what information is shared *via* Bluewave (a) or by context providers (b). Applications also automatically generate notifications whenever they share context with other devices (c).

In addition to augmenting GCF's ability to detect nearby devices and form groups, Bluewave also increases our understanding of the challenges of sharing context from both an end-user and developer standpoint. In this section, we highlight three important lessons that we learned from Bluewave's development, and show how they have been incorporated into GCF's design.

The Importance of End-User Privacy Controls. A key takeaway from Bluewave is that users want explicit control over when and how their context is being shared. During our pilot study, we learned that our participants were open to the general idea of sharing context with other (*i.e.,* new or unfamiliar) devices over a small geographic radius. In each case, however, we found that participants were more comfortable doing so when they knew 1) what device(s) and/or entities were requesting this information, 2) what information and services they would provide to the user in exchange. This was true even when the information being shared, by participants' own admission, was too innocuous ("It's just a book"—P3) to be used to personally identify or harm them.


Bluewave Context

The following information is broadcasted using Bluewave. Click on a row for more information.

IMPORTANT: To use this information, you need an API key. To register your application, click here.

Context	Description	Last Time Used
accessibility	Specifies what accessibility settings, if any, are enabled on this device.	2016-04-21 16:44:30
activity	Determines what type of physical activity the user is performing.	2016-04-21 18:17:35
device	Contains information about each device's GCF configuration.	2016-04-21 18:17:34
face	Contains photographs of the user's face.	2016-04-21 18:17:05

Context Providers

The following context providers have been observed sharing data with GCF. This list includes context providers that are included with GCF, as well as custom providers created by third-party developers.

IMPORTANT: App packages that begin with com.example. are not included in this list.*

Context	Description	Date Registered	Current Version
ACT	Shares information about your current activity (e.g., walking, in vehicle)	2016-04-23 16:32:32	1
LOC	Uses Google Fused Location provider to specify the user's location. Uses a mix of cell towers, Wi-Fi hotspots, and GPS to obtain this information as efficiently as possible	2016-04-23 16:32:32	1

Figure 65. Screenshot from GCF's updated dashboard. This page is automatically updated by the framework, and allows developers to see what information is available, and how it is formatted.

In response to these concerns, we added a number of privacy-centric features to Bluewave. As mentioned above, our system now generates a system notification when a nearby device requests context from them (*e.g.,* "navigationDestination") for the first time. Users can then see which application(s) want context, and grant access on a case-by-case basis. Additionally, Bluewave provides users with a web-based console so that they can review their sharing preferences across all applications. This tool lets users modify permissions (*e.g.,* revoke access) after the fact, and gives them the option to define global sharing policies so that they do not need to grant every application access.

Although we had originally intended for these tools to be used with Bluewave, we eventually came to realize that giving users control over their context would be useful across all of GCF. Inspired by this, we have added standardized permissions controls to our framework. Our new permissions screen (Figure 64) not only lets users control what information is currently being broadcasted *via* Bluewave (Figure 64a), but also allows users to see which context providers are currently loaded and what types of information they can provide (Figure 64b). Users can then specify which context provider(s) are allowed to respond to external requests, and which ones can only provide information to locally installed applications. Additionally, our system also lets users see which context providers are currently being used at any given time, and can even be configured to provide users with a notification when a context provider is being started by an external device (Figure 64c). This improves users' awareness over how their context is being collected and shared at runtime, which in turn helps them customize the framework to match their personal comfort levels.

By making these controls standard across *every* GCF application, our framework gives users a simple way to manage their information sharing preferences. To maximize the chances of finding and forming opportunistic applications, we would obviously prefer that every GCF application openly shared context. Our experience with Bluewave, however, has shown us that the desire to share is still highly personalized. Consequently, our controls give users the ability to restrict sharing for the interim, while still giving them the option to openly share if and when they are ready to do so.

The Need for "Living Documentation." Our work with Bluewave also highlights the importance of providing developers with accurate and up-to-the-minute documentation. As mentioned in our validation, we gave Bluewave to three external developers, and had them create applications using our system. While we had assumed that developers would be able to easily use our system to reuse the same context across multiple applications, we found that they had difficulty doing so because they did not know what context was available through our system, and how it was formatted. To overcome this, we created a simple web dashboard that let developers see what information was available through our system (Figure 55). This information was automatically generated by our context brokers, and gave developers a simple way to see what context they could use in their applications without having to explicitly coordinate with each other.

Through this experience, we realized that we needed to give developers a better idea of what information they can access through our framework. Based on this, we have modified Bluewave's dashboard so that it provides this information for all of GCF. Using our tool Figure 65 developers can still search for contexts that are shared *via* Bluewave. Additionally, however, they can *also* see which context providers are available through our system. Our modified dashboard keeps track of the number of applications that use this context providers, so that developers can more accurately gauge how likely it is that they will be able to take advantage of this type of information in real-world conditions. Most importantly, we have modified GCF so that this information is automatically uploaded to our dashboard. This ensures that the information on our website is always up-to-date, and maximizes the likelihood of developers taking advantage of opportunistic groups in their applications.

Supporting Opportunistic, Dynamic Context Sharing. Our final insight from Bluewave was learned while creating our prototype applications. When we created our remote sensor demonstration (Figure 60), we discovered that we could use Bluewave to let devices share *socket connection settings* instead of raw sensor readings. This let devices connect to the sensor and obtain live data, even when they are not connected to the same network.

Through this workaround, we realized that we could use Bluewave to overcome GCF's most significant technical limitation (*i.e.,* the inability to form groups and share real-time context in environments where no common network connectivity exists). When we originally created this application, we had to write custom code to 1) obtain the remote sensor's network connection details, 2) connect to the correct socket, and 3) request and receive using GCF's context request messages. Since then, however, we have made several additions to GCF to automate this process.

First, we have modified GCF so that it can advertise its capabilities using Bluewave. Each time the client service uploads context to the broker, our framework also inserts a system generated JSON element which contains 1) the context providers are currently registered on the device, and 2) the communications channel that it is listening on. A sample JSON element is provided below:

Additionally, we have modified the group context manager so that it automatically requests this JSON element whenever the device performs a Bluewave scan. When a developer calls SENDREQUEST(), the group context manager uses the results from the most recent Bluewave scan to determine which devices are nearby and can provide the requested context. The framework will then connect to these devices and request context from them. As devices move out of Bluetooth range, GCF automatically determines that they are no longer nearby and unsubscribes from their context providers. This allows the framework to seamlessly and dynamically switch context providers as the user moves from one location to another, without requiring developers to manually keep track of which devices are nearby.

```
“device”:{
 “deviceID”:”Nexus 5-A”,
 “battery”:50,
 “applications”:[
  {
   “name”:”Impromptu”,
   “contextproviders”:[“TEMP”,”LOC”,”CAL”]
   “comm”:{“mode”=“MQTT”,“ipAddress”=“epiwork.hcii.cs.cmu.edu”,“port”=1883}
  }
 ]
}
```

Figure 66. GCF generated context for a specific device. By using Bluewave to share information about 1) the types of context a device can provide, and 2) the network address the device is listening on, GCF can automatically form *ad hoc* network connections.

Through this approach, we strike a balance between functionality and efficiency. By giving GCF the ability to share system level information *via* Bluewave, we provide GCF with a way to request and receive both static and real-time context without requiring each device to be connected to the same communications channel. It is important to note, however, that our approach is not entirely automatic. In order to use this capability, developers must manually direct GCF to perform Bluewave scans, and specify the desired scan interval. This makes it easy for developers to take advantage of this functionality, while still giving them the option to turn it off when not needed.

4.5. Summary

In this chapter, we described four systems that explore how opportunistic groups can be utilized in context-aware computing. The first system, Didja, demonstrates how the ability to share and compare context across devices allows them to form more precise groups than what is possible using traditional proximity based techniques. The second system, Snap-To-It, shows how the ability to share context (*i.e.,* a photograph) with an environment allows users to quickly and seamlessly interact with the devices in their immediate surroundings. The third system, Impromptu, shows how the ability to form opportunistic groups can be extended to applications and services, thereby allowing users to access these programs without having to explicitly install them. Finally, the fourth system, Bluewave, shows how we can leverage Bluetooth to extend GCF’s ability to broadcast context to nearby devices. This increases the range of environments in which GCF can detect and form groups, and allows our system to work under a wider range of real world conditions.

The work described in this section has allowed us to make progress towards answering our second research question. By using each of these systems as probes, we have been able to create a number of representative examples that show how GCF can be utilized to support a diverse range of interactions between users, devices, and any combination of the two. As mentioned at the beginning of this chapter, the purpose of this exploration is not to provide a comprehensive list of *every possible* way that opportunistic groups can be used in context-aware computing. Yet by covering many common cases, we not only provide a foundation of understanding that can be extended and built upon in the future, but demonstrate how our framework provides the abstractions needed to facilitate these explorations.

We have learned a lot from creating these systems. From a technological standpoint, developing these systems has allowed us to validate GCF’s design, and show that the framework works as intended. Moreover, these systems have given us firsthand knowledge of GCF’s limitations, and have helped us refine our design and add additional features. In the following chapters, we summarize the modifications that were made to our framework, and show how developers can utilize its functionality in both new and existing applications. We then conduct a study to show that our framework works in a wide range of use cases, and that it has the abstractions needed to be used as a general purpose development and research platform.

5. Revisiting, Refining, and Using the Framework

In **CHAPTER 3**, we presented an *initial* implementation of the Group Context Framework. We described the framework's core abstractions and architectural components, and showed how it supported all four of the group types identified in our conceptual model. We then used GCF in **CHAPTER 4** to develop four context-aware appliances and/or systems. Collectively, these systems increased our understanding of how the ability to form opportunistic groups supports a wide range of serendipitous interactions with users, the environment, and information/services. They also gave us the opportunity to evaluate GCF's functionality, and identify areas where the framework is either in need of further improvement, or lacking altogether.

In this chapter, we show how our experience building these systems has helped us refine GCF. First, we summarize the changes that we have made to GCF to support Didja, Snap-To-It, Impromptu, and Bluewave, and show how our updated architecture addresses all of the functional shortcomings we identified in **CHAPTER 3.** Afterwards, we introduce a generalizable design process for creating opportunistic context-aware applications, and present three case studies to show how this process can help guide developers when using our framework.

This chapter continues our exploration of **RQ1** (*"How can we allow devices to form opportunistic groups and share context"*). Through our experiences in **CHAPTER 4**, we have identified a number of additional features that the framework needs to provide to allow devices to form opportunistic groups. In this chapter, we show how the lessons learned from our prior work have been incorporated into the framework. This increases our understanding of how we can support opportunistic groups at the technical level, and provides a functional implementation that addresses all of the issues we have identified through prior work.

5.1. Overview of Major Changes

Our experiences with Didja, Snap-to-It, Impromptu, and Bluewave have led us to make several modifications to GCF's functional requirements (for the complete listing, refer to Appendix B). In this section, we summarize these changes, and present our framework's updated architecture. Afterwards, we show how this updated framework addresses or mitigates all of the shortcomings identified in **CHAPTER 3**.

5.1.1. Communication Protocol Changes

As shown in Table 14, we have made two modifications to GCF's communication protocol in order to support a wider range of interactions between group members:

Supporting Direct Messaging. Based on our experience with Didja, we learned that there are times when a device needs to send messages to a specific set of devices rather than with every device in communications range. To provide this capability, we have added a new *destination* field to GCF's CommMessage base class (Table 14. When applications need to send a message (*e.g.,* a context request) to one or more devices, they populate this field with the ID(s) of the intended recipients. The communications manager will then examine the destination field of each incoming message, and only deliver it to the group context manager if 1) the device's ID is included in this list, or 2) the list is empty. By including this field in *every* GCF message, we give devices the ability to ignore messages that are not intended for them. This lets the framework deliver highly specialized information, even when all of the devices are communicating over the same broadcast channel.

Table 14. GCF’s revised communication message types. Changes are colored.

Message Type	Description	Example Message (Serialized as JSON)
Context Request	Denotes a device’s desire for contextual information.	`{ "deviceID":"Device A", "contextType": "LOC", "requestType": "SINGLE_SOURCE", "refreshRate": 10000, "destination": [], "payload": [], "messageType": "R", "version":1 }`
Context Capability	Denotes a device’s willingness to provide contextual information. Sent in response to a Context Request message.	`{ "deviceID": "Device B", "contextType": "LOC", "alreadyProviding": "FALSE", "batteryLife": "47", "heartbeatRate": 30000, "sensorFitness": 1, "destination": ["Device A"], "payload": [], "messageType": "C", "version": 1 }`
Context Subscription	Used to subscribe/unsubscribe from a context provider.	`{ "deviceID": "Device A", "contextType": "LOC", "updateType": "Subscribe”, "refreshRate": 10000, "heartbeatRate": 30000, "destination": ["Device B"], "payload": [], "messageType": "S", "version": 1 }`
Context Data	Used to share context data.	`{ "deviceID": "Device B", "contextType": "LOC", "destination": ["Device A"], "payload": "[\"LATITUDE=123.45\”,\"LONGITUDE=67.890\"]", "messageType": "D", "version":1 }`
Compute Instruction	Used to send directed commands to a context provider *after* a group has been formed.	`{ "contextType": "PRINTER", "deviceID": "Device A", "destination": ["Device B"], "command": "PRINT_WORD_DOCUMENT", "payload": "[\"Filepath=file.docx\"]", "messageType": "I", "version": 1, }`

Allowing Devices to Send/Receive Remote Commands. Our work with Snap-To-It has revealed that there are times when devices need to be able to send remote commands once they have formed a group. To support these use cases, we have created a new COMPUTEINSTRUCTION message type (Table 14, blue). When an application needs to send a remote command, they provide GCF with 1) the context type of the context provider, 2) the name of the custom command (*e.g.,* “PRINT”, “PLAY_MUSIC”), and 3) zero or more parameters. The message in then transmitted to the destination device’s context provider, where it is used to perform a predefined action (*e.g.,* print a document, start/stop music playback). Compute instructions give developers a highly customizable way to send arbitrary messages to a context provider without having to issue a new context request. This gives developers a simple way to create interactive applications using our framework.

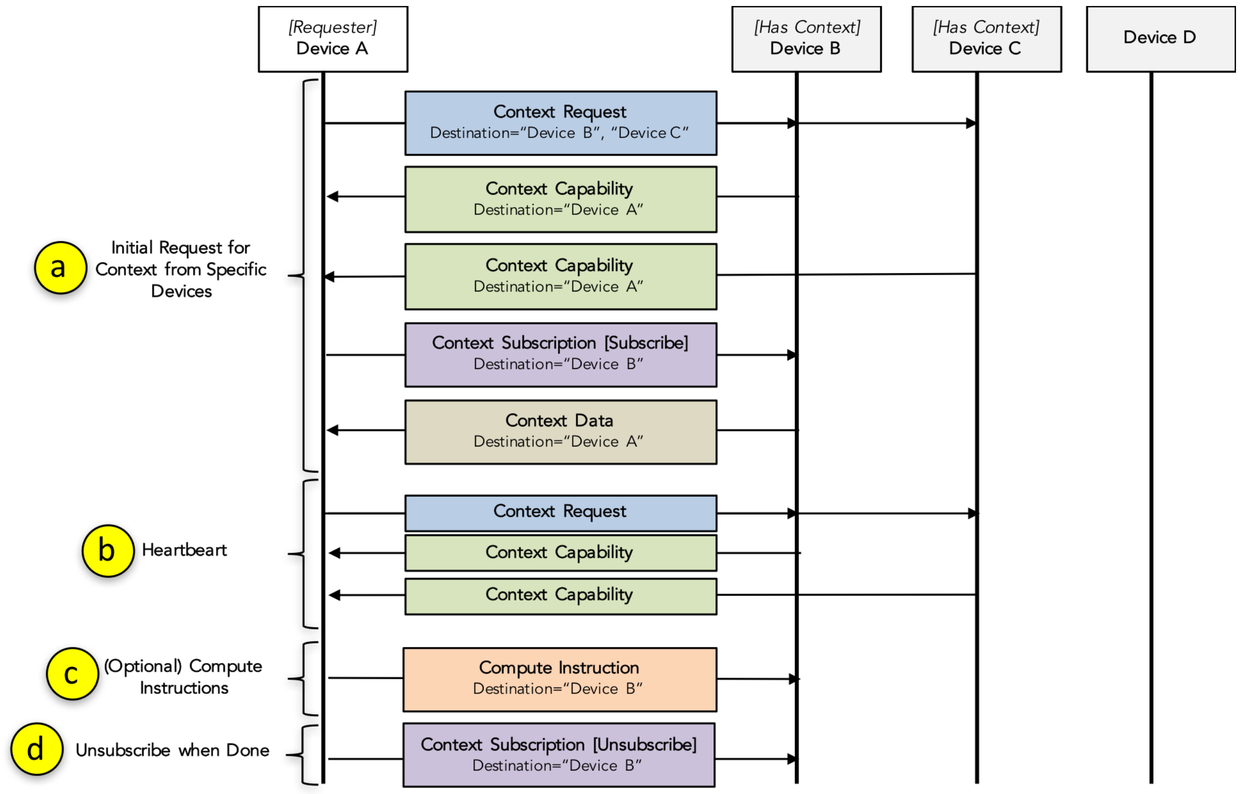


Figure 67. Sample communications between multiple GCF-enabled devices, showing the process by which devices request and receive context (a,b,d), and send asynchronous commands (c). By utilizing the destination field, Device A can direct messages to specific devices for additional customization options. Note that time is flowing down.

Figure 67 shows how these additional features affect GCF's abilities to find, form, and work in groups at runtime. Rather than have to request context from every device, Device A can now use the destination field to send a request to *just* Devices B and C (Figure 67a). These devices can then use the same mechanism to transmit an advertisement that only Device A will receive. Additionally, once Device A forms a group, it can use the group context manager's SENDCOMPUTEINSTRUCTION() method to transmit one or more remote commands to Device B (Figure 67c). These messages are then forwarded to the context providers ONCOMPUTEINSTRUCTIONRECEIVED() method, where they can be converted into any number of pre-canned actions.

Thus, while the changes to GCF's communications protocol seem simplistic, they augment the framework's capabilities in subtle but important ways. Whereas previous versions of GCF could only broadcast context requests and data, our updated framework now gives applications a great deal of flexibility in deciding which devices they want to interact with and when. Moreover, through the inclusion of the Compute Instruction message type, our framework can now be used to facilitate a wide range of real-time interactions. For backwards compatibility reasons, our framework can still be used to openly collect context over any network or subnet. By offering these additional capabilities, however, we increase the range of use cases that the framework can easily support.

5.1.2. *Group Context Manager Changes*

We have also made a number of modifications to GCF's group context manager in order to improve the framework's ability to find and form opportunistic groups:

Supporting User-Defined Groups. As observed in our work with Snap-To-It and Impromptu, there are times when users need to be able to explicitly specify which device(s) and/or service(s) they would like to form an opportunistic group with. In response to this, we have added a new MANUALARBITER class to GCF. Instead of forming groups according to a predefined policy (as is the case with GCF's Single Source, Multi-Source, and Local Only arbiters), manual arbiters collect context capability messages for a given context request. Applications can then use this information to present users with a list of options, and tell the arbiter which device(s) to group with. Although they lack any intelligence, manual arbiters make it easy for developers to include user-defined grouping in their applications. This addresses an important gap in GCF's functionality, and gives developers an additional way to use our framework.

Running GCF in the Background. Our work with Impromptu has also revealed the importance of being able to run GCF continuously. To address this, we have created a new wrapper class that allows the group context manager to run as a service on Android devices. To take advantage of this capability, developers call Android's STARTSERVICE() method in their application, and tell it to create an instance of the GCFSERVICE class. The service can then be used request, receive, and provide context to other devices, even when the user is not running the app on her mobile device, or looking at her screen. Additionally, the service can also restart itself when it detects that it has been deactivated by the user or operating system. This ensures that GCF is always online and available.

Bluewave Integration. Through Bluewave, we discovered that the ability to share context using Bluetooth names is useful in a wide range of use cases and real-world environments. As a result, we have modified GCF so that each group context manager has its own Bluewave client service and context broker. When developers use our framework, they can choose to either share context using a context provider, or to broadcast it *via* Bluewave to all nearby devices. Similarly, to obtain context, applications can either transmit a context request message across a network, or by conducting a Bluetooth scan. By offering multiple ways to request and receive context, GCF can now find and form groups in a wider range of environments. This allows devices to share information without requiring devices to always be connected to the same network or server.

Enabling *Ad Hoc* Communications. Another insight gained from our work with Bluewave is that we can use the technique to share network connection details (*e.g.,* IP Addresses, ports, communication protocols). This lets devices form temporary connections and share real-time context, even when they are on logically separate networks. In light of this discovery, we have modified the group context manager so that this capability is now provided "for free." Our updated group context manager now uses Bluewave to let devices advertise 1) what context providers they have installed, and 2) what channel they are listening on, to their immediate neighbors. Whenever a GCF application calls SENDCONTEXTREQUEST(), the group context manager uses Bluewave to determine if there are any nearby devices that can provide the requested information. If so, the group context manager will then form an *ad hoc* connection with the device, and subscribe to its context provider. When devices move out of Bluetooth range, the group context manager automatically detect that the devices are no longer nearby, and terminate the connection. By utilizing Bluewave in this manner, GCF is able to share real-time context data in a wider range of environments. This increases the types of applications that can be easily built, without creating additional complexity for developers.

5.1.3. *MISCELLANEOUS CHANGES*

Our final set of modifications to GCF consist of "quality of life" improvements. These modifications do not alter the framework's behavior, but rather provide additional backend support so that both users and developers can more easily make use of its features.

User Permissions. Inspired by our work with Bluewave, we have modified GCF to provide users with a more complete set of privacy controls. Our updated permissions screen provides users with a single, consolidated interface to see how their context is being collected and shared. From there, users can see which context providers are currently running, and what context is being broadcasted *via* Bluewave. They can then decide whether to share this information with all devices, or with specific applications. In addition to our permissions screen, GCF now notifies users when a new application is requesting context from them for the first time, and allows them to get more information about the application before deciding whether or not to share. Furthermore, the system also displays an icon on their phone whenever their device is providing context for another application. Collectively, these features help users better understand when and how their context is being shared. This improves GCF's intelligibility, and lets users configure the system so that they only share context when they feel comfortable doing so.

Live Dashboard. Finally, we have created an online dashboard to help developers get familiarized with our framework. Building on our work in Bluewave, this dashboard lets developers see 1) which context providers are commonly registered by applications, and 2) what information is commonly available. Developers can then use this information to get a better sense of which contexts are available "in the wild," which in turn allows them to decide which ones they want to request and receive in their applications. One of the key features of this dashboard is that its contents are dynamically populated by devices running our framework. This ensures that developers always have an up-to-date snapshot of how GCF is being utilized, which in turn increases their ability to take advantage of opportunistic groups in their applications.

5.2. Reevaluating the Framework

In **CHAPTER 3**, we evaluated GCF to see how the framework's functionality compared to other context-sharing systems and toolkits. Our analysis (summarized in Table 5) showed that GCF improved upon the state of the art by supporting all four group types. At the same time, our analysis also identified three technical limitations that prevented GCF from supporting the widest possible range of use cases. Specifically, early versions of our framework suffered from:

1. Inability to (easily) form groups based on physical distance
2. Lack of support for user-defined groups
3. Inability to communicate with devices unless they are on the same broadcast channel.

In this section, we reevaluate our updated framework (using the same criteria we used in chapters 2 and 3) to illustrate how our modifications have improved its functionality. The results of this analysis are presented below:

Table 15. Analysis of GCF's functionality. Differences between this version of GCF and the one presented in Table 5 are highlighted.

	Supports Multiple Apps	Supports Cross-App Context Sharing	Static	Dynamic	User Specified	Long	Short	Repeated	One Time	Sensors (Real-Time)	Software (Interpreted Context)	User Input	Static	Ad Hoc	Collaborative (Same Task/Data)	Cooperative (Same Task, Dif Data)	Coincidental (Dif Task, Same Data)	In-Situ (Dif Task/Data)
System Name	Reusability		Group Discovery			Group Range		Group Longevity		Supported Contexts			Communications		Group Types			
Group Context Framework (Updated)	x	x	x	x	o	x	x	x	x	x	x	x	x	x	x	x	x	x

X = Fully Supports, O = Partially Supports

As illustrated in Table 15, our revised framework (Figure 68) addresses and/or mitigates each of the problems identified in **CHAPTER 3:**

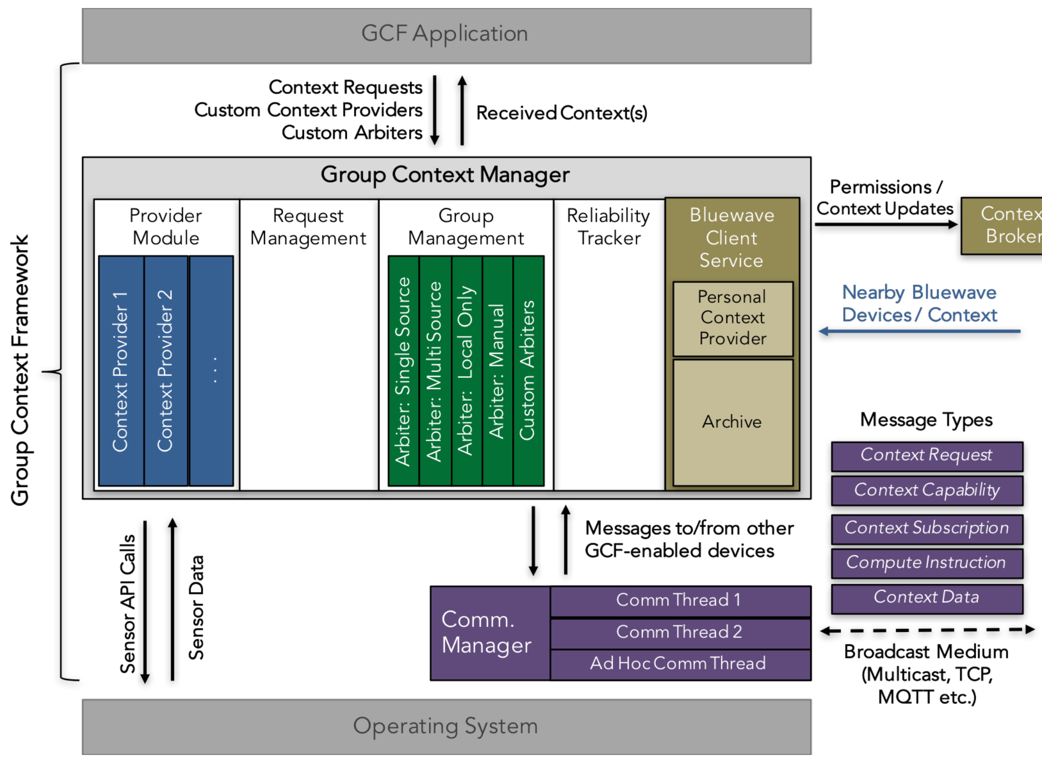


Figure 68. GCF's final architecture, which incorporates all of the lessons learned in CHAPTER 4.

- To address limitation #1, GCF now comes with mechanisms to form groups over both short and long distances. In addition to being able to broadcast messages across local area networks, GCF can also utilize Bluewave to detect nearby GCF-enabled devices (Figure 68, brown). This lets the framework form groups with devices when they are within short range of each other. Meanwhile, by using a well-known communications broker (*i.e.,* a server), GCF can also allow devices to communicate over any arbitrary distance. By supporting multiple communications technologies (*e.g.,* Bluetooth, local area networks, servers), GCF allows developers to decide which technologies work best for their particular application. This lets them form groups across a wide range of physical distances.
- To address limitation #2, GCF now provides more explicit support for user-defined groups. As shown in Figure 68 (green), we have added a new MANUALARBITER class to GCF that lets applications collect context capability messages from other devices, and programmatically specify which devices they want to group with. This solution still requires developers to provide their own selection interfaces. Nevertheless, it provides developers with a highly customizable way to add manual grouping to their applications.
- To address limitation #3, we have given GCF the ability to form *ad hoc* communications on the user's or application's behalf. By sharing network connection details (*e.g.,* IP Addresses, ports) *via* Bluewave, our system is automatically able to detect nearby devices that can provide context, and form a temporary connection with them. This allows GCF to form groups with devices in almost any Internet connected environment.

Our analysis shows that our revised framework significantly improves upon our initial release. Clearly, there may be additional features that might need to be added to GCF in future releases. Nevertheless, by incorporating the lessons learned from the previous chapter into the framework, we are able to address all of the problems we have identified in other systems thus far. In doing so, we provide a better answer to our first research question.

5.3. A Design Process for Developing Applications that Use Opportunistic Groups

How can developers effectively utilize GCF in their applications? Until now, our work has focused on showing how GCF *can* be used to support a broad range of applications. Yet while one of our main goals with GCF is to make it easy to use, we have yet to provide developers with explicit guidance as to how the framework *should* be incorporated into new or existing software systems. This is important, as developers will not have the same knowledge of GCF's inner workings as we do, and thus may not intuitively know how to best make use of its features.

One way to help developers utilize a new framework is to provide them with high level instructions to help guide their thinking. In [31], Dey identified a design process for building context-aware applications, and showed how this process could guide developers' during the planning and implementation stages. His process consisted of the following steps:

1. **Specification:** Specify the problem being addressed and a high-level solution
   1.1. Specify the context-aware behavior(s) to implement
   1.2. Determine what context is required for those behavior (with a knowledge of what is available from the environment) and request it.
2. **Acquisition:** Determine what hardware or sensors are available to provide that context and install them.
3. **Action:** Choose and perform context-aware behavior.

In this section, we introduce a generalizable design process for developing context-aware applications using GCF. Our process is inspired by [31], and includes the same high level steps (specification, acquisition, and action). However, our experiences from **CHAPTER 4** have revealed that there are several additional considerations that developers need to take into account when utilizing opportunistic groups *via* our framework. As a result, we have added several sub-steps to this process to bring these considerations to light.

Our modified design process consists of the following steps (modifications to Dey's process are underlined):

1. **Specification:** Specify the problem being addressed and a high-level solution
   1.1. Specify the context-aware behavior(s) to implement
   1.2. Determine what context is required for those behavior, <u>and how often it is needed</u>
2. **Acquisition:** Determine which <u>device(s)</u> can provide the needed context
   2.1. <u>Determine how each context should be collected and shared</u>
   2.2. <u>Define group membership criteria</u>
      2.2.1. <u>Size (*i.e.,* "How many devices does the application need to obtain context from?")</u>
      2.2.2. <u>Range (*i.e.,* "How far apart can devices be from each other before they are no longer considered members of the same group?")</u>
   2.3. <u>Request and/or listen for context</u>
3. **Action:** Choose and perform context-aware behavior.

In the following sections, we present three case studies that show how our design process can be put into practice. In the first case study, we create an enhanced version of GroupMap (Figure 13) that can share location data in arbitrary environments (as opposed to just environments with a local area network). In the second case study, we create an opportunistic sensing platform that allows us to task users' smartphones to perform arbitrary sensing tasks over a large geographic area. Finally, in the third case study, we develop a peer-to-peer collaboration application that lets users share clipboard data (*e.g.,* text, links, photographs/files) with both personally owned, and nearby devices. For each case study, we describe the application's desired functionality. We then step through each phase of our design process, and show how the results of this analysis are converted into working code.

5.3.1. *CASE STUDY #1: GROUPMAP 2.0*

Our first case study revisits the GroupMap application that we introduced in **CHAPTER 3**. As mentioned earlier, GroupMap is a mobile app that lets users view their location on their phones. In contrast with existing map applications that only use a single GPS sensor (*i.e.,* the sensor on the device running the application), GroupMap is able to request and receive location data from other GroupMap devices *via* GCF. This not only lets devices take turns running their GPS (conserving battery life), but also allows users to track each other's locations when they are nearby.

GroupMap is inspired by the our first motivational scenario (**CHAPTER 1**), in which we described a situation where a mobile phone is able to obtain GPS data from a bus in order to track the user's current location. When we initially created GroupMap in section 3.5.1, we used an early version of GCF that could only communicate with devices on the same local area network. This prevented the application from forming opportunistic groups in non-networked environments. In this section, we use our design process to create an enhanced version of GroupMap that can detect nearby devices and form *ad hoc* network connections *via* GCF. This lets it share location data in a wider range of real-world environments, and demonstrates how our framework addresses our inspirational scenarios.

5.3.1.1. Using the Design Process

In this section, we show how our design process was used to create GroupMap. We highlight the major considerations that were made at each step, and show how the insights obtained from this process informed our final implementation.

Specification

The first step in our design process is to determine what context-aware behavior(s) we want the application to implement (step 1.1). As mentioned above, our goal with GroupMap is to let devices share location data with each other so that they can either conserve battery life, or let users see who is nearby. As a result, the behavior we want to provide is a map that either shows users 1) their current location (when in navigation mode), or 2) the locations of all nearby users (when in tracking mode).

Next, we need to determine which context(s) are needed to support this behavior, and how often they need to be collected (step 1.2). The first half of this task is straightforward, as the only context that needs to be collect and shared is the user's location. The second half, however, is more subjective. Since GroupMap is a navigation tool, we can assume that the application will need to receive regular location updates (*e.g.,* once per second) order to display the correct icon(s) on the map. Furthermore, since GroupMap's output is a map, we can also assume that users will only need to collect context when the app is in the foreground. This means that we can run GCF directly within the application rather than create an "always on" background service.

Acquisition

In the *acquisition* step, we determine how we can obtain the desired context using GCF. From the previous step, we know that GroupMap needs to be able to collect location from other devices if and when they are nearby. At the same time, the application should also be able to default to the local device's GPS sensor in the event that no other devices are available. Given these requirements, we include a LocationProvider in GroupMap. This ensures that the context is accessible from both the user's device, as well as from all other devices (step 2.1). Additionally, it also allows GroupMap to obtain context from any device that has a LocationProvider, regardless if they have installed our GroupMap application. This increases GroupMap's chances of finding a suitable source of context when used in a real-world setting.

The next step is to determine which device(s) GroupMap should form a group with (step 2.2). The answer depends on which mode the user has selected. In navigation mode, for example, GroupMap should only group with a single device (*i.e.* a device that has a LocationProvider, and a higher remaining battery life). Meanwhile, when the application is in tracking mode, it should form a group with every device so that it can display all of their locations on the user's map. In both modes, the application should only form a group with devices that are nearby. This means that the application will need to periodically scan its surroundings and only group with those devices that it detects.

The last step is to actually request and receive the context (step 2.3). Using the above discussion as a guide, we create two context requests. When the application is in navigation mode, we use GCF to transmit a *single source* request for location data. When the application is in tracking mode, we issue a *multi-source* request instead. To ensure that the application will only form a group with nearby devices, we direct GCF to conduct a Bluewave scan every 60 seconds. The framework will automatically identify nearby devices that are willing to provide the context, form an *ad hoc* network connection, and request and receive information from them. It will then deliver this context back to the application through an Android Intent (*ACTION_GCF_DATA_RECEIVED)*.

Action
The final step in the design process is to perform the context-aware behavior (step 3). When GroupMap receives location data, it checks to see what mode it is currently in. If the application is in navigation mode, the app needs to place a single marker on the map to indicate the user's approximate location. Otherwise, the app will display multiple markers on the map (*i.e.,* one per each device), and will color code each marker so that the user can tell where he/she is in relation to the other devices.

5.3.1.2. Implementation

In Figure 69, we show how we implemented GroupMap in Android. When the application is first started, it calls Android's ONCREATE() method. When this occurs, we create the GroupContextManager (line 12), register the LocationProvider (line 15), and tell Bluewave to start scanning for nearby devices every 60 seconds (line 18). The application then waits for the user to select which mode he/she wants to use (by pressing a button), and transmits either a single source or multi-source request (lines 26 and 28, respectively). Each time GCF receives location data, it broadcasts an Android Intent that is captured by the ONRECEIVE() method. This method forwards the context to the UPDATEMAP() (line 45), which users it to modify the map.

Not counting the code required to create or update the user interface, adding GCF to GroupMap took less than 50 lines of code. This demonstrates the simplicity of our framework, and shows how easy it is for developers to incorporate it into their applications.

```java
// This code runs when the application is first started
public void onCreate() {
 super.onCreate();

 // Create Intent Filter and Receiver
 this.intentReceiver = new ApplicationIntentReceiver();
 this.filter = new IntentFilter();
 this.filter.addAction(AndroidGroupContextManager.ACTION_GCF_DATA_RECEIVED);
 this.registerReceiver(intentReceiver, filter);

 // Creates the Group Context Manager
 this.gcm = new AndroidGroupContextManager(this, "Device A", false);

 // Initialize Context Providers
 gcm.registerContextProvider(new LocationContextProvider(this, gcm));

 // Tell Bluewave to Start Scanning
 gcm.getBluewaveManager().startScan(60000);
}

// Requests Context Based on the Current Mode
public void setMode(int mode) {
 gcm.cancelRequest("LOC");

 if (mode == NAVIGATION_MODE) {
  gcm.sendRequest("LOC", ContextRequest.SINGLE_SOURCE, new String[0], 30000, new String[0]);
 }
 else if (mode == TRACKING_MODE) {
  gcm.sendRequest("LOC", ContextRequest.MULTIPLE_SOURCE, new String[0], 30000, new String[0]);
 }
}

// This intent receiver listens for context received by GCF
public class ApplicationIntentReceiver extends BroadcastReceiver {
 @Override
 public void onReceive(Context context, Intent intent) {
  if (intent.getAction().equals(AndroidGroupContextManager.ACTION_GCF_DATA_RECEIVED)) {
   // Extracts the values from the intent
   String  contextType = intent.getStringExtra(ContextData.CONTEXT_TYPE);
   String  deviceID  = intent.getStringExtra(ContextData.DEVICE_ID);
   String[] payload   = intent.getStringArrayExtra(ContextData.PAYLOAD);

   // Updates the Map
   ContextData data = new ContextData(contextType, deviceID, payload);
   updateMap(data);
  }
 }
}
```

Figure 69. Implementation of the GroupMap Application

5.3.2. *CASE STUDY #2: IMPROMPTU SENSE*

For our second case study, we created Impromptu Sense, a mobile sensing platform that lets researchers opportunistically task users' smartphones to perform remote sensing tasks. The Impromptu Sense system consists of 1) a mobile app that runs continuously on the user's phone, and contains context providers for each of the phone's common sensors (*e.g.,* audio magnitude, GPS, Bluetooth), and 2) a desktop application that lets other users specify the specific sensor streams that they want to request and receive. When a user requests context *via* our desktop application, GCF transmits a ContextRequest message to all users' smartphones. The application then forms a group with the smartphone(s) that are willing to provide the context, and either stores the data (*e.g.,* sensor readings) in a file for future analysis, or produces a visualization.

Impromptu Sense has been deployed and used outside of the lab. In collaboration with the Huntington County Emergency Management Agency (EMA), we installed Impromptu Share on 28 police, firefighter, and medical personnel smartphones during the CreationFest 2015 Music Festival. We then conducted a series of field trials to demonstrate how the ability to opportunistically request and receive context could be used to improve first responders' situational awareness. Some of the sensing tasks that we performed during these trials included:

1. Collecting location data from each first responder in order to help EMA leadership identify areas of the festival grounds that are infrequently patrolled
2. Directing users' phones to perform periodic Bluetooth discovery scans in order to estimate crowd density, and search for specific individuals (*e.g.,* a lost child)
3. Analyzing audio amplitude data from smartphone microphones in order to identify and triangulate noisy events (*e.g.,* fireworks, loud music)

In the following section, we discuss how we created Impromptu Share using GCF. Our framework provides all of the basic components needed to support remote sensing tasks. This makes it easy for developers to collect and analyze large amounts of context without having to generate a large amount of custom code.

5.3.2.1. Using the Design Process

The process used to create Impromptu Share is identical to the one we used in GroupMap. Consequently, rather than describe each step again, we instead highlight the important design decisions that were made during the specification, acquisition, and action phases.

Specification

Step 1.1: Specify the context-aware behaviors to implement. The goal of Impromptu Share is to let researchers opportunistically task users' mobile phones to collect and deliver context through our desktop application. As a result, the desired behavior is to save all incoming context so that it can be analyzed and/or used to produce a visualization.

Step 1.2: Determine what context is required for those behavior, and how often it is needed. The context needed can significantly vary depending upon the sensing task being performed. Some tasks, for example, may only require devices to report their current location, while others may need to know the user's location, as well as their compass heading and activity. For now, we have chosen six contexts that are 1) commonly used in context-aware applications, and 2) easily collected and/or sensed on most smartphones/tablets:

- Location
- Activity
- Accelerometer
- Compass Heading
- Audio Magnitude
- Bluetooth

In addition to knowing which contexts are required, is also important to consider *when* this information is needed. Since the desktop application can request context at any time, it is important that each smartphone is always listening for a ContextRequest message, even if the user is not running the Impromptu Share app. This means that the mobile app will need to run GCF as a background service.

Acquisition

Step 2.1: Determine how each context should be collected and shared. The contexts described in step 1.2 can be obtained *via* the following GCF context providers:

- LocationProvider
- ActivityProvider
- AccelerometerProvider
- CompassProvider
- AudioAmplitudeProvider
- BluetoothProvider

These context providers are not needed on every device. Since the desktop application is only responsible for requesting, receiving, and storing context, it does not need any context providers of its own. The mobile app, on the other hand, *is* responsible for producing and sharing context. Consequently, all of the context providers listed above need to be registered with the mobile app.

Step 2.2: Define group membership criteria. Since the purpose of Impromptu Sense is to opportunistically collect data, the desktop application needs to be able to form a group with any device that is willing to provide the requested information. Depending upon the request, the resulting group may only consist of one device, or with every device that has our mobile app.

Additionally, we assume that the smartphones running our mobile app are distributed across a large geographic area, and will thus not always be on the same local area network or detectable *via* Bluetooth. This means that we need to set up a centralized communications channel (*e.g.,* an MQTT broker) so that the desktop application can request and receive context from devices regardless of where they are located.

Step 2.3: Request and/or listen for context. Each time a user requests context, our desktop app will transmit a multi-source request. This will ensure that the application forms as large a group as possible.

Action

Step 3: Choose and perform context-aware behavior. As Impromptu Sense is a data collection tool, the application only needs to save incoming context into a flat file for future analysis.

5.3.2.2. Implementation

Figure 70 shows how we implemented Impromptu Sense's desktop application in Java. When the application is started, it creates a new group context manager (line 5) and connects to our MQTT message broker (lines 11-12). The application then waits for the user to specify which context(s) he/she would like to collect, and uses this information

```
1.  public class ImpromptuShareDesktopApplication implements OnContextDataReceiver
2.  {
3.   public ImpromptuShareDesktopApplication() {
4.    // Creates the Group Context Manager
5.    GroupContextManager gcm = new DesktopGroupContextManager("Desktop A", false);
6.
7.    // Registers this object to receive on context data events
8.    gcm.registerEventReceiver(this);
9.
10.   // Connects to the Impromptu Share Channel
11.   String connectionKey = gcm.connect(CommMode.MQTT, "epiwork.hcii.cs.cmu.edu", 1833);
12.   gcm.subscribe(connectionKey, "IMPROMPTU_SHARE");
13.
14.   // Requests Context
15.   gcm.sendRequest("LOC", MULTI_SOURCE, new String[0], 15000, new String[0]);
16.  }
17.
18.  public void onContextData(ContextData data) {
19.   doSomething(data);
20.  }
21. }
22.
23. public static void main(String[] args) {
24.  new ImpromptuShareDesktopApplication();
25. }
```

Figure 70. Implementation of Impromptu Sense's Desktop App

to transmit one or more multi-source context requests (line 15). As GCF begins receiving data from other mobile devices, it is automatically forwards the ContextData messages to the desktop's onContextDataReceived() method (line 18). The data is then logged in a file, and/or sent to the application to generate a visualization.

Meanwhile, Figure 71 shows how we implemented our Android app. When the app is started, it starts the GCF service (line 14), and waits until Android broadcasts an Intent stating that the service has been created. When this event occurs, the app registers all of the context providers with the service's group context manager (lines 30-35). It then connects to the same MQTT broker as the desktop application (line 12), and waits for ContextRequest messages.

As was the case with GroupMap, the amount of custom code needed to create Impromptu Share using GCF was minimal. Not counting the code required for the user interfaces, we were able to implement both the Android app and desktop application in less than 60 lines of code, *combined*. This provides further evidence that our framework contains the abstractions needed to form groups and share context in many situations.

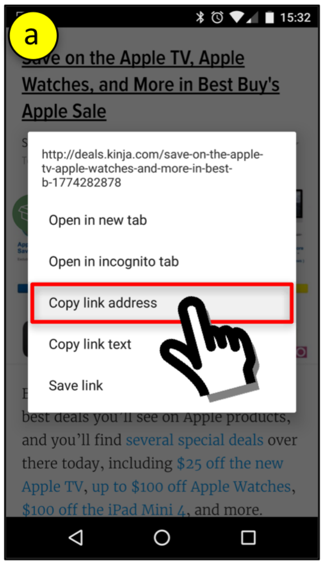
```java
// This code runs when the application is first started
public void onCreate() {
 super.onCreate();

 // Create Intent Filter and Receiver
 this.intentReceiver = new ApplicationIntentReceiver();
 this.filter = new IntentFilter();
 this.filter.addAction(GCFService.ACTION_GCF_STARTED);
 this.registerReceiver(intentReceiver, filter);

 // Creates Intent to Start the Service
 Intent i = new Intent(this, GCFService.class);
 this.bindService(i, gcfServiceConnection, BIND_AUTO_CREATE);
 this.startService(i);
}

// This intent receiver listens for context received by GCF
public class ApplicationIntentReceiver extends BroadcastReceiver {
 @Override
 public void onReceive(Context context, Intent intent) {
  if (intent.getAction().equals(GCFService.ACTION_GCF_STARTED)) {
   // Gets the Group Context Manager from the Service
   GroupContextManager gcm = gcfService.getGroupContextManager();

   // Connects to the Impromptu Share Channel
   String connKey = gcm.connect(CommMode.MQTT, "epiwork.hcii.cs.cmu.edu", 1833);
   gcm.subscribe(connKey, "IMPROMPTU_SHARE");

   // Registers Context Providers
   gcm.registerContextProvider(new LocationContextProvider(this, gcm));
   gcm.registerContextProvider(new AccelerometerContextProvider(this, gcm));
   gcm.registerContextProvider(new ActivityContextProvider(this, gcm));
   gcm.registerContextProvider(new CompassContextProvider(this, gcm));
   gcm.registerContextProvider(new AudioContextProvider(this, gcm));
   gcm.registerContextProvider(new BluetoothContextProvider(this, gcm));
  }
 }
}
```

Figure 71. Impromptu Sense's Android Implementation.

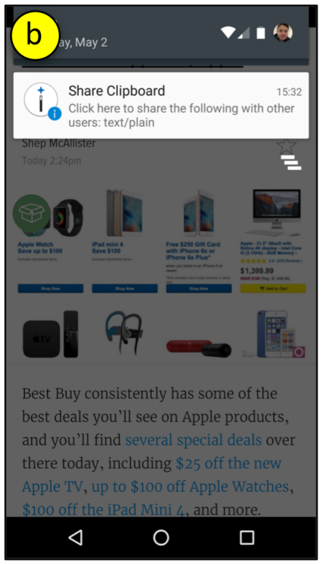
5.3.3. *Case Study #3: Impromptu Share*

For our third case study, we created Impromptu Share, an opportunistic collaboration tool that lets users serendipitously transfer content (*e.g.,* text, links, photographs, files) from one device to another. Impromptu Share continually monitors the user's clipboard, and detects when she has copied content to it (Figure 72a-b). The application then 1) automatically shares this content with the user's own devices so that they can update their clipboards, and 2) generates an optional interface to let users share this content with their neighbors (Figure 72c). If the user decides to share, Impromptu Share forms an opportunistic group with the recipients' devices, and transmits a message containing the clipboard data. The receiving devices (also running Impromptu Share in the background) can then either update their clipboards, or present the information to the user *via* a system notification (Figure 72d-e).

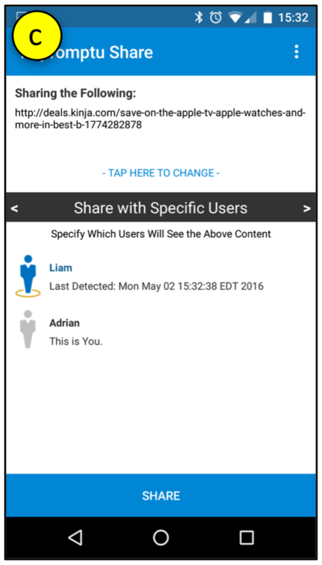
From a software implementation standpoint, Impromptu Share utilizes GCF's full range of capabilities. Our application manages multiple groups at runtime in order to let users easily share context with both their own devices (known *a*


a
15:32
Save on the Apple TV, Apple Watches, and More in Best Buy's Apple Sale
http://deals.kinja.com/save-on-the-apple-tv-apple-watches-and-more-in-best-b-1774282878
Open in new tab
Open in incognito tab
Copy link address
Copy link text
Save link

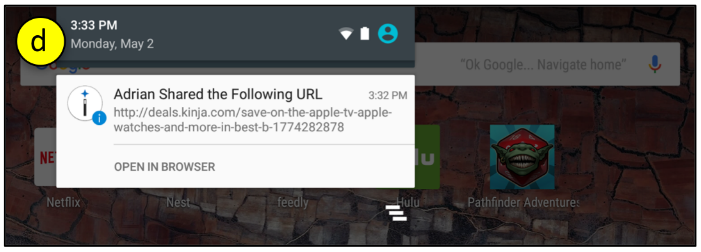
A user copies a web link on his phone.


b
Share Clipboard
15:32
Click here to share the following with other users: text/plain

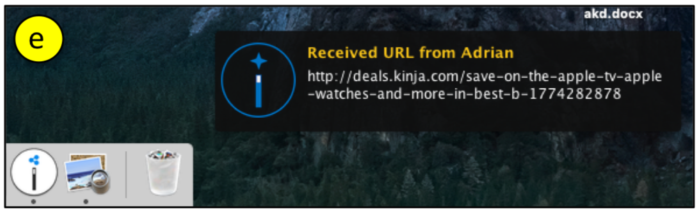
Impromptu Share detects that the clipboard has been modified, and asks the user if he wants to share.


c
15:32
Impromptu Share
Sharing the Following:
http://deals.kinja.com/save-on-the-apple-tv-apple-watches-and-more-in-best-b-1774282878
- TAP HERE TO CHANGE -
Share with Specific Users
Specify Which Users Will See the Above Content
Liam
Last Detected: Mon May 02 15:32:38 EDT 2016
Adrian
This is You.
SHARE

The user gets a list of nearby users, and selects the one(s) he wants to share with.


d
3:33 PM
Monday, May 2
Adrian Shared the Following URL
3:32 PM
http://deals.kinja.com/save-on-the-apple-tv-apple-watches-and-more-in-best-b-1774282878
OPEN IN BROWSER

The recipient receives a notification on his Android tablet . . .


e
akd.docx
Received URL from Adrian
http://deals.kinja.com/save-on-the-apple-tv-apple-watches-and-more-in-best-b-1774282878

. . . as well as a popup notification on his laptop computer.

Figure 72. An overview of the Impromptu Share application. When a user copies content to the clipboard (a), Impromptu Share asks him if he wants to share it (b), and provides an interface to select one or more nearby users (c). The recipients then see the shared content on their devices (d-e).

priori), as well as with those that just happen to be nearby. Furthermore, Impromptu Share broadcasts the user's identity and network connection settings *via* Bluewave, and lets devices from the same owner conduct Bluetooth scans on each other's behalf so that they can collectively know who is nearby. While each of these capabilities has been demonstrated before, Impromptu Share integrates all of them into a single application. In doing so, we show how these features can work together to create more sophisticated context-aware applications.

5.3.3.1. Using the Design Process

We now walk through each step of our design process to show how we created Impromptu Share.

Specification

Step 1.1: Specify the context-aware behaviors to implement. Impromptu Share lets users quickly and serendipitously share clipboard data with both their own devices as well as with their neighbors. In support of this goal, we have identified the following context-aware behaviors:

- *Let users automatically share clipboard content with their own devices.* Impromptu Share will let users create a shared clipboard with their own devices. This will let users copy content on one device (*e.g.,* a laptop), and immediately paste it on another (*e.g.,* a phone).
- *Allow users to easily share clipboard content with one or more nearby users.* Impromptu Share also needs to let users opportunistically push clipboard data to each other. Each time the user copies data to the clipboard, the system will generate a list of nearby users (using Bluetooth proximity). The user will then be able to select the specific user(s) she would like to share with, and the application will automatically group with these devices and push content to them.

Step 1.2: Determine what context is required for those behavior, and how often it is needed. There are four types of context that are needed to support the above behaviors. The first is the *contents of the user's clipboard*. Each time the user performs a copy operation, our system needs to transmit the current contents of the clipboard (*e.g.,* text, files, images) to the rest of the user's devices. These devices can then modify their own clipboards so that they remain synchronized.

The second type of context that needs to be shared is the *user's identity*. Rather than list devices by their system generated name (*e.g., "*f13phd1.hcii.cs.cmu.edu"), our system lets devices share their owner's name (*e.g.,* "Adrian"). Impromptu Share uses this information to generate a list of nearby people, providing users with a more natural way to specify who they want to share with.

The third type of context that needs to be openly shared is *the communications channel where Impromptu Share is listening for clipboard data*. This information is needed so that users can *push* clipboard data to each another without having to rely on a centralized server.

The last type of context that devices need to share is the *context collected via Bluewave*. Since Impromptu Share is intended to run on both mobile devices *and* desktops, there needs to be a way for non-Bluetooth equipped devices to detect nearby users. To address this, our system will let Bluetooth equipped devices scan for nearby devices on their behalf. This gives older devices a way to obtain context *via* Bluewave, and allows our system to offer the same capabilities across different platforms.

The above list contains a mix of static and dynamic data. For example, the user's identity and communications channel are not expected to change during the lifetime of the application. However, in order to monitor the user's clipboard and collect Bluewave context, GCF needs to be running continually. As a result, we have designed Impromptu Share to run as a background service.

Acquisition

Step 2.1: Determine how each context should be collected and shared. In order to collect and share the contexts listed above, we use the following techniques:

- To collect and share the user's clipboard contents, we need to create a custom context provider (CLIPBOARDPROVIDER, context type = "CLIP") that continuously monitors a device's clipboard, and transmits a CONTEXTDATA message each time its contents change. This context provider can also process

COMPUTEINSTRUCTION messages containing clipboard data so that other users can push content to the clipboard.

- To share the user's identity and communications channel, we use Bluewave. This allows nearby device to 1) know who is nearby, and 2) what channel they are listening on, without requiring the devices to be connected to the same network. This allows Impromptu Share to work in most real-world environments.
- Finally, to let user owned-devices share Bluewave context, we need to create a custom context provider (BLUEWAVEPROVIDER, context type = "BW") that can periodically scan for nearby devices, request their context *via* Bluewave, and deliver the information to all subscribers.

The first context needs to be collected and shared on both the mobile and desktop versions of Impromptu Share. The latter two, on the other hand, require Bluetooth, and as such are only collected and shared by our mobile app.

Step 2.2: Define group membership criteria. Impromptu Share forms two different groups at runtime. The first group consists of the user's own devices, which are assumed to be known *a priori*. This group lasts for the lifetime of the application, and is used to share both clipboard data, as well as context obtained through Bluewave (*e.g.,* user identity and network connection) settings. The second group consists of the devices that the user would like to share clipboard content with. These groups are manually formed by the user through our app, and only last for as long as it takes for the sending device to push clipboard data to the intended recipients.

The physical range of these groups varies. The first group should be able to transmit clipboard data regardless of where they are physically located. This means that this group should be able to communicate over a dedicated communications channel (e.g., the same channel on an MQTT broker). The second group, in contrast, only needs to share clipboard data when they are within Bluetooth range of each other. As a result, these devices only need to communicate using GCF's *ad hoc* communications capability.

Step 2.3: Request and/or listen for context. In order to collect the contexts identified above, Impromptu Share performs the following actions:

- To obtain clipboard data, Impromptu Share transmits a multi-source ContextRequest message. This lets it subscribe to the ClipboardProviders on the user's other devices and receive real-time updates.
- When the user shares clipboard data *via* the *mobile app*, the application begins scanning for nearby devices using Bluewave. This lets it determine which users are nearby, and which channel(s) they are listening for clipboard data on.
- When the user indicates that she wants to share content with other users *via* the *desktop application*, the application transmits a multi-source request for Bluewave. This allows other devices (owned by the user) to collect and deliver this information on its behalf.

Action

Step 3: Choose and perform context-aware behavior. There are four context-aware behaviors that Impromptu Share performs at runtime:

1. *Transmit Updated Clipboard Contents.* Each time the user copies content to the clipboard, the ClipboardProvider transmits a new ContextData message containing the updated clipboard contents to the user's other devices. The application then asks the user (*via* a notification) if she would like to share this data with her immediate neighbors.
2. *Maintain a List of Nearby Users.* Our application uses the information collected *via* Bluewave in order to determine which users are nearby. This information is used to create a custom interface so that users can select the names of the specific individuals they would like to share with.

3. *Share Clipboard Data.* When the user chooses to share clipboard data, Impromptu Share will connect to the recipient(s) communication channel and transmit a ComputeInstruction containing the clipboard contents. Once complete, the application will automatically disconnect.
4. *Process Shared Clipboard Data.* When the application receives a ContextData or ComputeInstruction message containing clipboard data, it checks to see what type of information is contained within. If the content is text, the application automatically updates the clipboard. However, if the content is a URL or file, the application displays a notification so that the user can decide whether or not to open the link or download the file, respectively.

5.3.3.2. Implementation

Impromptu Share's architecture is presented in Figure 73. Our system consists of:

- A **mobile application**, which contains both a ClipboardProvider and a BluewaveProvider, and runs as an Android service.
- A **desktop application**, which only contains a ClipboardProvider, and runs as a background thread.

To configure our system, users install the appropriate version of Impromptu Share on all of their devices, and provide each one with the same name and password. The applications then use this information to connect to two communications channels. The first channel is intended for private communications, and is only known to devices that are owned by the same user. Our apps use this channel to transmit multi-source requests for clipboard data, thereby allowing them to form a group and synchronize their clipboards (Figure 73a). Additionally, the desktop version of our app also uses this channel to transmit multi-source requests for Bluewave data. This lets the app obtain context from the user's Bluetooth-equipped devices so that it can determine who is nearby (Figure 73b).

The second communications channel is used to *push* clipboard data to a specific user. Whereas the first channel is intentionally kept secret from other users, the second channel's connection details (*e.g.,* IP address, channel) are publically advertised *via* Bluewave. When a user wants to share clipboard data, her mobile app uses Bluewave to determine which users are nearby and what channels their devices are listening on (Figure 73c). The app then temporarily connects to the recipients' public MQTT channels, and transmits a COMPUTEINSTRUCTION message

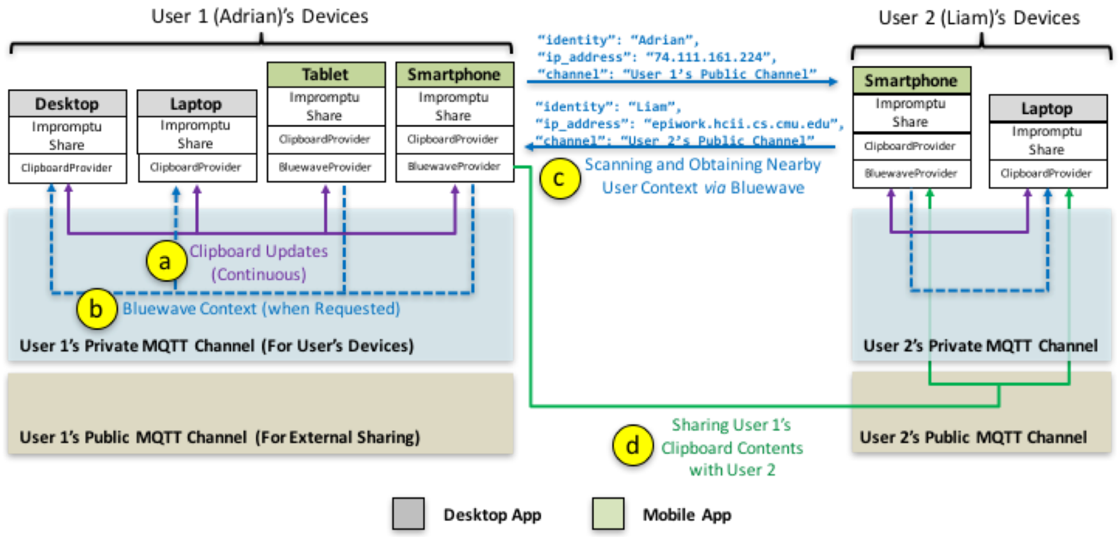

User 1 (Adrian)'s Devices
Desktop
Impromptu Share
ClipboardProvider
Laptop
Impromptu Share
ClipboardProvider
Tablet
Impromptu Share
ClipboardProvider
BluewaveProvider
Smartphone
Impromptu Share
ClipboardProvider
BluewaveProvider
"identity": "Adrian",
"ip_address": "74.111.161.224",
"channel": "User 1's Public Channel"
"identity": "Liam",
"ip_address": "epiwork.hcii.cs.cmu.edu",
"channel": "User 2's Public Channel"
User 2 (Liam)'s Devices
Smartphone
Impromptu Share
ClipboardProvider
BluewaveProvider
Laptop
Impromptu Share
ClipboardProvider
c
Scanning and Obtaining Nearby User Context *via* Bluewave
a
Clipboard Updates (Continuous)
b
Bluewave Context (when Requested)
User 1's Private MQTT Channel (For User's Devices)
User 2's Private MQTT Channel
User 1's Public MQTT Channel (For External Sharing)
User 2's Public MQTT Channel
d
Sharing User 1's Clipboard Contents with User 2
Desktop App
Mobile App

Figure 73. Impromptu Share Architecture

containing the clipboard contents (Figure 73d). The message is forwarded to all of the recipients' devices, where they are either used to update the clipboard or generate a system notification.

In comparison to the previous two case studies, Impromptu Share required significantly more work to implement. To create this system, we had to create custom context providers that could collect and share clipboard data and Bluewave scan results, respectively. Furthermore, to prevent users from being able to "sniff" each others clipboards with permission, we had to direct GCF to form multiple communications channels, and insert our own logic in order to tell the group context manager which channels to use when sharing and/or pushing clipboard data. Yet even with these additional requirements, we were able to create Impromptu Share in less than 300 lines of code. This shows how the framework can be customized for special use cases, while still making common grouping tasks (*e.g.,* requesting and receiving context) easier to implement.

5.4. SUMMARY

In this chapter, we looked at GCF through two different lenses. First, we examined GCF from an architectural standpoint, and highlighted the major changes that we have made to the framework based on our experiences using it in **CHAPTER 4**. We showed how our updated framework addresses all of the limitations found in previous context-sharing systems, thereby providing us with a finalized architecture that supports the finding and forming of opportunistic groups.

Afterwards, we looked at GCF from a developer standpoint, and presented a generalizable design process for creating context-aware applications using our framework. We showed how this process builds on prior work by highlighting the important considerations that developers should take into account when forming and using opportunistic groups in the specification, acquisition, and action phases. We then presented three case studies to show how this design process can put into practice. In each case, we found that the process helped us better understand which features of GCF we needed to utilize for a particular application. For example, step 1.2 ("Determine what context is required for those behavior, and how often it is needed") revealed whether or not we needed to run GCF as a background service or directly within the application. Step 2.1 ("Determine how each context should be collected and shared") was useful in helping us determine which contexts needed to be collected using context providers, and which contexts could be shared *via* Bluewave. Finally, step 2.2 ("Define group membership criteria") helped us better understand 1) what type of context request we should transmit (*e.g.,* single source, multi-source), and 2) whether or not we needed to set up a centralized communications channel to let devices communicate over arbitrary distances.

Through this work, we have taken an important step towards showing how GCF can actually be used. In the following chapter, we build on this idea by having developers see our framework and brainstorm ways that they would like to use it. In doing so, we validate the framework's core features, and show that it already supports the most common use cases.

6. Developer Validation

In this thesis, we have presented numerous examples that show how GCF can be used to form opportunistic groups and share context. Yet while our work in **CHAPTERS 3, 4,** and **5** shows that GCF supports all of the use cases we have come up with thus far, it is still unclear how well these applications cover the total design space. In order to more thoroughly evaluate GCF's robustness, we conducted an exploratory study in which we asked developers to brainstorm applications that require devices to form groups and share context. We then analyzed each of these applications (using the design process presented in the previous section) to see which ones our framework can support.

The purpose of this study is twofold. Our first goal is to validate GCF's functionality. By soliciting app ideas directly from the developer community, we are able to evaluate GCF against a more diverse range of applications than we can come up with on our own. This gives us a more comprehensive understanding of which use cases GCF can and cannot enable, which in turn allows us to better assess if the framework can be used as a general purpose grouping and context sharing tool.

Additionally, this study lets us illustrate how GCF can support opportunistic groups outside of the lab. As part of our study protocol, we asked each participant to come up with application ideas on their own. By doing so, we are able to see how likely it is that applications will have similar or complimentary information requirements, and demonstrate why our framework is both necessary and useful.

In the following sections, we provide an overview of our methodology and participants. We then analyze the responses from participants, and show how our results validate our framework's architecture.

6.1. Methodology

We conducted a brainstorming study in order to better understand the types of applications that developers would like to create using GCF. For the first phase of the study, participants filled out a short survey that asked them about their demographics (*e.g.,* age, gender) and programming experience (*e.g.,* number of years, preferred development platform(s)). Afterwards, they participated in a semi-structured interview (conducted either in person or online) in which we asked them about the types of context-aware applications they have created and deployed, the types of context they have used, and whether or not they have ever created an application where context sharing was necessary and/or useful.

When the interview was complete, we introduced GCF, and described its main features. We showed them how the framework lets devices 1) form groups and share context without having to know about each other in advance, 2) collect and share context in an application-agnostic way *via* context providers, and 3) detect and broadcast context to nearby devices using Bluewave. We then gave each participant 10 days to brainstorm as many applications as they could that require devices to form groups and share context. Participants were asked to think of novel applications that utilize some or all of GCF's features, as well as ways to utilize opportunistic groups in their current and/or past work. Furthermore, while we gave participants GCF's complete documentation and working code examples, we did not require them to generate working prototypes of their applications. Instead, participants were asked to submit their app ideas (*via* our web form) regardless of whether they thought it could be implemented using GCF or not. Participants were directed to complete this phase of the study on their own, and were compensated up to $40 USD ($15 for filling out the initial survey and participating in the interview; $25 for submitting 5 ideas).

Through this process, we were able to quickly generate a diverse population of app ideas. While our methodology is not guaranteed to provide us with a comprehensive list of apps, prior work has shown that this "proof by example" approach can effectively validate a system's robustness. In [16], Carter evaluated the Momento system by showing how other scientists had used it to create Ubicomp experiments. In [66], Klemmer showed how Paper Mâché could be used to easily create Tangible User Interfaces by collecting feedback from 8 real-world developers. Finally, in [29], Dey *et al.* analyzed 60 context-aware applications from 20 non-programmers to show how their iCAP system could help end-users create a variety of trigger-based context-aware applications. While our study is inspired by this work, our methodology is somewhat different in that we are not interested in seeing how developers would utilize the current implementation of GCF. Instead, by having them work with our framework at a conceptual level, our study is able to investigate a broader range of applications, and validate that the framework is architecturally sound.

6.2. Participants

Our participants consisted of 20 software developers from both academia and industry. Most of our participants were male (85%), and ranged in age from 20-38 years old (mean = 28.3 years). Participants consisted of a mix of both novice and expert software developers (mean = 2.8 years, SD = 2.4 years). All of our participants were screened to verify that they have developed at least one context-aware application during their programming career. Additionally, all but one (95%) had experience creating context-aware applications on mobile operating systems (*e.g.,* Android, iOS, Windows Phone), and nearly half of them (45%) had worked on both mobile and desktop operating systems.

Through our questionnaire and interview, we learned how participants currently obtain context for their applications. As expected, most participants (60%) only used contexts that they could reliably obtain from a single device. Surprisingly, though, 40% of our population *did* have experience creating applications that required devices to share context. In each case, though, it was only with devices that were known *a priori*. When we asked participants what made context sharing challenging, all of them mentioned the lack of reusable tools as being the primary issue: "If it was easy to do sharing, I would, but I don't think of it as a possibility" (P12), "I spend too much time developing my own communications because it's not out of the box" (P7). 85% of our participants relied primarily on an operating system's default APIs to access context on the local device, and thus lacked a prebuilt way to transfer context from one device to another. Meanwhile, a smaller percentage of participants (10%) used toolkits such as AWARE [143] to make it easier to collect context from multiple devices. However, these systems are optimized to share context with a centralized server, and thus still required developers to provide their own custom communications logic if they wanted to share this information with other devices.

Across all of our interviews, we found that participants were generally interested in the idea of sharing context, but felt like the effort needed to implement context sharing from scratch (opportunistically or premeditated) currently outweighs the benefits: "I once spent more than a month just making sure the communications worked. Then I had to make sure the system was up and running all of the time" (P6). As a result, we were interested in seeing what types of applications they would *want* to build if this capability were provided to them for free.

6.3. Study Results

Through our study, we obtained a total of 65 application ideas from participants (for a complete listing, refer to **APPENDIX C**). In this section, we analyze these applications along multiple dimensions, and show which ones our framework is able to support.

6.3.1. What Types of Applications do Developers Want to Create Using GCF?

Developers provided us with a wide range of ideas for how they would use GCF to form groups and share context. Through our analysis, we have identified 7 high level application categories:

Awareness Applications
20 applications (31%) were proposed to improve users' knowledge of their immediate physical surroundings. In these applications, devices collect context from nearby users and/or devices, such as users' names, personal and/or professional interests, contact information, *etc*. This information is then presented to users so that they can better decide which entities they want to interact with.

> ***Representative Example:*** *"*Imagine a bunch of people just walking around with a GCF app that reveals details of people you may stumble upon everyday but that you have not talked to simply because you don't know them. What if through the shared context feature, a GCF app figures out common interests or activities, the sharing of certain location daily and brings people together that way.*"—P9*

Personalized Services
14 applications (22%) were designed to provide users with personalized information and services. Here, applications (installed either on devices placed in the environment or on the user's phone) are able to scan for nearby users and obtain context from them. They can then customize their behavior based on users' collective interests or desired end goals.

> ***Representative Example:*** *"Currently gate information (in airport) are shown on a large screen but there are so many of them and cannot easily find your information (especially if it's [an] international hub airport). My app works if multiple travelers approach to a screen to find gate information for transfer … suppose 10 travelers are standing in front of the screen and if 7 of them are heading to Pitt, 2 are heading to D.C, and 1 is to New York, the gate information for Pitt will show up in the first row while D.C on second row and New York on third. So, the screen interactively show[s] the information depending on need of users."—P7*

Request/Receive Context
14 applications (22%) allowed users to request and receive context from a specific set of devices (*e.g.,* all of the devices at a specified location). This information can then be used by the application to answer a specific question.

> ***Representative Example: "****I'm not sure what's the hours of a place, and I want to know whether there are people moving in the area (which can tell me whether it's open or not)"—P12*

Location Specific Services
8 applications (12%) proposed by developers are designed to provide users with location specific services. These applications check to see when the user has entered a specific location (*e.g.,* buildings, shopping areas). They then use their real-time knowledge of the environment to tell the user which services are available.

> ***Representative Example: "****Allows [users] to find an available room on campus to spontaneously meet in groups, e.g., for project meetings, brainstormings, etc."—P2*

Spontaneous Communications Applications
5 applications (8%) were proposed to support one-time communications between users. These applications allow users to send messages, ask questions, or share content (*e.g.,* pictures) without having to trade contact information in advance. This makes them especially useful in situations where users are meeting for the first time and need to exchange information once or without warning.

> ***Representative Example:*** *"The world [is] full with suffering. Give/receive nice, wonderful, encouraging messages from/to people who you walk pass today. Tell the user how many messages you get."—P15*

Recommendations Based on Similar Interests

In 2 applications (3%), developers wanted to use GCF to form groups of users with similar preferences (*e.g.,* favorite movies, food, books), regardless if they are physically nearby or have met. The applications can then use these groups to provide more personalized recommendations.

> ***Representative Example: "***People can ask about a place to eat or drink or go to and based upon their taste, the reviews of the people having similar taste will be sent to them on their location."—P1

Reminder Applications
Finally, 2 applications (3%) wanted to use GCF to remind users of important events. These applications are similar to existing recommendation systems (*e.g.,* Google Now) in that they can be configured to deliver a reminder at a specified date, time, and/or location. Additionally, however, the proposed systems can also use GCF to detect nearby users and/or devices. This lets them deliver reminders under a wider set of unpredictable conditions (e.g., "Remind me the next time I am near Anind to ask him if I can take his new car for a spin!").

> ***Representative Example: "****A user has a meeting in Anind's office. He wants to take a note or make a reminder for the next meeting in the office. So he executes a GCF-Enabled memo application, then enter some text. He also add[s] a reminder condition to the memo as a "current location". At that time, the app stores the entered text and its contextual data (e.g., indoor location). Later, when he visits Anind's office, the application will notify the stored memo. <u>Or if he configured the trigger condition as "meet Anind", the application will notify the memo when the app detects Anind's phone.</u>"—P6*

Collectively, these applications represent a good mix of old and new ideas. While some of these applications have been proposed before (*e.g.,* sharing business cards when at a conference [7,28,34]), others, to the best of our knowledge, have yet to be explored by either academia or industry. As a result, we believe these applications will collectively provide us with a good test set to evaluate GCF's capabilities, and help us better understand how our work can support and advance the state of the art.

6.3.2. *What Types of Context do Developers Want to Share Using GCF?*

In addition to identifying the above high level categories, we also analyzed each application to see what types of context they require at runtime. The results of this analysis are summarized in Table 16.

Not surprisingly, participants identified a wide range of contexts that they would like to collect and share using GCF. On average, each of the proposed applications require two different contexts (mean = 2.0; stdev = 1; min = 1, max = 5). Additionally, Table 16 also shows that that the types of context needed by each application can largely vary. Some applications only need access to raw sensor data (*e.g.,* GPS coordinates, audio amplitude readings) in order to function, while others need more specialized contexts that can only be collected *via* software, such as the user's identity, preferences (*e.g.,* music, food, accessibility settings), and calendar schedule. As of this writing, GCF only provides prebuilt context providers for 10 of the 26 contexts identified in Table 16. Upon closer examination, however, we found that *all* of the contexts specified by developers can already be obtained using existing sensors and/or software APIs. As a result, it is conceivable that future versions of the framework can provide prebuilt context providers for all of these data types.

There are two important takeaways from this analysis. First, our results show the importance of being able to support a wide range of context through a single architecture. Table 16 shows that developers will oftentimes require GCF to be able to share contexts produced using hardware, software, or any combination of the two. Many of these contexts will not come with prebuilt context providers, and will require the developer to collect this information themselves. Through the context provider abstraction, GCF provides a simple way for developers to collect and share a wide range

Table 16. Contexts used by participants' application ideas.

Context	Count	Collection Method	Included with GCF?	Description
Identity	33	Software		Name, Appearance, Contact Information
Location	29	Hardware	x	GPS Coordinate
Preferences	12	Software		Food, Books, Music, Accessibility, Allergies
Activity (Physical)	7	Hardware/Software	x	Determined by Google Activity Recognition
Audio	5	Hardware	x	Amplitude
User Response	4	Software		Answers to questions, surveys, polls
Destination	3	Software	x	GPS Coordinate
Schedule	3	Software	x	Calendar appointments
Altitude	4	Hardware	x	Determined by phone's barometer
Web History	3	Software		Recent search terms, websites
Photos	3	Software		Photographs taken from the user's camera
Bluetooth	3	Hardware	x	Nearby devices detected via Bluetooth
Activity (Phone)	2	Software		Phone calls placed, apps used
Compass	2	Hardware/Software	x	User's current heading
Payment Info	2	Software		Credit card information
Time	2	Software	x	System clock on the device
System State	2	Software		Current status, malfunctions
Accelerometer	1	Hardware	x	Phone orientation and movement
Flight Number	1	Software		The flight you are scheduled to take
Health Vitals	1	Hardware		Weight, blood pressure, heart rate
Shopping List	1	Software		Items you want to purchase
Social Network	1	Software		Friends on a social networking site (Facebook, Tinder)
Speed	1	Hardware		Velocity
Messages	1	Software		User created messages intended for another user
Coupons	1	Software		Shopping coupons
Wi-Fi	1	Hardware		Nearby Wi-Fi hot spots, SSID, signal strength

of context through a single software interface. This lets the framework work with any foreseeable context, regardless of what it is or how it is obtained.

Secondly, our results also show how GCF can take advantage common information requirements. As shown in Table 16, many applications use the same three contexts (identity, location, and (arguably) preferences). On its own, this is an important finding, as it means that GCF does not need to provide a large number of context providers in order for it to be useful in the most common use cases. Additionally, this finding also means that there is a good chance that applications will be able to find the context providers they need on other devices regardless if they are meeting for the first time or have the same app installed. This is a key part of our strategy for forming opportunistic groups, and in the following sections, we will show how GCF can take advantage of these commonalities without requiring developers to have to explicitly coordinate with each other.

6.3.3. *Which Applications Can be Created with GCF?*

In order to determine which applications can and cannot be created using GCF, we examined each one using the design process described in section 5.3. In doing so, we found that we can implement 64 of them (98%) using 5 general design patterns. In this section, we describe each pattern, and highlight the conditions under which they are best utilized.

Pattern 1: Local Area Request

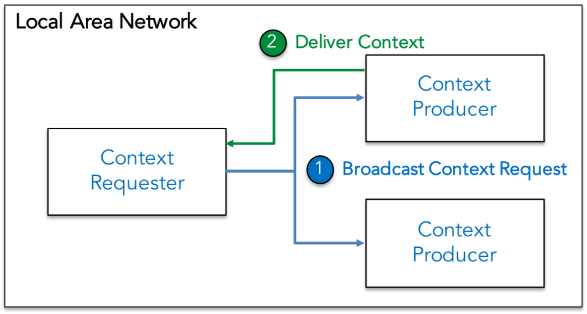


Figure 74. The local area request design pattern.

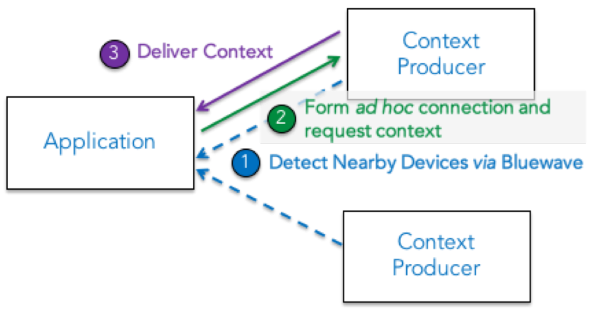
Our first design pattern is based on GCF's traditional grouping functionality. In this pattern, the entity requesting context (*e.g.,* an application running on a smartphone) broadcasts a context request message to all devices in communications range. The entity can then receive and analyze context capability messages from other devices, subscribe to one or more context providers, and begin receiving information.

The local area request pattern is useful for applications where the devices requesting and sharing context are:

1. In the same area (*e.g.,* a building or store), but not necessarily in Bluetooth range, and
2. Able to connect to the same local area network

While conceptually easy to understand, local area request is oftentimes impractical to use in the real world as there are many environments that do not have a local area network. Our participants *were* able to identify a small set of use cases where users could reasonably be expected to be on the same LAN (*e.g.,* an app that collects phone usage data from users in a classroom, concert, or sports arena to see who is mentally engaged—*P12*). Collectively, however, we found that the need for a local area network is a critical limitation, and that only six of the applications submitted by participants (9%) could actually be implemented using this pattern.

Pattern 2: Ad Hoc Request



3 Deliver Context
Context Producer
Application
2 Form *ad hoc* connection and request context
1 Detect Nearby Devices *via* Bluewave
Context Producer

Figure 75. The *ad hoc* request design pattern. Devices use Bluewave to detect nearby devices, and establish temporary connections with devices that can provide the desired information.

Our second design pattern takes advantage of GCF's ability to form *ad hoc* connections. In this pattern, the requesting entity performs periodic Bluewave scans to determine 1) which devices are nearby, 2) what contexts they can provide, and 3) what IP address they are listening on (using information shared by GCF by default). The entity can then use GCF to form an *ad hoc* network connection with one or more context producers, and request/receive information from them.

The *ad hoc* request pattern is useful for applications where devices are:

1. In Bluetooth range when they need to request/share context,
2. Connected to the Internet (Wi-Fi or 3G/4G), and
3. Expected to encounter each other in both indoor and outdoor locations

Since it does not rely on a local area network, the *ad hoc* request pattern is better suited for real-life spontaneous encounters (*e.g.,* an app that allows devices to share audio readings in order to determine if two users are hearing the same conversation—P13). This pattern is admittedly less efficient from a power consumption standpoint, as it requires the requesting device to conduct Bluetooth scans in order to locate and group with nearby devices. Furthermore, the pattern can only be used to form groups within a short distance. Yet despite these limitations, we we found that this pattern could be used to implement twice as many apps (twelve; 19%) as the local area request pattern.

Pattern 3: Bluewave Only

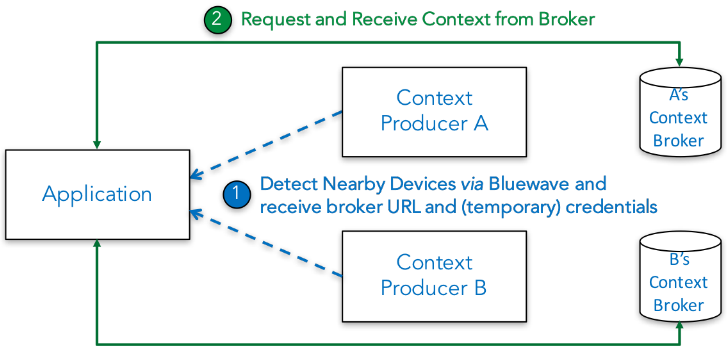

2 Request and Receive Context from Broker
Context Producer A
A's Context Broker
Application
1 Detect Nearby Devices *via* Bluewave and receive broker URL and (temporary) credentials
Context Producer B
B's Context Broker

Figure 76. Bluewave only design pattern.

Our third design pattern utilizes GCF's ability to broadcast and share context with nearby devices *via* Bluewave. Here, the requesting entity performs a Bluetooth scan to determine which devices are physically nearby. The application then uses GCF to automatically 1) extract each devices' context broker URL and credentials from its Bluetooth name, and 2) request and receive context.

The Bluewave only pattern is useful for applications where:

1. The devices requesting and sharing context are equipped with a Bluetooth antenna,
2. The devices requesting and sharing context have their own Internet connection (Wi-Fi or 3G/4G),
3. Devices only need to share context once or infrequently,
4. The context being shared is not continuously changing, and

The Bluewave only pattern provides a convenient way to share a small amount of context (*e.g.,* the user's name) over a short distance. Through our analysis, we found that this pattern could be used to implement approximately (30; 46%) of the applications submitted by our participants, ranging from simple icebreaker apps (*e.g.,* a part app that helps users find others with similar personal and professional interests—P3, P15) to intelligent displays (*e.g.,* a smart sign that displays everyone's exercise habits on a wall so that users can compare their performance—P10). This makes it the most commonly used pattern we have discovered thus far, and demonstrates why the ability to quickly and easily share context over short distances is an important capability of our framework.

Pattern 4: Relay

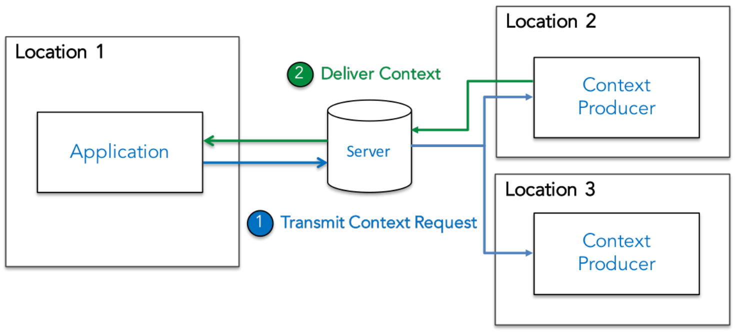


Figure 77. The relay design pattern. Devices send requests for context to a server or centralized communications channel (known *a priori*). These messages are forwarded to the other devices, and allow devices to form groups and share context over arbitrary distances.

In the relay pattern, each device connects to a centralized server that is known to all devices *a priori*. When an entity wants to request context, it sends a context request message to the server. This message is then relayed to the other devices (which may or may not be in the same location) so that the entity can form groups and share context over arbitrary distances.

The relay pattern is useful for applications where:

1. Devices need to form groups and share context over large physical distances,
2. The devices requesting and sharing context have their own Internet connection (Wi-Fi or 3G/4G), and
3. It is reasonable to assume that devices can be provided with the server's IP address and port beforehand

The relay pattern was utilized a total of 16 times (25%) by our participants. In these instances, participants wanted to be able to request context from devices at another location (*e.g.,* an app that lets users determine if a store or business is open by seeing if there are users moving in that location—P14). In the other seven applications, participants used a slightly modified version of the relay pattern (Figure 78) to upload user context to the server. This let the system *push* services (*e.g.,* an app that helps users find an available parking spot when they arrive at a shopping center—P1) when the user's context satisfies a set of predefined conditions.

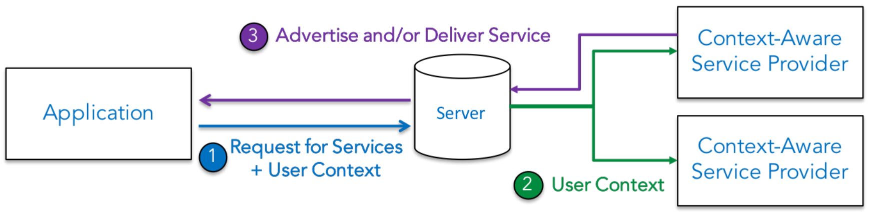

3 Advertise and/or Deliver Service
Application
Server
1 Request for Services + User Context
2 User Context
Context-Aware Service Provider
Context-Aware Service Provider

Figure 78. A variation of the relay design pattern. In this version, users share their context with a server, which forwards it to one or more service providers. These service providers can then opportunistically push information and/or services to users.

The relay pattern's biggest limitation is that it requires devices to know the address of the relay before they can communicate with each other. However, this pattern eliminates the need for devices to be nearby in order to form a group. This extends GCF's reach, which in turn allows our framework to share context under a wider set of conditions.

Pattern 5: Self Share

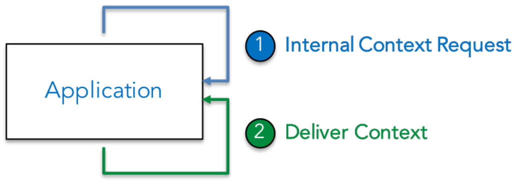


Figure 79. The self share design pattern. Devices request and receive context from themselves (using GCF's Local Only arbiter).

Our final design pattern takes advantage of GCF's ability to let devices request and receive context from themselves. In this pattern, the requesting application transmits a context request that uses the Local Only arbiter. The framework then subscribes to the device's local context provider(s), and begins receiving information.

The self-share pattern is useful for applications that:

1. Only need to collect context from themselves

Across all of the application ideas we received, only one (2%) needed this design pattern (*e.g.,* an app that monitors the user's location, and notifies him when he is near certain locations—P5). Nevertheless, we know that ability to obtain context from the local device is still an important capability in context-aware computing. Consequently, while this pattern does not make full use of GCF's capabilities, it *does* show how our framework can support traditional context collection tasks.

It is important to note that the five design patterns described above have actually been used throughout this thesis. The local area request pattern, for example, was used by Didja and Snap-To-It to let devices request and receive context and interfaces, respectively. Likewise, the *ad hoc* request pattern was used by GroupMap 2.0 to let devices share GPS coordinates in outdoor environments. Through our work with Bluewave, we showed how we can use the Bluewave only pattern to share context over short distances. Finally, Impromptu, Impromptu Share, and Impromptu Sense all demonstrate how we can use the relay pattern to request, receive, and push information to devices through a centralized application directory. The only pattern that we have not exclusively explored in this thesis is the self share pattern, as our work focuses on sharing context with *multiple* devices. Nevertheless, we have used this pattern in GroupHike to access a device's internal compass readings, as well as in Impromptu to collect user context.

Collectively, this work demonstrates that all of our design patterns can be implemented using the framework. This shows that GCF can technically support the types of systems that developers want to create.

6.3.4. Can GCF Support Legacy Context-Sharing Systems?

In addition to looking at participants' ideas, we also reevaluated each of the 28 context-sharing systems described in **CHAPTER 2** (Table 2) in order to determine if we could recreate them using GCF. Through this analysis (Table 17), we confirmed that the framework can be used to replicate all of the functionality we have mentioned or identified in prior work. Most of the systems proposed in **CHAPTER 2** require devices to share context over large physical distances, and as such can be implemented using GCF's relay pattern. However, we also identified a smaller set of applications which required devices to be able to share context over a local area network, form *ad hoc* connections, or broadcast context to nearby devices using Bluewave. Similar to the previous section, we were unable to identify a single application that only utilizes the self share pattern. This is to be expected, though, as these systems are focused on sharing context rather than collecting and using it internally.

Table 17. Analysis of the design patterns needed to recreate all of the systems mentioned in CHAPTER 2.

	Local Area Reqest	Ad Hoc Request	Bluewave Only	Relay	Self Share
System Name	GCF Design Pattern				
Active Badge and ActiveMap			x		
Hubbub				x	
ConChat				x	
Community Bar				x	
TeamSpace				x	
Group Interaction Support System (GISS)				x	
Context Aware Ephermeral Groups				x	
Panoply	x				
First Responder Reference Architecture				x	
Smart-Its Friends	x				
Social Serendipity			x		
Flocks		x			
MobilisGroups				x	
Personal Server		x			
Mobile Gaia		x			
Solar		x			
SenseWeb				x	
Virtual Personal Worlds		x			
Participatory Sensing Platforms*				x	
E2A2				x	
Remora		x			
ErdOS		x			
CoMon		x			

Across both sets of applications, we were unable to find a single application that could not be created using one of our five design patterns. This provides further evidence that GCF is robust enough to satisfy the vast majority of use cases, and that it is capable of being applied to both novel and existing context-aware systems.

6.3.5. Which Applications can we Not Create Using GCF?

Of the 93 systems we examined (65 participant applications, and 28 legacy systems), we only found one that our framework is currently unable to support:

> *"This app could allow self-driving cars to learn about other cars around and their navigation state. This for example could help emergency vehicles navigate faster through busy intersections without causing accidents.*

Similarly, in cases where navigation sensors like cameras or LIDARS may be failing the car could communicate this failure to other cars and prevent an accident."—P11

To create this application, we need a way for cars to form opportunistic groups while on the road. Yet while GCF provides this capability through Bluewave, the current framework requires 5-10 seconds to discover a nearby device using Bluetooth. When devices are traveling at a slow speed (*e.g.,* walking), this latency is acceptable as there is an extended period of time when the devices are nearby. In a driving scenario, however, cars may only be in range for a few seconds. Furthermore, while Bluetooth's effective range (15-30 feet) is good enough for most interpersonal encounters, it may not be sufficient to reliably detect devices over common driving environments (*e.g.,* cars at opposite ends of an intersection). Thus, while GCF could *occasionally* detect nearby devices and obtain context from them using Bluewave, it would not be reliable or consistent enough to be practically useful.

Through this example, we see that GCF's inability to quickly detect nearby devices can limit its ability to form opportunistic groups. One possible way to overcome this problem is to utilize additional radio technologies besides Bluetooth. RFID, for example, is commonly used on roads to detect cars when they pass through an intersection or toll route, and has both the range and speed needed to support the use case described above. Alternatively, it is also conceivable that future versions of Bluetooth could have a shorter discovery interval so that it can discover devices in a shorter window. For now, we have chosen to accept GCF's current limitations in order to allow it to run on the widest possible range of hardware. Overall, however, our analysis shows that GCF's biggest limitations are technological rather than architectural. This suggests that the framework, with the right combination of technologies, could eventually support applications like the one described by P11.

6.3.6. *Can GCF Support Opportunistic Grouping in the Wild?*

For our final analysis, we cataloged each application according to the conceptual model we introduced in **CHAPTER 2** (Table 1). Our results, which are summarized in Table 18, show that the majority of applications (66%) submitted by our participants utilize cooperative grouping (*i.e., Quadrant II*). This result, however, was expected. As part of our study protocol, as we asked participants to identify applications (*i.e.,* specific tasks) where forming groups and sharing context is useful. Consequently, while some participants identified ways to use GCF in a more general manner (*e.g.,* sharing sensor readings to conserve power—P3; sharing the user's current goal/task to allow multiple appliances in a smart home to customize their behavior—P8), the majority focused on collecting different data from each device in support of a single use case.

Given that most developers were focused on sharing context for their own purposes, can GCF actually form opportunistic groups across applications? To answer this question, we reexamined participants' application ideas from a platform standpoint in order to see how their contexts complement each other. Consider, for example, the following applications, which were proposed by three different participants:

> **Air Business Card (P15):** *"Imagine, you are in a conference or job fair. Would it be cool to always get your contact from people you meet[?] The app shows information in business cards or people around them. Users can decide to save some of it or filter only the one they want (or automatically save everything)."*
>
> **Student Attendance App (P2):** *"The app would make it easy for lecturers to gather information on who (aka mobile phone) has been attending a lecture or exercise session. This can be helpful for both lecturers and students for classes and labs where attendance is required."*
>
> **Simple Photo Sharing (P13):** *"Whenever people take a group photo, they often use more than one camera and still have to harass the people whose camera was used to share the photos. This app would just register everyone who is in a photo and send a copy of the picture to them automatically…"*

Table 18. Classification of participants' application ideas using our conceptual model.

	Need Same Data	Need Different Data
Performing Same Task	I. *3*	II. *42*
Performing Different Tasks	III. 2	IV. *17*

In each of these examples, users need to share their identity (*i.e.,* their name, contact information) in order to let them see who they have met, take attendance, and share photos, respectively. To create these applications using GCF, we could create an Identity context provider that can collect this information from the user's phone. Each developer could independently use this context provider to obtain this information for their own applications. The applications would then be able to share the same context with each other, thereby allowing users with the Simple Photo Sharing app to email pictures to users with the Air Business Card and Student Attendance Apps.

Although this example seems coincidental, it is important to note that this is not the only example of cross application compatibility that we observed. Across our entire dataset, we have identified 102 separate instances where apps with similar or complimentary contextual information requirements are able to provide information to each other. Given that our participants were not allowed to share ideas with each other, our dataset closely approximates the types of context-aware applications that might exist in a realistic deployment of our framework. Thus, our results provide strong evidence that GCF will be able to find and form opportunistic groups outside of a laboratory environment.

6.4. SUMMARY

In this chapter, we asked developers from both academia and industry to tell us how they would like to use GCF to create a new generation of context-aware applications. Through this process, we showed that our framework is not only able to technically support the vast majority of use cases, but that it can also allow devices to form groups and share context regardless of whether or not they are performing the same task.

The work presented in this chapter concludes our discussion of our first two research questions. Yet while our work has allowed us to validate GCF's architecture (addressing RQ1) and ensure that the framework supports a wide range of use cases (addressing RQ2), it has also helped us better understand the specific challenges with allowing devices to opportunistically form groups and share context from a developer, end-user, and system perspective. In the next chapter, we focus on this topic in greater detail, and identify the unique challenges associated with using a framework like GCF outside of a laboratory environment.

7. Identifying the Challenges with Utilizing Opportunistic Groups in Context-Aware Computing

The Group Context Framework provides a flexible research platform to explore how opportunistic groups can be utilized in context-aware applications. Yet while our work has helped us identify the potential *benefits* of using a framework like GCF, it has also helped us identify the unique *challenges* that can arise when devices are able to form groups and share context on their own. In this chapter, we group these challenges into three broad categories. First, we discuss the potential issues that can make it difficult for application developers to use GCF in their own applications. Afterwards, we describe the issues that can discourage users from wanting to share context in an opportunistic fashion. Finally, we look at GCF at the technical level, and discuss how the hardware, power, and networking requirements of our framework create practical limitations on the types of use cases that can be easily and practically supported.

The work presented in this chapter directly addresses our third research question ("What are the unique challenges associated with utilizing opportunistic groups in context-aware applications?") by identifying specific conditions where the framework either breaks down or becomes difficult to use. In order to provide as complete a list as possible, this chapter examines the issues that we encountered in our own work (**CHAPTER 3** and **CHAPTER 4**), as well as issues brought up by developers during our developer study (**CHAPTER 6**). Additionally, we have also reexamined the application ideas submitted by developers in **CHAPTER 6** to better understand the challenges that can arise when GCF is utilized as a platform. Through this exploration, we identify the most important issues that affect GCF's usability. This, in turn, will provide researchers a general body of knowledge to help inform the design of future context-aware systems, frameworks, and research.

7.1. Developer Challenges

In this section, we discuss three challenges that can make it difficult for developers to use GCF. First, we discuss the issue of platform fragmentation, and present several ways to encourage developers to utilize a standardized set of context providers rather than create their own. Next, we look at the challenges associated with debugging applications built using GCF, and describe the strategies that we have used thus far to test our applications prior to deployment. Lastly, we discuss the challenges associated with relying on context produced by other devices, and describe two techniques that devices can use to determine if they are trustworthy when they meet for the first time.

7.1.1. Platform Fragmentation

GCF's context providers provide a simple and extensible way to collect and share a wide range of context in an application agnostic manner. Yet while this abstraction allows devices to form groups and work together with minimal coordination, it can also fragment the platform if used improperly. For example, if two developers create context providers for the same type of information (*e.g.,* a user's shopping list), it can create competing ontologies, and prevent devices from being able to easily request or use this information. Similarly, if developers modify a context provider's output (*e.g.,* changing a temperature context provider so that it outputs values in degrees Kelvin instead of Celsius) without keeping other developers informed, they run the risk of breaking every application that relies on it for information.

Our experience with these types of problems is currently limited since we have only used context providers that we have created and deployed ourselves. However, as GCF becomes more widely adopted, the risk of fragmentation will increase. We already have empirical evidence that suggests this will happen. In our developer study (**CHAPTER 6**, Table 16), we found that 80% of the application ideas submitted by developers required them to develop at least one custom context provider. Additionally, we observed 8 separate instances where two or more developers needed to create a custom context provider for the same type of information (*e.g.,* user preferences, identity). Thus, while developers may not intentionally try to fragment GCF, our results suggest that they will do so if left on their own.

From this discussion, it is clear that future generations of GCF will need to provide more direct oversight over how context is collected and shared across the framework. This oversight can take several different forms. One possible way to reduce fragmentation is to help developers better understand what information is already shared through GCF. Our web dashboard (Figure 65) partially addresses this need by allowing developers to see which context providers have already been deployed on the framework. This lets them decide when they need to create a new context provider. Another way is to have context providers report their version number so that applications can 1) detect when a context provider has been updated, and 2) request context from specific versions. Finally, future versions of GCF may need to either come with an extensive library of prebuilt context providers, and/or provide mechanisms for developers to submit and share their context providers with each other (*e.g.,* using an online code repository like GitHub). Even with these measures in place, it is not realistic to assume that platform fragmentation can be completely avoided. Yet by encouraging standardization at the platform level, it should be possible to allow devices to form opportunistic groups under many common situations.

7.1.2. Debugging

Another challenge with utilizing opportunistic groups in context-aware computing is that the resulting applications can be more difficult to debug. As mentioned in **CHAPTER 1**, one of the main characteristics of opportunistic groups is that it is hard to predict when and where they will occur. Yet while GCF addresses this problem by allowing devices to detect and form groups on their own, the developers from our study noted that it can be challenging to anticipate all of the groups that can be formed by the framework at runtime. This can make it difficult for them to come up with a set of test cases that thoroughly evaluates their applications, or precisely recreate the conditions (*i.e.,* the exact combination of devices) that led to a bug or software defect.

In own experience, allowing devices to form opportunistic groups *does* add an additional layer of complexity to the software development process. However, we have also found that there are still practical ways to debug applications created using GCF. In systems such as Didja and Snap-To-It, we were able to create a test environment in our lab which contains one or more GCF-enabled devices. We could then use this environment throughout the development process in order to simulate a wide range of real-world encounters. In contrast, when we developed the Impromptu Share system, we realized that we needed to be able to simulate large numbers of devices in order to see how our system could request and receive information from all of them at once. To achieve this, we ran multiple instances of GCF on a laptop, and had the system request and receive context from them. Using this approach, we could simulate any number of devices without the need for actual hardware. Additionally, since our simulated devices were using the same context providers, we were able to swap them for real devices without having to alter our application's code.

Collectively, these examples show that debugging GCF applications is more practical than it might seem at first. One limitation of our current approach is that it still requires developers to manually test their applications (*i.e.,* bring two devices into range so that they can detect each other and form a group). This means that there is a good possibility that developers will be unable to thoroughly test their applications against every possible use case. Although it is beyond the scope of this thesis, it is conceivable that future versions of the framework could come with automated testing tools (*e.g.,* a dedicated simulator environment—see our discussion of related work for more details). This would conform to existing software engineering best practices, and give developers a more systematic and efficient way to verify that their programs work as intended.

7.1.3. Trusting Context from Newly Encountered Devices

A third issue that developers need to consider when using a framework like GCF is determining how to trust context from newly encountered devices. Throughout this thesis, we have largely assumed that the devices running GCF are not malicious, and that the context that they report is both accurate and truthful. This is not a problem when the framework is only being used to help users share *e.g.,* potential conversation topics, as there is little incentive for users and/or devices to lie. As seen in our work with Snap-To-It and our developer study, however, there is interest in using GCF in order to perform a diverse range of secure transactions, ranging from unlocking computers to performing financial transactions. In these use cases, there is more of an incentive for a malicious device to pretend to be someone else (a technique commonly referred to as *spoofing*) and/or provide inaccurate information (user identity and payment information, respectively). Consequently, it is necessary for devices to have some way to establish trust when they meet for the first time so that the device receiving the context knows that the information comes from a reputable source.

As noted in our discussion of Snap-To-It (**CHAPTER 4)**, establishing trust between unfamiliar devices is a challenge in every networked system. Consequently, there are many well-established techniques that have been developed to overcome this problem. One way is to utilize a *hierarchical* trust model in which devices are vetted by a third party (*i.e.,* a certificate authority or OAuth service [144]), and receive a digital certificate (*e.g.,* X.509 [145]) stating that they are trustworthy. When devices meet for the first time, they can exchange certificates and validate them with the certificate authority. This lets them know that the devices can be trusted before requesting and receiving information from them. Another popular technique is to use a *web of trust* model [51] in which each device creates its own digital certificate, and forms *ad hoc* trust relationships with specific devices. The devices can then form new trust relationships with newly encountered devices *via* the transitive property (*e.g.,* "If A trusts B, and B trusts C, then A trusts C").

Given that the focus of this thesis is on identifying use cases where opportunistic groups are needed, we did not design the GCF with a specific trust model in mind. It is important to note, however, that the framework can be used with either the hierarchical or web of trust methods. GCF allows applications to insert their own information when transmitting a context request message, and thus it is already possible for applications to insert a digital certificate or other form of authentication as one of these parameters. Additionally, it is conceivable that future versions of GCF could come with a built-in trust mechanism so that the framework can automatically establish trust relationships on its own. However, it is unclear if the latter option would actually prevent developers from having to implement their own trust mechanisms into their applications, and thus it is an open question as to whether or not this approach would actually be welcomed or useful.

7.2. End User Challenges

In this section, we describe three challenges that can discourage users from wanting to use applications that are built with GCF. First, we examine GCF from a user privacy standpoint, and discuss the difficulties in protecting users' information from being openly shared without repeatedly asking them for permission. Afterwards, we discuss the potential security implications of our framework, and consider how it addresses (or can be extended to address) important concepts such as confidentiality, integrity, and availability. Finally, we look at how the ability to form opportunistic groups can overwhelm users with extraneous information and services, and offer suggestions as to how users can take advantage of GCF's functionality without significantly increasing their cognitive load.

7.2.1. Privacy

GCF's design is based on the philosophy that devices should be able to form groups and share context as needed. Yet while our work shows that this goal is both technically achievable and creates new interaction opportunities, the idea of allowing devices to automatically share personal information is potentially concerning to end users. Consequently, a significant challenge in using GCF is convincing users that the technology gives them control over their information, and that they actually want to share their context in this manner.

We have explored the issue of privacy on numerous occasions in this thesis. During the development of Impromptu, for example, we realized that users may be uncomfortable with the idea of continually sharing their identity, activity, location, *etc.* with our system's application directory. This led us to develop techniques to obfuscate sensitive information (*e.g.,* sharing the hash of an email address rather than the address itself), and add basic privacy controls so that users could specifically identify what types of information they are willing to broadcast. Meanwhile, during the early phases of our work with Bluewave, we conducted a study with 15 participants in order to determine what types of information users would likely be willing to share in an opportunistic manner (section 4.4.1.3). We then used this feedback to develop finer grained privacy controls so that users can see what information applications need, and grant permissions on an individual basis. Finally, as part of our exploration of Bluewave, we created a number of prototypes, ranging from self-translating signs to navigation aids, that only required a small amount of context (*e.g.,* a user's language preferences and navigation destination, respectively). In doing so, we showed how GCF can be utilized without revealing personally identifiable information, thereby providing an alternative, and more privacy friendly way to utilize opportunistic groups in context-aware computing.

Through this work, we have gained two important insights as to how frameworks like GCF can alleviate user's privacy concerns. First, our work emphasizes that users want direct control over what information they are sharing and when. In our exploratory work with Bluewave, for example, we have found that users were not opposed to the idea of sharing context so long as the decision to share is left to them. Based on this, we created a standardized set of privacy controls to let users 1) see what information is being collected by GCF, and toggle which ones they would like to share, and 2) see when the framework is actually providing context to other users and/or devices. Users can then modify their sharing settings to more closely match their personal comfort levels.

Secondly, our work shows that users are more willing to share information when they know what services and/or benefits they are getting in exchange. During our work in Bluewave, we learned that users tended to frame context sharing in terms of a financial transaction (*i.e.,* "If I give a device context A, then I will get B in return"), and that they were more willing to share when the benefits of doing so outweighed the costs. Consequently, it is important that applications make the benefits of sharing more explicit. To support this, we have modified GCF so that applications can tell users what context they need at runtime, and what information/services they provide in exchange. Users can then view this information and decide whether or not they want the group to be formed.

Although these insights have already influenced GCF's final design, our current implementation only provides a partial solution. It is technically possible, for instance, to provide users with additional controls so that they can precisely specify if, when, and with whom their context is shared (*e.g.,* "Share my location only with my female friends between 8am to 5pm on weekdays"). Additionally, it is also feasible to mandate that every application provide detailed descriptions of their functionality and information requirements so that users can better decide which devices they want to group with. Requiring users to be involved in every grouping decision, however, increases their cognitive load, and can prevent GCF from taking advantage of opportunistic groupings as they naturally and spontaneously occur. From a user experience standpoint, there is a balance that needs to be struck between giving users fine-grained control when they want to share information, and allowing them to benefit from GCF's ability to autonomously detect and form groups. Giving users the ability to openly share *some* (but not all) context is one way to let users delegate responsibility to the framework, and prevent them from having to make every grouping decision. However, it is still an open question as to what additional controls are needed to help users find a balance that works for them.

7.2.2. SECURITY

Another challenge with getting users to adopt GCF is with regards to security. Our framework relies on broadcasting technologies (*e.g.,* UDP multicast, Bluetooth radio IDs) and plaintext (*e.g.,* JSON) in order to discover devices and share context, respectively. Although this makes it easy for devices to work together when they meet for the first time, it also allows malicious entities to collect and/or modify sensitive information about users (*e.g.,* payment information, user credentials) when is it transmitted through the framework. This risk is acceptable in a laboratory environment, but effectively prevents GCF from being able to support use cases where confidential information is needed.

While we did not focus on security in this thesis, we recognize that it is important to consider how the framework can be used to transmit sensitive information. According to [88], there are three important properties that a system needs to address for it to be considered secure. The first property, *confidentiality*, is concerned with making sure that the information being transmitted is only available to its intended recipient(s). To date, this is most commonly achieved through the use of encryption algorithms. Although GCF does not currently use encryption, it is reasonable to assume that future versions of the framework could allow devices to encode context using either a public key infrastructure (e.g., RSA [97], PGP [146]), or through a symmetric key known only to the sender and receiver (distributed using an algorithm such as Diffie Hellman [32]). This would allow members of the same group to see and use the data, while preventing outsiders from doing the same.

The second property, *integrity*, is concerned with making sure that data has not been altered from the sender to the receiver. Once again, GCF does not currently check context data messages to see if they have been altered on route to their destination. However, one commonly accepted way to do so is by using hashing algorithms. By having GCF's communications manager compute the hash a communications message prior to transmission (and inserting the value in the message's payload) and once it is received, the framework would automatically detect if tampering has occurred. This can increase the application's ability to trust context from other devices, and reduce the chances of successful man-in-the-middle attacks.

The third property, *availability*, is concerned with making sure that information is accessible when needed. GCF's ability to be run as a background service addresses this property to some extent, as it ensures that the framework is always ready to receive and respond to incoming context requests. A more pressing concern, however, is how the framework will respond when dozens, or even hundreds of devices are simultaneously requesting context, as doing so can lead to a *denial of service* (DoS) situation. For now, GCF has been configured to ignore context requests from devices if they occur too frequently (more than once every 10 seconds), and can even make use of channelized communications protocols (*e.g.,* MQTT) to limit a malicious device's ability to request context from multiple devices at once. Despite these precautions, the problem with maintaining availability of services is well documented in the literature [47]. As a result, it is still largely unknown if these measures will be sufficient in a large-scale deployment.

From this discussion, it is clear that there is still much work that needs to be done before GCF can be considered secure. Although our discussion identified a number of areas that can be improved upon, it should be noted that all of the enhancements mentioned above make use of existing technologies and/or techniques. Thus, while GCF may not be provably secure today, it is conceivable that future versions of the framework can prevent malicious users from taking advantage of the framework's open nature to access and use information without permission. This in turn will make GCF more practical to use.

7.2.3. COGNITIVE OVERLOAD

Finally, GCF's ability to form opportunistic groups can have an adverse effect on a user's cognitive load. As our work in CHAPTERS 3-5 show, it is technically possible to develop a GCF application that can continually collect sensor data from nearby smartphones, or form an opportunistic group with every appliance in Bluetooth range. Yet while these systems can increase users' access to relevant information and services, they become cumbersome if they have to manually analyze multiple streams of context to find the one piece of information they need, or identify a specific appliance from a list containing dozens, or even potentially hundreds of items. By increasing the opportunities for devices to obtain context, GCF can also inundate users with extraneous data. This can increase users' cognitive load, and draw their attention away from important tasks.

Although cognitive load is a problem, it is important to note that it is not unique to GCF. On the contrary, the risks of cognitive overload have been well studied by the context-aware community, and researchers have developed numerous techniques (utilizing rule or role based architectures [73], machine learning [86], novel visualizations [119]) in order to limit the amount of context that a user has to see and process. In our own work, we have tried to reduce cognitive load by carefully designing our systems to minimize the amount of information that they present at a time. In Snap-To-It, for example, we only showed users a list of the top five appliance matches as opposed to every appliance in range, and compared multiple types context (*e.g.,* location, device orientation, photographs) to ensure that the appliances at the top of the list had the best chance of being the one that the user selected. Meanwhile, in our work with Impromptu, we implemented tools to let users sort opportunistic apps by category, and generated a passive notification when a new app was found to be contextually relevant. This let users be aware of new apps without having to keep our app in the foreground.

In fairness, the solutions we used in our work cannot be easily generalized to every GCF application. However, every application is unique, and as such, designers need to consider how much information should be presented to the user at any given time. As a result, while it would be interesting to see what safeguards can be incorporated into GCF to help alleviate cognitive overload in users, we believe that these issues can largely be resolved at the design level, and as such should be left to developers.

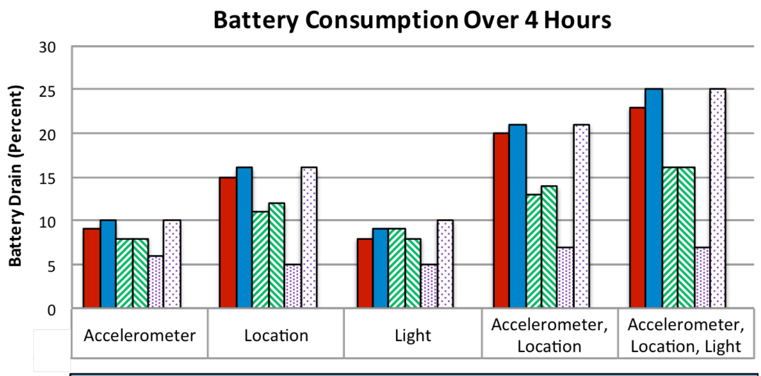
7.3. Technical Level Challenges

In this section, we explore the challenges associated with finding and forming opportunistic groups at the technical level. First, we look at GCF's battery consumption and data usage, and discuss the specific tradeoffs that need to be made when using our framework. Afterwards, we address the issue of scalability, and describe our experiences trying to form large groups of devices using the framework.

7.3.1. Power Consumption

A critical challenge with leveraging a context sharing framework like GCF is finding a balance between the increased functionality our framework offers and battery consumption. During our work with Didja, for example, we showed how using GCF to collect and analyze sensor data from multiple devices could provide users with a useful service (*i.e.,* finding relevant groups in arbitrary environments). To do so, however, the system needed to continually scan for nearby devices using Bluetooth, and request, receive, and process context from other devices. This resulted in the system consuming approximately 20% of each device's battery life *per hour*, thereby making it too inefficient to be left continually running. In our experience, this result is a worst-case assessment, as many of the applications we have created using Didja or GCF (as well as those proposed by developers in CHAPTER 6) do not require smartphones to perpetually request and receive context. Nevertheless, this work serves as a warning that the energy costs of forming groups and sharing context can be significant, and can degrade the user experience if left unchecked.

We have studied GCF's power consumption on numerous occasions in order to better understand its energy costs. As part of our initial work (CHAPTER 3), we conducted a series of power drain experiments to analyze the impact of our framework when devices are able to share sensor data with each other. For these experiments, we had two identically configured Galaxy S IV smartphones request and receive various combinations of context (accelerometer, location, and light) with each other once every second for 4 hours, and tested scenarios where 1) both devices produce and



	Accelerometer	Location	Light	Accelerometer, Location	Accelerometer, Location, Light
	Baseline (No Context Sharing)				
🟥 One Device (No GCF)	9	15	8	20	23
🟦 One Device (GCF)	10	16	9	21	25
	Both Devices Need Context (Produce and Share Equally)				
▨ Device #1	8	11	9	13	16
▧ Device #2	8	12	8	14	16
Avg Savings (vs. No GCF)	11.1%	23.3%	-6.3%	32.5%	30.4%
	One Device Needs Context (Only Second Device Produces and Shares)				
⬚ Device #1	6	5	5	7	7
☐ Device #2	10	16	10	21	25

Figure 81. Results from our power drain experiments with sharing one, two, or three streams of context simultaneously. We deployed GCF on two smartphones, and measured the power consumed when devices are not sharing context, both devices are evenly sharing context, and when only one device is sharing context. All values represent battery loss in percentage points.

share context with each other, and 2) one device produces context for another. Our results, shown in Figure 80, not only reveal that using the framework consumes little power on its own, but that it can actually result in power savings so long as devices take turns producing and sharing multiple streams of context over an extended period of time. Our results *do* show that devices can consume more battery life if they are continually producing contexts for others without receiving any services in return. On average, however, our results suggest that the costs of GCF are manageable, and that the opportunities for energy savings will increase as the size of the group increases.

We also explored the issue of power consumption during our work with Impromptu. As discussed in section 4.3.3.1, we ran Impromptu on a Nexus 5 smartphone for 8 hours, and compared its battery consumption to when the phone ran for the same period of time without our app installed. Our results (Table 10) showed that the system only consumed 20% more battery than our idle condition. It is difficult to directly compare these results with the previous experiment, as Impromptu was designed to share context at a much lower rate (*i.e.,* once every minute as opposed to once every second). Still, our results show that GCF can be practically run all day without significantly hampering the end user experience.

Finally, in section 4.4.2.2, we performed experiments in order to see how much power was consumed when devices used Bluewave to broadcast context or scan for nearby devices. Our results (Table 12) showed that the former only consumes an additional 6% battery life per day, making it practical to be continually enabled on users' smartphones. Our results also showed that the latter configuration consumes significantly more battery life (29% per day). In practice, however, we have found that most applications either 1) only required users' devices to scan for a short period of time, or 2) required devices positioned in the environment to scan for the user. This means that the system only needs to run in its low power configuration to support the majority of use cases.

Collectively, this work has given us a better understanding of GCF's energy costs under various configurations. Our work shows that GCF *can* significantly drain the battery if it is continually producing and sharing context with other devices, and/or scanning for other devices. At the same time, it also demonstrates that the framework can run all day on a device if applications collect or share context at a reduced rate, or only utilize a subset of its features. Based on our results from our developer study, we expect that the majority of use cases for GCF will only need to share context once or for a short period of time, thus making the energy costs of our framework negligible. However, it will be up to developers to find a balance between functionality and efficiency that works best for their applications.

7.3.2. Data Usage

Another challenge with using a framework like GCF to collect context is managing its data consumption. Although our framework only consumes a small amount of data when devices only need to share information (*e.g.,* a GPS coordinate) once or sparingly, its data usage significantly increases when applications need to receive continual updates from multiple devices over an extended period of time. Furthermore, since all communication in GCF is transmitted over broadcast channels, there are occasions where devices will receive messages that are not intended for them (*e.g.,* context data or capability messages). As long as GCF is only used over a Wi-Fi network, the amount of data consumed by our framework is negligible. Many users, however, have a limited amount of data that they can transfer over a cellular network (*e.g.,* 3G/4G). Consequently, as GCF is used in a wider range of indoor and outdoor environments, it is important to keep the framework's data usage in check so that users do not incur unexpected operational costs.

We have tried to address the problem of data usage on two fronts. At the networking layer, we have tried to make GCF's communications protocol as efficient as possible. As shown in Table 14, a standard context request message takes up less than 100 bytes, while a typical context data message (*e.g.,* a message containing a GPS coordinate) can be encoded in less than 150 bytes. By keeping each message small, we are able to minimize GCF's data footprint. This reduces the cost of receiving a message for another device, and makes it possible for devices to share context for an extended period of time without consuming large amounts of data.

Additionally, we have also tried to minimize GCF's data usage by taking advantage of more efficient communications technologies. When we created Impromptu, for instance, we realized that it would be impractical for devices to communicate with the application directory over a UDP multicast channel, as doing so would also cause users to receive each other's context. To address this, we created a new communications thread that can send messages to specific channels on an MQTT broker. Our host application can then transmit the user's context to the application directory's channel, and listen for relevant apps on a device-specific channel. Although this solution requires additional hardware (*i.e.,* a dedicated communications servers), it prevents GCF from having to receive and ignore messages intended for other devices. This lets the framework operate in environments where multiple GCF-enabled devices are requesting and receiving information at the same, and allows applications to run perpetually in the background without consuming large amounts of unnecessary data.

Although this work takes steps towards reducing GCF's data usage, it is important to note that it is up to developers to use the framework responsibly. It is still possible, for example, for applications to insert a large amount of data into a context data message, or request context at an unnecessarily high rate (*e.g.,* requesting location data every 100ms when the user is most likely not moving that quickly). As a result, while GCF provides the basic safeguards to minimize data usage, it is still up to developers to determine how to most efficiently utilize the framework in their applications.

7.3.3. Scalability

The third challenge we have identified with using a framework like GCF to form groups and share context is scalability. In order to participate in a group, GCF requires each device to 1) maintain an open communications channel, 2) filter incoming messages, 3) transmit requests for context, and 4) produce and deliver context as needed. When the size of the group is small, the cost of these operations is negligible. In our developer study, however, we found that developers also want to use GCF to group with a larger number of devices at the same time (*e.g.,* all of the devices in a room or conference). In these use cases, devices need to spend a larger proportion of their time finding, forming, and communicating with group members. This increases GCF's overhead, and can prevent devices from having the computing resources needed to reliably perform other tasks.

Although many of our example applications focus on small groups (*e.g.,* 2-3 devices), our work shows that the framework *can* applied to larger sized groups. In Didja, for example, we successfully used GCF to collect and analyze over twenty independent streams of context without any noticeable lag. Meanwhile, for the past year, we have successfully deployed Impromptu to over 60 concurrent users, and have been able to analyze their context and recommend useful information and services as they go about their normal routine. Finally, during our collaboration with the Huntington County Emergency Management Agency (Section 4.3.2.2), we deployed Impromptu Sense to 28 volunteers' smartphones at a music concert. We then spent the next three days requesting and receiving context from these devices over a highly congested cellular network. In each of these examples, the maximum size of the group was dictated by the number of devices we could install our software on rather than the framework's upper bound performance. As a result, while our results show that GCF is scalable enough for many context-aware applications, we expect these systems to form even larger groups in the future.

Through this work, we have found that GCF's scalability depends in large part on: 1) the reliability of the communications channel, and 2) the amount of time needed for devices to process context. In Didja, for example, we were able to share context with devices at a high rate (*i.e.,* once every 250 milliseconds) because the devices were all connected to a stable Wi-Fi network. In contrast, when we tried to collect context in Impromptu Sense, we found that the cellular network was unable to reliably transmit sensor data to our MQTT server at the same rate; to overcome this, we had devices insert multiple sensor readings into each context data message, and transmit them at a lower rate (*e.g.,* once per second). Meanwhile, in our work with Impromptu, we discovered that the application directory took approximately 10 milliseconds to analyze a user's context and recommend relevant apps. In order to accommodate potentially hundreds of concurrent users, however, we decided to only have host applications report their context once per minute. This prevented the system from becoming inundated with data, and allows Impromptu to support a larger user base than what we have been able to directly test.

Thus, while GCF's architecture does not place any restrictions on the size of the groups that can be formed, our work shows that there will always be practical limitations to the amount of context that devices can request, receive, and process given their hardware and networking capabilities. Our work shows that GCF can potentially be scaled to accommodate larger sized groups. However, it is up to developers to carefully consider how much context their application needs so that they can choose a group size that is both useful and realistic.

7.4. Summary

In this chapter, we have described the challenges associated with utilizing opportunistic groups in context-aware computing. First, we looked at the developer level, and identified three challenges (platform fragmentation, debugging, and trusting context from newly encountered devices) that can make it difficult for developers to easily incorporate GCF into new and existing applications, or rely on the information it provides. Afterwards, we noted how issues like privacy and security can prevent users from wanting to use the framework, and looked at possible ways to augment GCF in order to minimize the framework's impact on the user's cognitive load. Finally, at the technical level, we discussed how the framework consumes power, uses data, and scales under various configurations. We then described the various tradeoffs that need to be made in each of these areas in order for the framework to be usable.

Table 19. Summary of the challenges with using GCF from a developer, user, and technical standpoint.

	Description	Areas for Further Exploration
Developer Level Challenges		
1. Platform Fragmentation	Developers can accidentally create multiple context providers for the same context (*e.g.*, location), which can lead to competing standards.	1. Identify the most commonly used contexts, and provide prebuilt context providers for them. 2. Create integrated IDE tools to help developers understand what information is already shared using GCF.
2. Debugging	It is difficult for developers to anticipate and/or recreate the conditions under which an opportunistic group can form. This makes it hard to test applications or recreate bugs.	1. Create a robust simulator environment to recreate or replicate common grouping conditions and/or device configurations.
3. Trusting Context from New/Unfamiliar Devices	GCF currently lacks any way to verify that the context provided by another device is valid and/or accurate.	1. Integrate a top-down or bottom-up trust model to let devices know which group members can be trusted.
User Level Challenges		
1. Privacy	Users want explicit control over how their context is shared, but this can prevent devices from being able to form groups autonomously.	1. Determine what level of controls users would like to have when managing their privacy settings. 2. Create automated tools that can learn users' privacy preferences over times.
2. Security	GCF currently lacks mechanisms to ensure the confidentiality, integrity, and availability of context.	1. Add encryption to GCF so that devices can encrypt context, and verify that information has been tampered with.
3. Cognitive Load	The ability to form opportunistic groups can inundate users with unnecessary information and services.	1. Identify design guidelines to minimize cognitive load in GCF apps. 2. Identify features that can be added to GCF to reduce or mitigate the effects of cognitive overload.
Technical Level Challenges		
1. Power Consumption	GCF can significantly increase a device's power consumption if devices are continually requesting, receiving, and sharing context.	1. Monitor GCF's power consumption on user devices to better understand its real-world energy costs.
2. Data Consumption	GCF can significantly increase a device's data consumption if devices are continually requesting, receiving, and sharing context.	1. Monitor GCF's data consumption on user devices to better understand its real-world bandwidth costs.
3. Scalability	Devices are limited in the amount of context they can receive and process, which limits the size of the groups that they can form.	1. Perform larger scale tests to determine GCF's practical grouping limitations.

Table 19 summarizes the challenges that we identified in this chapter, and offers possible directions for further exploration. Although our work answers our third research question, it should be noted that the idea of allowing devices to form groups on their own is still a new concept, and that we expect more challenges to arise as the framework becomes more widely adopted and used. By using GCF as a research test bed, however, our work gives both users and developers an improved understanding of the types of issues that they are most likely to face when they utilize our framework for the first time. It is important to stress that many of the challenges identified in this chapter are inherently complex, and do not have a quick or simple fix. Yet by highlighting GCF's current benefits and limitations, it is our hope that our work can motivate and inspire further generations of research to make the framework even more usable that it is today.

8. Conclusions and Future Work

This thesis has presented a novel framework that allows devices to form opportunistic groups and share context. Our work not only describes the technical architecture and abstractions needed to allow devices to detect each other and form groups with minimal prior coordination, but also investigates 1) the types of applications that benefit from this capability, as well as 2) the challenges with utilizing our framework from a developer, user, and system standpoint. In this chapter, we summarize our research, and recommend areas for future work.

8.1. Summary of Research

In **CHAPTER 1**, we motivated the need for devices to be able to form opportunistic groups in context-aware computing. We provided an operational definition of the term *opportunistic group*, and presented a series of motivational examples that showed how allowing devices to automatically detect each other and form groups increases their access to relevant information and/or services. Afterwards, we conducted a survey of existing context-sharing systems and/or frameworks in **CHAPTER 2** in order to identify both the high level use cases that benefit from context-sharing, as well as the software architectures that have been proposed thus far to support them. Through this process, we developed a conceptual model that identifies the most common conditions that cause devices to form groups. We then applied this model to the literature, and found that current context-sharing systems suffer from one or more of the following limitations:

1. An inability to share context across different applications.
2. Lack of support for dynamic groups.
3. Over-reliance on a single, well-known, communications channel.
4. Emphasis on long lasting groups.
5. Inability to support the full range of group interactions (as defined by our conceptual model).

In **CHAPTER 3**, we showed how the insights obtained from our literature review influenced GCF's early design. Specifically, we showed that we could address all of the limitations described above through the following high level requirements:

1. **Provide standardized mechanisms for requesting and receiving context**
   1.1. Define a standardized communications protocol to request and receive context
   1.2. Provide software abstractions to make context easier to package, reuse and share
   1.3. Allow devices to collaborate with each other, regardless of if they are performing the same task or require the same information
2. **Support multiple communication technologies**
   2.1. Support a wide range of communication technologies
   2.2. Let devices use multiple communication technologies simultaneously
3. **Allow devices to automatically form and maintain groups**
   3.1. Allow devices to form groups on the user's behalf
   3.2. Allow devices to dynamically update group membership over time

We then presented GCF's architecture, and showed how it satisfies each of the above requirements. Afterwards, we described two proof-of-concept applications (GroupMap and GroupHike) that were built using the framework. Individually, these applications showed how GCF lets developers easily request and receive context and receive from other devices. When used together, however, these applications also demonstrated how the framework supports grouping across all four quadrants of our conceptual model, and lets devices work together regardless if they are performing the same task or need the same information.

In **CHAPTER 4,** we described how we used GCF to create a variety of context-aware systems. We showed how the framework could be used to 1) detect when users and/or devices are experiencing the same contextual state (Didja), 2) interact with specific appliances in new or unfamiliar environments (Snap-To-It), 3) push relevant applications to users' devices at the exact moment they are needed (Impromptu), and 4) allow users to temporarily share information with their immediate surroundings (Bluewave). In addition to showcasing GCF's versatility, these applications also gave us the opportunity to evaluate GCF's ease of use and extensibility under realistic conditions. This let us discover common use cases that our framework was unable to support, and identify areas for improvement.

In **CHAPTER 5**, we used our experiences building Didja, Snap-To-It, Impromptu, and Bluewave to refine GCF's functional requirements (refer to Appendix B for a full listing). We summarized the modifications made to our framework in order to address the issues we encountered in the previous chapter, and presented a generalizable design process that highlights the important design considerations developers need to make when incorporating GCF into their applications. As a final validation of GCF's robustness, we had 20 developers from academia and industry brainstorm possible applications that they would like to build with the framework in **CHAPTER 6**. We then showed how GCF could support all but one of them, and how its ability to share context across applications increases the chances of applications getting the information they need at runtime.

Finally, in **CHAPTER 7**, we identified the challenges associated with using a framework like GCF in the real world. First, we focused on the developer level, and showed how the risk of platform fragmentation, as well the lack of dedicated debugging tools and trust mechanisms can prevent developers from being able to easily use GCF in their applications. Next, we looked at the user level, and described the challenges in finding ways to share context that are both secure and respectful of the user's privacy in a way that minimizes the impact to his/her cognitive load. Lastly, at the technical level, we looked at ways to balance GCF's functionality with its increased power and data consumption, and identified the factors that need to be taken into consideration when using the framework to form large groups.

Through this body of work, we have made the following contributions:

1. A conceptual model, based on an analysis of prior literature, which describes the conditions under which users and/or devices form and work in groups.
2. An implementation of the Group Context Framework, which highlights the software abstractions and architecture needed to support all of the group types identified in our conceptual model.
3. A demonstration of the value of opportunistic groups in context-aware computing, through the creation of four major systems and numerous smaller applications.
4. A validation of GCF's robustness, through an examination of 65 ideas submitted by 20 developers.
5. An examination of the challenges associated with utilizing opportunistic groups in context-aware applications, based on our own experiences using GCF, as well as from issues raised by developers from academia and industry.

8.2. Directions for Future Work

Although our research provides an initial understanding of how opportunistic groups can be used in context-aware computing, there are still many aspects of our work that merit further study. In addition to the high level challenges identified in **CHAPTER 7**, we would also like to highlight five additional areas for future work that we believe are especially interesting from a technical and/or research perspective.

8.2.1. One-Time and Event-Based Context Delivery

GCF's context providers are currently designed to collect and share information at a fixed rate (*e.g.,* once per second). Yet while this is effective for sensor data, not all context is well-suited to this type of delivery strategy. Contexts such as the user's identity may never change, and thus only need to be delivered once. Meanwhile, contexts such as the user's activity may change several times per minute, or stay the same for several hours at a time; for these types of context, it is better if context providers transmit a context data message whenever the user changes states (*e.g.,* transitioning from sitting, to standing, to walking) rather than at a fixed interval.

Although GCF does not prevent context from being delivered one-time or event-based delivery, it currently requires developers to implement this functionality on their own, or to come up with creative ways to use the framework (*e.g.,* canceling a request for context immediately after receiving a context data message). As a result, future versions of GCF should come with built-in support for a wider range of context delivery strategies. It should be possible, for example, for applications to specify that they only need to receive context once in their context request message. The framework could then automatically unsubscribe from context providers once this information was received. Similarly, it should be possible for applications to include additional constraints, such as a range of values or threshold, and have the framework only deliver context when the value of the context meets and/or exceeds these values. By supporting a diverse range of delivery options, GCF will make it easier for applications to request and receive context in more natural ways. This will expedite the development process, and allow developers to explore more complex use cases.

8.2.2. *Sharing the Framework Between Multiple Applications*

GCF currently requires each application to have an instance of the framework running either as a background service or within the application's lifecycle. This tight coupling is practical when the total number of GCF-enabled applications installed on a phone is small, but becomes inefficient when multiple GCF instances are requesting context, sharing information, and managing their own network connections at the same time. To address this, future versions of the framework should allow its functionality to be shared with multiple applications. This can be as simple as creating a dedicated GCF app that all users have to have installed on their phone (a strategy used by toolkits such as AWARE [143]), or as complex as creating a new distribution of Android that has the framework "baked in" (similar to work done in ErdOS [110]).

In addition to the increased efficiency that this offers, sharing GCF between multiple applications also creates several interesting research challenges. For example, it is unclear how the framework should handle contradictory requests for the same context (*e.g.,* a request for temperature data in degrees Celsius once every minute, immediately followed by another request for temperature data reported in degrees Fahrenheit once every second). Additionally, it is not immediately apparent what type of security model is needed to let applications freely upload functional code (*e.g.,* context providers, arbiters) to a centralized GCF service or app. At an architectural level, GCF already allows applications on the same device to share information. Thus, the emphasis of this work should not be simply to see if GCF *can* be abstracted to a higher level, but rather to identify the specific problems that can occur when the framework is utilized in this manner.

8.2.3. *Evaluating and Extending GCF's Developer Tools*

GCF's online dashboard is automatically updated by the framework, and provides developers with up-to-date information concerning 1) what types of information are available through the framework, and 2) how this information is structured. This tool was created based on our experiences working with the framework, as well as from informal feedback from developers during our validation of Bluewave (section 4.4.1.3). It is unclear, however, just how helpful the information provided through our website is sufficient or usable in its current form. One possible extension of our work is to conduct a more formal evaluation in order to see how our documentation is actually used by developers. This would allow us to identify major information gaps, and identify ways to effectively collect this information and provide it to them in the future.

A more ambitious extension to our research, however, is to see what other types of tools would be useful to developers. It would be interesting, for instance, to create auto-completion tools (similar to those found in modern IDEs) that can actively suggest contexts to users, and automatically register context providers and/or arbiters to developers as they are needed. These tools could even be combined with live usage data in order to provide developers with further *in situ* assistance. For example, if a developer types "**`gcm.sendRequest(`**" into the IDE, the auto-completion tool could automatically recommend context types and provide a description of the information that she can expect to receive (based on sample messages archived on our website). Likewise, if a developer is trying to extract elements from a context data message's payload, the tool could automatically show her what types of payload values are typically contained within based on the context type.

Another tool that might be useful to developers is a dedicated simulator environment. In this environment, developers would be able to instantiate any number of simulated GCF devices (containing, and have them be placed in a virtual location (*e.g.,* a building or outdoor environment). Developers could then run their applications within this environment and see how it behaves under various conditions. In addition to reducing the need for developers to own multiple physical devices, this tool would allow developers to begin generating unit tests for GCF. This would speed up the development process, and lets developers more thoroughly test their applications without having to physically set up or manually test every possible situation on their own.

By creating these types of tools and releasing them to the developer community, we will obtain further insights as to what types of information developers really need in order to use GCF effectively. This will allow us better understand developers' needs, and make it easier for them to get started using our framework.

8.2.4. Adaptive Privacy Controls

A fourth area that we believe should be addressed in future work is the development of dynamic privacy controls. GCF's current privacy controls were inspired by our work with Bluewave, and let users explicitly specify which applications they would like to share their information with and when. Yet while these features were specifically asked for by users, the decision to bring users back into the loop reduces the opportunities for devices to form groups, as it once again requires users to 1) be aware of the specific groups that they (or their devices) can participate in at any given time, and 2) understand what information and/or services they need. This overdependence on the user prevents the framework from being as opportunistic as it could be, which in turn limits users' abilities to seamlessly take advantage of these groupings as they naturally occur.

For now, we address this problem by allowing users to identify specific contexts that they would like their devices to share at all times. Yet this solution still requires the user to explicitly know what information she wants to share *a priori*. A more interesting approach is to let GCF automatically learn the user's privacy preferences and make future decisions on her behalf. One way to achieve this is to define privacy generic policies that define how the framework should behave when in specific environments. This could be achieved by using the "information spaces" model suggested in [69]. A more dynamic approach is to dynamically learn the user's sharing preferences based on their prior privacy choices. For example, if a user has a tendency to share his name but not his location with unfamiliar devices, the framework could learn this preference and apply it the next time another device requests context from it. Machine learning techniques such as deep learning are potentially one way to learn these complex policies over time. Alternatively, an architecture like the one proposed in [102] could potentially be used to alter the amount and type of information delivered to devices depending on the user's current situation.

Clearly, there is a risk to this approach, as there will be times when the system will make the wrong decision (*i.e.,* deciding to sharing context when the user would have done otherwise). However, prior work in intelligibility [74] shows that users are far more likely to accept mistakes from a context-aware system so long as they can tell *why* the system behaved the way they did. Thus, by balancing automation with occasional user verification, it should be possible to create privacy controls that can be realistically utilized in an opportunistic setting, while still giving users the feeling that their information is being managed in a responsible way.

8.2.5. Generalizing GCF Beyond Context-Sharing

The fifth and final area that we believe should be addressed in future work is in exploring GCF's usefulness beyond context sharing. Although this thesis focuses on sharing information, the ability to detect and form opportunistic groups is applicable to a wide range of application domains. One potentially interesting domain that might benefit from GCF's capabilities is cloudlet computing [101], where mobile devices temporarily borrow resources from nearby servers in order to perform computationally expensive or latency sensitive tasks (*e.g.,* analyzing real-time video [112], providing cached search results [67]). Currently, cloudlet systems assume that the computational resources being accessed are connected to the same local area network (*i.e.,* a Wi-Fi hotspot). However, GCF's ability to discover and form groups creates new opportunities for devices to "forage" for resources at runtime. This could increase the range of environments where cloudlet computing can work, and improve users' access to high-power, low-latency computing.

Our work with Snap-To-It and Impromptu takes an initial step along this research path. In both systems, we show that it is possible to use GCF to transmit user interfaces and send asynchronous remote commands—both of which are use cases that we did not intend on supporting when we designed the framework. In the future, we envision that developers will be able to use context request messages to transmit remote code (similar to the way that we transmitted photographs in Snap-To-It) and receive results in context data messages (similar to the way that we shared user interfaces and applications in Impromptu). Obviously, this is just *one* possible way that GCF could be extended. Nevertheless, our framework already provides the building blocks to form groups under a wide range of situations, and it would be interesting to investigate how this capability can be further utilized.

8.3. Closing Remarks

This thesis has presented a novel framework to allow devices to form opportunistic groups and share context. Our work extends the reach of context-aware computing by allowing devices to form groups and work together without requiring them to know of each other or pair in advance. This makes it practical for devices to participate in quick, one-time exchanges of information and/or services, and allows devices to more naturally support users through the planned and unplanned encounters that occur throughout their lives.

The work presented in this thesis brings us closer to achieving Weiser's vision of calm and peaceful computing. For many, the idea of letting our devices automatically form groups and share context is unsettling, as it suggests a future where users no longer have control over their information and how it is shared. As computing becomes more pervasive, however, it is becoming increasingly difficult to anticipate or be aware of all the ways that devices will need to interact each other in order to provide users with relevant information and/or services. GCF's ability to form opportunistic groups represent one possible way for developers to support a wider range of context-aware interactions without increasing the user's cognitive load. We are not so presumptuous as to assume that our work alone can realize Weiser's vision of making computing as refreshing as taking a walk in the woods. It *is* our hope, however, that this work at least represents a sizable, and hopefully calm step along that path.

References

1. Gregory D. Abowd, Christopher G. Atkeson, Jason Hong, Sue Long, and Mike Pinkerton. 1997. Cyberguide: A mobile context-aware tour guide. *Wireless Networks* 3, 5: 421–433. http://doi.org/10.1023/a:1019194325861

2. Gregory D. Abowd and Elizabeth D. Mynatt. 2000. Charting past, present, and future research in ubiquitous computing. *ACM Transactions on Computer-Human Interaction* 7, 1: 29–58. http://doi.org/10.1145/344949.344988

3. Sungjin Ahn and Daeyoung Kim. 2006. *Wireless Sensor Networks*. Springer Berlin Heidelberg, Berlin, Heidelberg. http://doi.org/10.1007/11669463

4. Amro Al-Akkad, Leonardo Ramirez, Alexander Boden, Dave Randall, and Andreas Zimmermann. 2014. Help Beacons: Design and Evaluation of an Ad-Hoc Lightweight SOS System for Smartphones. *CHI '14*, 1485–1494. http://doi.org/10.1145/2556288.2557002

5. Rafael Ballagas, Michael Rohs, and Jennifer G Sheridan. 2005. Sweep and point and shoot. *CHI '05*, 1200–1203. http://doi.org/10.1145/1056808.1056876

6. Debasis Bandyopadhyay and Jaydip Sen. 2011. Internet of Things: Applications and Challenges in Technology and Standardization. *Wireless Personal Communications* 58, 1: 49–69. http://doi.org/10.1007/s11277-011-0288-5

7. Aaron Beach, Mike Gartrell, Sirisha Akkala, et al. 2008. WhozThat? evolving an ecosystem for context-aware mobile social networks. *IEEE Network* 22, 4: 50–55. http://doi.org/10.1109/MNET.2008.4579771

8. Michael Beigl. 1999. Point & click-interaction in smart environments. *Handheld and Ubiquitous Computing*, 311–313.

9. Victoria Bellotti, Maribeth Back, W Keith Edwards, Rebecca E Grinter, Austin Henderson, and Cristina Lopes. 2002. Making Sense of Sensing Systems: Five Questions for Designers and Researchers. 1: 415–422.

10. Matthias Böhmer, Gernot Bauer, and Antonio Krüger. 2010. Exploring the design space of context-aware recommender systems that suggest mobile applications. *CARS '10*.

11. Elisa G Boix, Andoni L Carreton, Christophe Scholliers, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt. 2011. Flocks: Enabling Dynamic Group Interactions in Mobile Social Networking Applications. *SAC '01*, ACM, 425–432. http://doi.org/10.1145/1982185.1982277

12. Nathalie Bricon-Souf and Conrad R Newman. 2007. Context awareness in health care: A review. *International Journal of Medical Informatics* 76, 1: 2–12.

13. Peter J. Brown. 1998. Triggering information by context. *Personal Technologies* 2, 1: 18–27. http://doi.org/10.1007/BF01581843

14. Matthias Budde, Matthias Berning, Christopher Baumgärtner, et al. 2013. Point & control--interaction in smart environments: you only click twice. *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, 303–306.

15. Andrew T. Campbell, Shane B. Eisenman, Nicholas D. Lane, Emiliano Miluzzo, and Ronald A. Peterson. 2006. People-centric urban sensing. *Proceedings of the 2nd annual international workshop on Wireless internet - WICON '06*, ACM Press, 18–es. http://doi.org/10.1145/1234161.1234179

16. Scott Alan Carter. 2007. Supporting early-stage ubicomp experimentation.

17. Scott Carter, Elizabeth Churchill, Laurent Denoue, Jonathan Helfman, and Les Nelson. 2004. Digital Graffiti: Public Annotation of Multimedia Content. *CHI '04 Extended Abstracts*, ACM, 1207–1210. http://doi.org/10.1145/985921.986025

18. Tsung-Hsiang Chang and Yang Li. 2011. Deep shot: a framework for migrating tasks across devices using mobile phone cameras. *CHI '11*, 2163–2172. http://doi.org/10.1145/1978942.1979257

19. Guanling Chen, Ming Li, and David Kotz. 2008. Data-centric middleware for context-aware pervasive computing. *Pervasive Mobile Computing* 4, 2: 216–253. http://doi.org/10.1016/j.pmcj.2007.10.001

20. Shiva Chetan, Jalal Al-Muhtadi, Roy Campbell, and M. Dennis Mickunas. 2005. Mobile Gaia: A Middleware for Ad-Hoc Pervasive Computing. *CCNC '05*, 223–228. http://doi.org/10.1109/ccnc.2005.1405173

21. William R. Clay and Dongwan Shin. 2009. Secure Device Pairing Using Audio. *International Carnahan Conference on Security Technology*, IEEE, 77–84. http://doi.org/10.1109/CCST.2009.5335562

22. Tathagata Das, Prashanth Mohan, Venkata N. Padmanabhan, Ramachandran Ramjee, and Asankhaya Sharma. 2010. PRISM. *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10*, ACM Press, 63. http://doi.org/10.1145/1814433.1814442

23. Christoffer Davidsson and Simon Moritz. 2011. Utilizing implicit feedback and context to recommend mobile applications from first use. *Proceedings of the 2011 Workshop on Context-awareness in Retrieval and Recommendation - CaRR '11*, ACM Press, 19–22. http://doi.org/10.1145/1961634.1961639

24. Nigel Davies, Adrian Friday, Peter Newman, Sarah Rutlidge, and Oliver Storz. 2009. Using bluetooth device names to support interaction in smart environments. *Mobisys '09*, 151–164. http://doi.org/10.1145/1555816.1555832

25. Fred D Davis. 1985. A technology acceptance model for empirically testing new end-user information systems : theory and results. Retrieved September 4, 2014 from http://dspace.mit.edu/handle/1721.1/15192

26. David Dearman, Richard Guy, and Khai Truong. 2012. Determining the Orientation of Proximate Mobile Devices Using their Back Facing Camera. *CHI '12*, 2231. http://doi.org/10.1145/2207676.2208377

27. Anind K Dey, Gregory D Abowd, and Andrew Wood. 1998. CyberDesk: a framework for providing self-integrating context-aware services. *Knowledge-Based Systems* 11, 1: 3–13. http://doi.org/10.1016/S0950-7051(98)00053-7

28. Anind K Dey, Daniel Salber, Gregory D Abowd, and Masayasu Futakawa. 1999. The conference assistant: Combining context-awareness with wearable computing. *Wearable Computers, 1999. Digest of Papers. The Third International Symposium on*, 21–28.

29. Anind K Dey, Timothy Sohn, Sara Streng, and Justin Kodama. 2006. iCAP: Interactive prototyping of context-aware applications. In *Pervasive Computing*. Springer, 254–271.

30. Anind K. Dey and Gregory D. Abowd. 1999. Towards a better understanding of context and context-awareness. *CHI '00 Workshop paper*, 304–307.

31. Anind K. Dey. 2000. Providing Architectural Support for Building Context Aware Applications. Retrieved from http://www.cc.gatech.edu/fce/ctk/pubs/dey-thesis.pdf

32. Whitfield Diffie and Martin E Hellman. 1976. New directions in cryptography. *Information Theory, IEEE Transactions on* 22, 6: 644–654.

33. James R. Eagan, Michel Beaudouin-Lafon, and Wendy E. Mackay. 2011. Cracking the cocoa nut. *Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST '11*, ACM Press, 225. http://doi.org/10.1145/2047196.2047226

34. Nathan Eagle and Alex Pentland. 2005. Social Serendipity: Mobilizing Social Software. *IEEE Pervasive Computing* 4, 2: 28–34. http://doi.org/10.1109/MPRV.2005.37

35. W. Keith Edwards, Mark W. Newman, Jana Sedivy, and Shahram Izadi. 2002. Challenge: *Proceedings of the 8th annual international conference on Mobile computing and networking - MobiCom '02*, ACM Press, 279. http://doi.org/10.1145/570645.570680

36. Shane B. Eisenman, Emiliano Miluzzo, Nicholas D. Lane, Ronald A. Peterson, Gahng-Seop Ahn, and Andrew T. Campbell. 2009. BikeNet. *ACM Transactions on Sensor Networks* 6, 1: 1–39. http://doi.org/10.1145/1653760.1653766

37. Kevin Francis Eustice. 2008. *Panoply: active middleware for managing ubiquitous computing interactions*. ProQuest.

38. Kevin Eustice, V Ramakrishna, Alison Walker, Matthew Schnaider, Nam Nguyen, and Peter Reiher. 2007. nan0sphere: Location-Driven Fiction for Groups of Users. In *Human-Computer Interaction. HCI Intelligent Multimodal Interaction Environments*, JulieA Jacko (ed.). Springer Berlin Heidelberg, 852–861. http://doi.org/10.1007/978-3-540-73110-8_94

39. Kevin Eustice, V. Ramakrishna, Nam Nguyen, and Peter Reiher. 2008. The Smart Party: A Personalized Location-Aware Multimedia Experience. *CCNC '08*, IEEE, 873–877. http://doi.org/10.1109/ccnc08.2007.204

40. A Ferscha. 2000. Workspace awareness in mobile virtual teams. *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.*, IEEE, 272–277. http://doi.org/10.1109/enabl.2000.883742

41. Alois Ferscha, Clemens Holzmann, and Stefan Oppl. 2004. Context Awareness for Group Interaction Support. *MobiWac '04*, 88–97. http://doi.org/10.1145/1023783.1023801

42. David Franklin and Joshua Flaschbart. 1998. All gadget and no representation makes jack a dull environment. *AAAI 1998 Spring Symposium on Intelligent Environments*, 155–160.

43. Adrian A. de Freitas and Anind K. Dey. 2015. The Group Context Framework: An Extensible Toolkit for Opportunistic Grouping and Collaboration. *CSCW '15*, 1602–1611. http://doi.org/10.1145/2675133.2675205

44. Adrian A. de Freitas and Anind K. Dey. 2015. Using Multiple Contexts to Detect and Form Opportunistic Groups. *CSCW '15*, 1612–1621. http://doi.org/10.1145/2675133.2675213

45. Adrian A. de Freitas, Michael Nebeling, Xiang "Anthony" Chen, Junrui Yang, Akshaye Shreenithi Kirupa Karthikeyan Ranithangam, and Anind K. Dey. 2016. Snap-To-It: A User-Inspired Platform for Opportunistic Device Interactions. *CHI '16*. http://doi.org/http://dx.doi.org/10.1145/2858036.2858177

46. Adrian A. de Freitas, Michael Nebeling, and Anind K. Dey. 2016. Bluewave: Enabling Opportunistic Context Sharing via Bluetooth Device Names. *EICS '16*.

47. Fosca Giannotti and Yücel Saygin. 2010. Privacy and security in ubiquitous knowledge discovery. In *Ubiquitous knowledge discovery*. Springer, 75–89.

48. Andrea Girardello and Florian Michahelles. 2010. AppAware: Which mobile applications are hot? *MobileHCI '10*, 431–434. http://doi.org/10.1145/1851600.1851698

49. Saul Greenberg, Michael Boyle, and Jason Laberge. 1999. PDAs and shared public displays: Making personal information public, and public information personal. *Personal Technologies* 3, 1-2: 54–64. http://doi.org/10.1007/BF01305320

50. Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. 2004. An ontology-based context model in intelligent environments. *Proceedings of communication networks and distributed systems modeling and simulation conference*, 270–275.

51. Ramanthan Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. 2004. Propagation of trust and distrust. *Proceedings of the 13th international conference on World Wide Web*, 403–412.

52. Bin Guo, Zhu Wang, Zhiwen Yu, et al. 2015. Mobile Crowd Sensing and Computing. *ACM Computing Surveys* 48, 1: 1–31. http://doi.org/10.1145/2794400

53. Peter Hamilton and Daniel J. Wigdor. 2014. Conductor. *CHI '14*, 2773–2782. http://doi.org/10.1145/2556288.2557170

54. Todd D. Hodes, Randy H. Katz, Edouard Servan-Schreiber, and Lawrence Rowe. 1997. Composable ad-hoc mobile services for universal interaction. *MobiCom '97*, 1–12. http://doi.org/10.1145/262116.262121

55. LarsErik Holmquist, Friedemann Mattern, Bernt Schiele, Petteri Alahuhta, Michael Beigl, and Hans-W Gellersen. 2001. Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artefacts. In *Ubicomp '01*. 116–122. http://doi.org/10.1007/3-540-45427-6_10

56. Chung-Pyo Hong, Cheong-Ghil Kim, and Shin-Dug Kim. 2013. An Effective Personalized Service Provision Scheme Based on Virtual Space for Ubiquitous Computing Environment. *Wireless Personal Communications*: 1–14. http://doi.org/10.1007/s11277-013-1236-3

57. Elaine M Huang and Elizabeth D Mynatt. 2003. Semi-public displays for small, co-located groups. *CHI '03*, 49–56. http://doi.org/10.1145/642611.642622

58. Yun Huang, Anthony Tomasic, Yufei An, Charles Garrod, and Aaron Steinfeld. 2013. Energy Efficient and Accuracy Aware (E2A2) Location Services via Crowdsourcing. *WiMob '13*, 436–443. http://doi.org/10.1109/WiMOB.2013.6673396

59. Richard Hull, Philip Neaves, and James Bedford-Roberts. 1997. Towards situated computing. *International Symposium on Wearable Computers*, 146–153. http://doi.org/10.1109/ISWC.1997.629931

60. Mitsuru Ikeda, Shogo Go, and Riichiro Mizoguchi. 1997. Opportunistic group formation. *AIED '97*, 167–174.

61. Ellen Isaacs, Alan Walendowski, and Dipti Ranganthan. 2002. Hubbub: A Sound-Enhanced Mobile Instant Messenger that Supports Awareness and Opportunistic Interactions. *CHI '02*, 179–186. http://doi.org/10.1145/503376.503409

62. Sandeep Kamath and Joakim Lindh. 2010. Measuring bluetooth low energy power consumption. *Texas instruments application note AN092, Dallas*.

63. Aman Kansal, Suman Nath, Jie Leu, and Feng Zhao. 2007. SenseWeb: An Infrastructure for Shared Sensing. *MultiMedia, IEEE* 14, 4: 8–13. http://doi.org/10.1109/mmul.2007.82

64. Matthew Keally, Gang Zhou, Guoliang Xing, and Jianxin Wu. 2013. Remora: Sensing Resource Sharing Among Smartphone-Based Body Sensor Networks. *IWQoS '13*, 1–10. http://doi.org/10.1109/IWQoS.2013.6550261

65. Tim Kindberg and John Barton. 2001. A Web-based nomadic computing system. *Computer Networks* 35, 4: 443–456. http://doi.org/10.1016/S1389-1286(00)00181-X

66. Scott Robert Klemmer. 2004. Tangible user interface input: Tools and techniques.

67. Emmanouil Koukoumidis, Dimitrios Lymberopoulos, Karin Strauss, et al. 2011. Pocket cloudlets. *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '11*, ACM Press, 171. http://doi.org/10.1145/1950365.1950387

68. Joanna Kulik, Wendi Heinzelman, and Hari Balakrishnan. 2002. Negotiation-Based Protocols for Disseminating Information in Wireless Sensor Networks. *Wireless Networks* 8, 2-3: 169–185. http://doi.org/10.1023/a%253a1013715909417

69. J.A. Landay. 2002. Modeling privacy control in context-aware systems. *IEEE Pervasive Computing* 1, 3: 59–63. http://doi.org/10.1109/MPRV.2002.1037723

70. Marc Langheinrich. 2001. Privacy by Design — Principles of Privacy-Aware Ubiquitous Systems. In *Ubicomp '01*. 273–291. http://doi.org/10.1007/3-540-45427-6_23

71. Youngki Lee, Younghyun Ju, Chulhong Min, Seungwoo Kang, Inseok Hwang, and Junehwa Song. 2012. CoMon: Cooperative Ambience Monitoring Platform with Continuity and Benefit Awareness. *MobiSys '12*, 43–56. http://doi.org/10.1145/2307636.2307641

72. Jonathan Lester, Blake Hannaford, and Borriello Gaetano. 2004. "Are You with Me?" – Using Accelerometers

to Determine if Two Devices are Carried by the Same Person. *PerCom '04*, 33–50. Retrieved March 12, 2014 from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.75.2328

73. Grace Lewis, Marc Novakouski, and Enrique Sánchez. 2013. A Reference Architecture for Group-Context-Aware Mobile Applications. In *Mobile Computing, Applications, and Services*. 44–63. http://doi.org/10.1007/978-3-642-36632-1_3

74. Brian Y Lim and Anind K Dey. 2013. Evaluating Intelligibility Usage and Usefulness in a Context-Aware Application. In *Human-Computer Interaction. Towards Intelligent and Implicit Interaction*, Masaaki Kurosu (ed.). Springer Berlin Heidelberg, 92–101. http://doi.org/10.1007/978-3-642-39342-6_11

75. Felix Xiaozhu Lin, Daniel Ashbrook, and Sean White. 2011. RhythmLink. *UIST '11*, 263–272. http://doi.org/10.1145/2047196.2047231

76. D.G. Lowe. 1999. Object recognition from local scale-invariant features. *International Conference on Computer Vision*, 1150–1157. http://doi.org/10.1109/ICCV.1999.790410

77. Gabriela Marcu, Hayden Demerson, Chanamon Ratanalert, et al. *The Lilypad System: Designing for Collaborative Reflection*.

78. Friedemann Mattern and Christian Floerkemeier. 2010. From the Internet of Computers to the Internet of Things. *From Active Data Management to Event-Based Systems and More*: 242–259. Retrieved February 12, 2015 from http://dl.acm.org/citation.cfm?id=1985625.1985645

79. R. Mayrhofer and H. Gellersen. 2009. Shake Well Before Use: Intuitive and Secure Pairing of Mobile Devices. *IEEE Transactions on Mobile Computing* 8, 6: 792–806. http://doi.org/10.1109/TMC.2009.51

80. Erika McCallister, Timothy Grance, and Karen A. Scarfone. 2010. *Guide to protecting the confidentiality of Personally Identifiable Information (PII)*. National Institute of Standards & Technology, Gaithersburg, MD. http://doi.org/10.6028/NIST.SP.800-122

81. Joseph F McCarthy and Eric S Meidel. 1999. Active Map: A Visualization Tool for Location Awareness to Support Informal Interactions. *HUC '99*, 158–170. http://doi.org/10.1007/3-540-48157-5_16

82. Will McGrath, Mozziyar Etemadi, Shuvo Roy, and Bjoern Hartmann. 2015. fabryq. *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS '15*, ACM Press, 164–173. http://doi.org/10.1145/2774225.2774835

83. Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. 2012. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks* 10, 7: 1497–1516. http://doi.org/10.1016/j.adhoc.2012.02.016

84. Matei Negulescu and Yang Li. 2013. Open project: a lightweight framework for remote sharing of mobile applications. *UIST '13*, 281–290. http://doi.org/10.1145/2501988.2502030

85. Jeffrey Nichols, Brad A. Myers, Michael Higgins, et al. 2002. Generating remote control interfaces for complex appliances. *UIST '02*, 161–170. http://doi.org/10.1145/571985.572008

86. Tadashi Okoshi, Julian Ramos, Hiroki Nozaki, Jin Nakazawa, Anind K. Dey, and Hideyuki Tokuda. 2015. Attelia: Reducing user's cognitive load due to interruptive notifications on smart phones. *2015 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, IEEE, 96–104. http://doi.org/10.1109/PERCOM.2015.7146515

87. Nadia Pantidi, Stuart Moran, Khaled Bachour, et al. 2014. Field testing a rare species bioacoustic smartphone application: Challenges and future considerations. *2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS)*, IEEE, 376–381. http://doi.org/10.1109/PerComW.2014.6815235

88. D Parker. 2010. Our excessively simplistic information security model and how to fix it. *ISSA Journal*: 12–21.

89. Jason Pascoe. 1998. Adding generic contextual capabilities to wearable computers. *International Symposium on Wearable Computers*, 92–99. http://doi.org/10.1109/ISWC.1998.729534

90. Shwetak N Patel and Gregory D Abowd. 2003. A 2-way laser-assisted selection scheme for handhelds in a physical environment. *UbiComp '03*, 200–207.

91. Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. 2014. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys & Tutorials* 16, 1: 414–454. http://doi.org/10.1109/SURV.2013.042313.00197

92. Hubert Pham, Justin Mazzola Paluska, Rob Miller, and Steve Ward. 2012. Clui: a platform for handles to rich objects. *UIST '12*, 177–188. http://doi.org/10.1145/2380116.2380141

93. Maria Poveda Villalon, Mari Carmen Suárez-Figueroa, Raúl García-Castro, and A. Gómez-Pérez. 2010. A Context Ontology for Mobile Environments.

94. Anand Ranganathan, Roy Campbell, Arathi Ravi, and Anupama Mahajan. 2002. ConChat: A Context-Aware Chat Program. *Pervasive Computing* 1, 3: 51–57. http://doi.org/10.1109/mprv.2002.1037722

95. Jukka Riekki, Ivan Sanchez, and Mikko Pyykkönen. 2008. Universal Remote Control for the Smart World. *Ubiquitous Intelligence and Computing*, 563–577. http://doi.org/10.1007/978-3-540-69293-5

96. Matthias Ringwald. 2002. UbiControl: Providing New and Easy Ways to Interact with Various Consumer Devices. *UbiComp '02*, 81.

97. Ronald L Rivest, Adi Shamir, and Len Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* 21, 2: 120–126.

98. Tom Rodden, Keith Cheverst, Nigel Davies, and Alan Dix. 1998. Exploiting context in HCI design for mobile systems. *Workshop on human computer interaction with mobile devices*, 21–22.

99. Nick S. Ryan, Jason Pascoe, and David R. Morse. 1998. Enhanced reality fieldwork: the context-aware archaeological assistant. *Computer Applications in Archaeology*.

100. Daniel Salber, Anind K. Dey, and Gregory D. Abowd. 1999. The context toolkit. *CHI '99*, 434–441. http://doi.org/10.1145/302979.303126

101. Mahadev Satyanarayanan, P Bahl, R Caceres, and N Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *Pervasive Computing, IEEE* 8, 4: 14–23. http://doi.org/10.1109/mprv.2009.82

102. Florian Schaub, Bastian Konings, Michael Weber, and Frank Kargl. 2012. Towards context adaptive privacy decisions in ubiquitous computing. *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*, IEEE, 407–410. http://doi.org/10.1109/PerComW.2012.6197521

103. Bill Schilit, Norma Adams, and Roy Want. 1994. Context-Aware Computing Applications. *Mobile Computing Systems and Applications*, 85–90. Retrieved January 29, 2014 from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4624429

104. Bill N. Schilit and Marvin M. Theimer. 1994. Disseminating active map information to mobile hosts. *Network, IEEE* 8, 5: 22–32.

105. Dominik Schmidt, David Molyneaux, and Xiang Cao. 2012. PICOntrol: using a handheld projector for direct control of physical devices through visible light. *UIST '12*, ACM Press, 379–388. http://doi.org/10.1145/2380116.2380166

106. Minho Shin, Cory Cornelius, Dan Peebles, Apu Kapadia, David Kotz, and Nikos Triandopoulos. 2011. AnonySense: A system for anonymous opportunistic sensing. *Pervasive and Mobile Computing* 7, 1: 16–30. http://doi.org/10.1016/j.pmcj.2010.04.001

107. João Pedro Sousa and David Garlan. 2002. Aura: An Architectural Framework for User Mobility in Ubiquitous

Computing Environments. In *Software Architecture*. Boston, MA, 29–43. http://doi.org/10.1007/978-0-387-35607-5_2

108. Kimberly Tee, Saul Greenberg, and Carl Gutwin. 2006. Providing Artifact Awareness to a Distributed Group Through Screen Sharing. *CSCW '06*, 99–108. http://doi.org/10.1145/1180875.1180891

109. Arvind Thiagarajan, Lenin Ravindranath, Katrina LaCurts, et al. 2009. VTrack. *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems - SenSys '09*, ACM Press, 85. http://doi.org/10.1145/1644038.1644048

110. Narseo Vallina-Rodriguez and Jon Crowcroft. 2011. ErdOS: Achieving Energy Savings in Mobile OS. *MobiArch '11*, 37–42. http://doi.org/10.1145/1999916.1999926

111. Narseo Vallina-Rodriguez, Christos Efstratiou, Geoffrey Xie, and Jon Crowcroft. 2011. Enabling Opportunistic Resources Sharing on Mobile Operating Systems. *S3 '11*, 29–32. http://doi.org/10.1145/2030686.2030696

112. Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. 2012. Cloudlets. *Proceedings of the third ACM workshop on Mobile cloud computing and services - MCS '12*, ACM Press, 29. http://doi.org/10.1145/2307849.2307858

113. Chuong Cong Vo. 2013. A Framework for a Task-Oriented User Interaction with Smart Environments Using Mobile Devices.

114. Bin Wang, J Bodily, and S K S Gupta. 2004. Supporting Persistent Social Groups in Ubiquitous Computing Environments Using Context-Aware Ephemeral Group Service. *PerCom '04*, 287–296. http://doi.org/10.1109/percom.2004.1276866

115. X.H. Wang, Da Qing Zhang, Tao Gu, and H.K. Pung. 2004. Ontology based context modeling and reasoning using OWL. *PERCOM '04*, 18–22. http://doi.org/10.1109/PERCOMW.2004.1276898

116. Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. 1992. The active badge location system. *ACM Transactions on Information Systems* 10, 1: 91–102. http://doi.org/10.1145/128756.128759

117. Roy Want, Trevor Pering, Gunner Danneels, Muthu Kumar, Murali Sundar, and John Light. 2002. The Personal Server: Changing the Way We Think about Ubiquitous Computing. *UbiComp '02*, 194–209. Retrieved from http://portal.acm.org/citation.cfm?id=741493

118. Andy Ward, Alan Jones, and Andy Hopper. 1997. A new location technique for the active office. *IEEE Personal Communications* 4, 5: 42–47. http://doi.org/10.1109/98.626982

119. Matthias Weing, Amrei Röhlig, Katja Rogers, et al. 2013. P.I.A.N.O. *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication - UbiComp '13 Adjunct*, ACM Press, 75–78. http://doi.org/10.1145/2494091.2494113

120. Mark Weiser. 1991. The Computer for the 21st Century. *Scientific American* 265, 3: 94–104. http://doi.org/10.1038/scientificamerican0991-94

121. Jason Wiese, Patrick G Kelley, Lorrie F Cranor, Laura Dabbish, Jason I Hong, and John Zimmerman. 2011. Are You Close with Me? Are You Nearby?: Investigating Social Groups, Closeness, and Willingness to Share. *Ubicomp '11*, 197–206. http://doi.org/10.1145/2030112.2030140

122. Hanno Wirtz, Jan Rüth, Martin Serror, Jó Ágila Bitsch Link, and Klaus Wehrle. 2014. Opportunistic interaction in the challenged internet of things. *Proceedings of the 9th ACM MobiCom workshop on Challenged networks - CHANTS '14*: 7–12. http://doi.org/10.1145/2645672.2645679

123. Peifeng Yin, Ping Luo, Wang-Chien Lee, and Min Wang. 2013. App recommendation. *Proceedings of the sixth ACM international conference on Web search and data mining - WSDM '13*, ACM Press, 395. http://doi.org/10.1145/2433396.2433446

124. Tile. Retrieved March 1, 2015 from https://www.thetileapp.com/

125. MQTT. Retrieved April 19, 2016 from http://mqtt.org/

126. Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. Retrieved March 1, 2016 from https://www.raspberrypi.org/

127. Recognizing the User's Current Activity | Android Developers. Retrieved March 12, 2014 from http://developer.android.com/training/location/activity-recognition.html

128. Disney StoryLight. Retrieved February 25, 2014 from http://www.usa.philips.com/c-p/719945548/disney

129. Philips Hue. Retrieved February 29, 2016 from http://www2.meethue.com/en-us/

130. Philips Hue API. Retrieved February 29, 2016 from http://www.developers.meethue.com/

131. Barcode Scanner - Android Apps on Google Play. Retrieved September 20, 2015 from https://play.google.com/store/apps/details?id=com.google.zxing.client.android&hl=en

132. ActivityRecognitionApi | Google APIs for Android | Google Developers. Retrieved March 1, 2016 from https://developers.google.com/android/reference/com/google/android/gms/location/ActivityRecognitionApi

133. Enyo JavaScript Application Framework Test. Retrieved March 16, 2015 from http://enyojs.com/

134. Phaser - A fast, fun and free open source HTML5 game framework. Retrieved March 16, 2015 from http://phaser.io/

135. The Physical Web. Retrieved March 16, 2015 from https://google.github.io/physical-web/

136. hOurworld. Retrieved September 21, 2015 from https://www.hourworld.org/

137. Uber. Retrieved September 21, 2015 from https://www.uber.com/

138. Apple - iBeacon for Developers. Retrieved September 18, 2015 from https://developer.apple.com/ibeacon/

139. Project Brillo | Google Developers. Retrieved September 17, 2015 from https://developers.google.com/brillo/?hl=en

140. Schema.org. Retrieved September 17, 2015 from https://schema.org/

141. Dashboards | Android Developers. Retrieved September 18, 2015 from https://developer.android.com/about/dashboards/index.html

142. Building an Android Beacon (Android iBeacon Tutorial Overview) - PubNub. Retrieved September 18, 2015 from http://www.pubnub.com/blog/building-android-beacon-android-ibeacon-tutorial-overview/

143. AWARE | Android Mobile Context Instrumentation Framework. Retrieved March 16, 2015 from http://www.awareframework.com/

144. OAuth Community Site. Retrieved April 19, 2016 from http://oauth.net/

145. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Retrieved April 19, 2016 from https://www.ietf.org/rfc/rfc2459.txt

146. The International PGP Home Page. Retrieved April 20, 2016 from http://www.pgpi.org/

147. 2013. App recommendation using crowd-sourced localized app usage data. Retrieved March 1, 2016 from https://www.google.com/patents/US20130325856

148. 2015. Bump (application) - Wikipedia. Retrieved March 16, 2015 from http://en.wikipedia.org/wiki/Bump_(application)

149. 2015. Airbnb. Retrieved from https://www.airbnb.com/

150. 2016. Google Gson. Retrieved from https://github.com/google/gson

Appendix A: GCF Documentation

The Group Context Framework is available at: http://epiwork.hcii.cs.cmu.edu/~adrian/wordpress/. The website contains:

- An overview of GCF's high level architecture and features
- Java source code and support libraries
- Tutorials and code samples
- A list of relevant publications

Appendix B: GCF Functional Requirements

The following table describes the Group Context Framework's functional requirements. This list was derived from our experiences building, using, and revising the framework in **CHAPTERS 3, 4,** and **5**, respectively, and describes the capabilities needed to recreate all of the systems and applications described in this thesis.

Requirement	Supporting GCF Component(s)	Reference
1. Provide standardized mechanisms for requesting and receiving context.		
1.1. Define a standardized communications protocol that lets devices:	Communication Messages	Table 14
1.1.1. Request a specific type of context (*e.g.*, location, age)	Request Message	3.2.1
1.1.2. Advertise their willingness to provide context	Context Capability Message	
1.1.3. Subscribe to another device's context feed	Context Subscription Message	
1.1.4. Deliver raw sensor data (*e.g.*, accelerometer readings) and/or inferred context (*e.g.*, shopping lists)	Context Data Message	
1.1.5. Transmit asynchronous commands/instructions	Compute Instruction Message	5.1.1
1.1.6. Specify the intended recipient(s) of a message (*e.g.*, a single device, a set of devices, or all devices in range)	Destination Field	5.1.1
1.2. Offer software abstractions to make context easier to package, reuse, and share	Context Providers	3.2.2
1.2.1. Provide application-agnostic modules (*i.e.*, Java classes) that can each collect, store, and deliver a specific type of context		
1.2.2. Give developers the ability to create and deploy their own context collection modules		Figure 2
1.3. Allow devices to collaborate and cooperate with each other regardless of if they are performing the same task or require the same information	Group Context Manager	3.5.3
1.3.1. Form groups and share context across different applications	Context Provider	3.5.3
1.3.2. Form groups and share context in the background	GCF Background Service / Thread	5.1.2
2. Support multiple communication technologies		
2.1. Support two-way communications by allowing devices to transmit messages *via*:	Communication Threads	3.2.1
2.1.1. Network broadcasting (*e.g.*, UDP multicast)		

Requirement	Supporting GCF Component(s)	Reference
2.1.2. Client server architectures (*e.g.*, MQTT)		
2.1.3. Peer-to-peer connections (*e.g.*, TCP sockets)		
2.2. Support one-way communications by allowing devices to broadcast context with each other (regardless if they are on the same local area network or paired)	Bluewave	4.4
2.2.1. Allow devices to share any application level context over short distances (*i.e.*, Bluetooth range)	Bluewave Client Service	4.4.1.1
2.2.2. Allow the framework to broadcast system level information with nearby devices, such as:		Figure 66
2.2.2.1. The context(s) that the device can produce or provide		
2.2.2.2. The network address(es) and port(s) where the device is listening for context requests		
2.3. Allow devices to utilize multiple communications technologies simultaneously	Communications Manager	3.2.1
2.4. Allow devices to form *ad hoc* connections	Group Context Manager	5.1.1
2.5. Give developers the ability to support additional communication technologies as needed	Communications Manager / Thread	3.2.1
3. Allow devices to intelligently form and maintain groups		
3.1. Allow devices to form groups on the user's behalf	Arbiters	3.2.3.1
3.1.1. Allow devices to only form a group with itself	Local Only Arbiter	
3.1.2. Allow devices to form a group with the single "best" device that can provide the requested information	Single Source Arbiter	
3.1.3. Allow devices to form a group with two or more devices	Multi Source Arbiter	
3.1.4. Give users the ability to select which device(s) they would like to group with, when appropriate	Manual Arbiter	
3.2. Allow devices to dynamically update group membership as members arrive and/or leave	Arbiters	
3.3. Give developers the ability to define their own grouping policies	Arbiters	Figure 5
4. Give users explicit control over when and how their context is being shared		
4.1. Notify users when another device is requesting their context	Group Context Manager	Figure 54a
4.2. Allow users to see why an application/device needs their context prior to sharing it		Figure 54b
4.3. Let users see what context(s) they are sharing at any given time		Figure 64b-c
4.4. Let users share context with a specific application/device, or with everyone		Figure 64a

APPENDIX C: DEVELOPER SUBMITTED APPLICATIONS

The following table contains all of the application ideas submitted by developers during our validation study (**CHAPTER 6**). These responses were modified slightly in order to correct obvious spelling or grammatical errors.

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Airport Helper	If your phone could tell the plane that you are going to board, then we can have the application guide you to the gate or list other places around your current location like a Cafe, Restroom, Currency Exchange Center, Baggage Weighing Area, etc. This would prove to be useful when going to really big airports and many people do not visit them frequently.	2	Location, flight number	Relay
Parking Assist	When going a place like the Waterfront where the shops are far apart and parking places are scattered, it would be great if an application could tell us the parking space nearest to our destination or one that is most convenient to our destination when we enter Waterfront.	2	Location	Relay
Place Finder	People can ask about a place to eat or drink or go to and based upon their taste, the reviews of the people having similar taste will be sent to them on their location.	2	Location, preferences	Relay
Route Guidance	When a user enters a new building or a place, he/she has the room number as a reference. So, the guidance to the room will be provided to the user using the room number by the system.	2	Destination	Bluewave Only
Recommended books	An application that offers book recommendations to users by letting them rate for books or authors periodically. It could also recommend books based on user's activity, interests or location.	2	Activity (physical), preferences, location	Relay
Impromptu reminders	A reminder app that not only takes 'time' as a factor, but also includes the factors like 'location' 'activity'. It is an effective context aware app that reminds the users with some kind of signal. An example of it could be, in a traditional reminder app, a user could only set a reminder on 5pm to buy milk. But then if the user had to meet his/her boss suddenly at 5pm, there is no use in reminding about the milk at that time. Instead, when the user walks near a grocery the reminder should appear with a signal. So the factors like location and activity are taken into account to remind the users.	N/A	Time, location, activity	Self Share

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Home Assistant	The assistant is an agent that interacts with visitors at the home door and manages the house owner's schedule. The assistant is activated when a visitor approaches, which is detected by two pressure sensitive mats placed on both side of the house door,and it will adapt its behavior to such contextual information as the identity of the visitor, the house owner's schedule status and busy status, and the owner's willingness to see the current visitor. The visitor's name and his schedule could be collected which could be used for automatic appointment setting.	2	Schedule, Altitude, Identity, Preferences, Location	Bluewave Only
GroupScheduleR	Allows to find an available room on campus to spontaneously meet in groups, e.g., for project meetings, brainstormings, etc.	2	Location	Local Area Request
ICE Breaker	If you meet a new person, it's important to start conversation with some ice breakers. It's useful if your phone can access to the context of the person you meet and find common interest. For example, the system can use stored previous context such as the location information (the places visited), web browsing logs (news articles), social networks, and near-by-persons. The app searches for the common things from the two persons' context logs and recommends both of you a list of "ice breakers"	4	Location, web history, social networks, preferences	Bluewave Only
Escourt	It'll help you to go home safely in the night. It can communicate with streetlights, nearby persons, caps, CCTV, payphone and so on, and actively manipulate your nearby environments to be safe. For example, it can increase the brightness around you and broadcasts contexts related to safety to nearby persons.	1	Location	Bluewave Only
Personal Radar	It's like a radar used in airplane to visualize your around objects. It visualizes nearby objects (persons, bulbs, computer, coffee machine and so on) and their contexts (condition, health, mood, working status, and so on). It helps you to find people, service, items around you and also can improve your experience in personal navigation service.	4	Location, identity, system state	Bluewave Only

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Letter in a Bottle	Using this app, you can leave your message onto any objects. Like the letter in a bottle, it's not predictable who will get the message. When someone near the object with some messages, it will pop up them on his phone's screen. If necessary, you can set some context-based rules on the message to filter out the message receivers. For example, you can leave a message about the important message or your know-how, or cautions only you know, personal message, opportunistic message to the refrigerator or bulb or anything.	2	Identity	Ad Hoc Request
Where is Michael now?	The app would indicate to others whether I'm on campus or not, and if I am on campus, then it would show the last known location.	1	Location, Identity	Local Area Request
SimpleSmartHomeApp	I would like to have one app that I could use to remotely (even when I'm out of the house) control power (e.g. turn Christmas tree lights on/off), heating, lights, and blinds in my house. I'm sure there are some integrated smart home solutions out there, but I believe I could use GCF to build my own solution that is potentially cheaper but still integrated. Assuming that everything can be controlled via WiFi, I would use my old Google Nexus as a beacon at home and the receiver for remote commands sent from my current phone when I'm out and about. The SimpleSmartHomeApp running on the Nexus would be using GCF to connect to appliances in the house via WiFi. I would use my current mobile to send requests to the old phone when I'm not at home and could probably do this using GCF as well.	2	Identity	Relay
Student Attendance App	The app would make it easy for lecturers to gather information on who (aka mobile phone) has been attending a lecture or exercise session. This can be helpful for both lecturers and students for classes and labs where attendance is required.	4	Identity	Bluewave Only
My Conference App	The app would provide participants not only with a schedule of the papers and a map of rooms etc., but also help them keep track of what they have seen (papers, tutorials, demos, etc.) and who they have met at the conference. Another feature would be to plan joint lunches and dinners during the conference and even schedule a follow-up meeting after the conference.	2	Identity, location, schedule	Bluewave Only

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Health Display	In a social public space like a student lounge, people come together, chat, share ideas and work on things together. The space is ideal for focussing on the health of people using the space. It is known that social encouragement is a powerful motivational factor. A public display can leverage this social motivation to encourage people to be more physically active. In the literature, Fish'n'steps showed the public displays can lead to users forming healthy routines. With much more contextual information, it might be possible to create even better interactive public displays. One idea would be to use nearby people's step count and calendar information to encourage people to be active together. So if users A and B are in the space and both haven't met their Physical Activity (PA) goal and both are not busy for the next couple of hours, the display will encourage them to go do an exercise class nearby.	2	Identity, schedule, activity (physical)	Bluewave Only
Connected Shoes for behavior change	In this idea, I imagine that the shoes people wear can talk to other nearby shoes. The shoes are able to communicate to each other their wearers physical activity. Other shoes can in turn encourage the user to be more physically active by reflecting their activity to them. The shoes can be equipped with LEDs or displays in order to visualize or communicate with people.	2	Identity, activity (physical)	Ad Hoc Request
Smarthome Assistant	There are many smart-home assistant devices available in the market today like Amazon Echo or Jibo. I imagine that these devices can be improved considerable if they are allowed to know user's contexts. For this idea, we take the case of knowing where they have been or where they plan to go. Knowing the location/activity history will allow the device to be more sympathetic and humane. For e.g. if the user came back from a long drive, the robot may ask how the user is feeling given how tiresome trip might have been. Knowing the location searches will allow the robot to give additional information to the user, like "don't forget the umbrella" if its raining or keep in mind there is a lot of traffic in the destination.	3	Web history, location	Local Area Request

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Air gate keeper	Currently gate information (in airport) are shown on a large screen but there are so many of them and cannot easily find your information (especially if it's international hub airport). My app works if multiple travelers approach to a screen to find gate information for transfer. The 'air gate keeper' request traveler information to the device of travelers and once after they collect them, they provide gate information in a order of travelers' destination. I.E, suppose 10 travelers are standing in front of the screen and if 7 of them are heading to Pitt, 2 are heading to D.C, and 1 is to New york, the gate information for Pitt will show up in the first row while D.C on second row and Newyork on third. So, the screen interactively show the information depending on need of users.	2	Destination	Bluewave Only
Cafe music DJ	Suppose you are running a cafe and need to select a background music but do not really know which music to play, we can try this system. When a bunch of people (if you are successful owner) are staying in your cafe, and if you could collect music information that they are listening to (or have in their phone), you can provide a background music that best fits (or covers best or in common) to the music lists that users have in their phone. As an extreme example, if your cafe is full of french people, your system can provide french music.	2	Preferences	Ad Hoc Request
Smart Sync	The application allows to collect timestamps from surrounding devices in order to properly synch data from each device.	2	Identity, time	Ad Hoc Request
Shared ESM	The application sends a request to all nearby users on nearby devices to answer a simple question. This can be used for group ESM studies.	4	Location, user response	Relay
OpenOrNot (this question is actually the hardest..)	I'm not sure what's the hours of a place, and I want to know whether there are people moving in the area (which can tell me whether it's open or not)	4	Location, activity (physical)	Relay
TakePhotoForMe	If my phone is dead, I can ask other people to take a photo for me and the app can recognize it's me and automatically send the photo to me.	4	Identity, photos	Ad Hoc Request
EnvironmentTeller	I can know in advance how crowded, how loud, how hot, etc a certain environment is before I go, e.g. a restaurant.	4	Location, audio, temperature	Relay
EngagementChecker	Based on the device using frequency, the app can figure out how engaged people are at a certain event, e.g. classroom, concert, sports game.	4	Activity (phone), location	Local Area Request

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Ad Hoc Accessibility	When people with impairments try to access an inaccessible digital device in the environment, the device can automatically share the functionalities to the user's phone so they can control the devices remotely.	2	Identity, preferences	Relay
PopularitySourcing	The app can tell the user what place of a park is worth spending more time at, and where of a scene to take the best photos based on the history of other people's behavior.	4	Location, activity (physical), photos	Relay
Marketeer	Walking in a mall, and it notifies you of discounts, mall help, etc. when you are in a particular section or looking at something for a certain duration of time	2	Compass, location	Relay
StreetArt	An art installation that changes depending on who is looking at it.	4	Photos, location, identity, web history	Bluewave Only
Restaurant Service	When you enter a restaurant, you get your reservations setup, your ID (if you are 21 or not) gets checked without the need to actually carry an ID.	2	Identity, bluetooth	Bluewave Only
PayWell	When you enter a public transport, it detects where you board from and where you leave from and it gets deducted from your virtual wallet.	2	Identity, payment info	Bluewave Only
BodyMeasure	Measuring your basic bodily things like temperature, weight, height, blood pressure (the things they do every time you go for a checkup).	2	Identity, health vitals	Local Area Request
autonomous driving cars	This app could allow self driving cars to learn about other cars around and their navigation state. This for example could help emergency vehicles navigate faster through busy intersections without causing accidents. Similarly, in cases where navigation sensors like cameras or lidars may be failing the car could communicate this failure to other cars and prevent an accident.	2	Speed, destination, system state, identity	Bluewave Only
smart payments	The user enters a restaurant, a bluetooth beacon located on the table tells the smartphone app that the user is not sitting on this table. GCF shares this contextual information with another GCF payment app that the waitress has access to. Now, before the waitress serves the user, it already could know dietary restrictions of the user, likes and dislikes, etc. This can all be displayed on a GCF app running on a tablet. The user gets a bill which is displayed through GCF, there the user can accept the charges and verify if there are any errors, also it can tip the waitress. At the end the GCF app on the user side tells the GCF on the waitress side that the bill has been accepted and proceeds to charge the users card.	2	Identity, preferences, payment info	Bluewave Only

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
smart grocery shopping	The user walks through a supermarket, as it walks different bluetooth beacons the smartphone GCF app then can infer the products proximity and crosscheck with the users shopping list and alert whenever the user does not stop at the respective shelves.	2	Shopping list	Bluewave Only
smart sensors	The GCF app leverages other people's or devices sensor to save battery and obtain more accurate or simply useful information. For example, a user is in the middle of the bus where the GPS signal is spotty, another user near the window with better GPS signal could share its GPS to other users. Similarly other sensors could use the same principle. This kind of social sensing could be leveraged by Building sensors. For example, imagine a building reporting the inside temperature, humidity, location of known people, etc. This could be useful for someone in a new place like in a hospital looking for the doctors office. GCF could check the calendar, find the doctors room and display a map with information of where the office is.	1	Location	Ad Hoc Request
megaSocial	Imagine a bunch of people just walking around with a GCF app that reveals details of people you may stumble upon everyday but that you have not talked to simply because you don't know them. What if through the shared context feature, a GCF app figures out common interests or activities, the sharing of certain location daily and brings people together that way.	2	Identity, preferences	Bluewave Only
Space Sticky	(Application Scenario) A user has a meeting in Anind's office. He wants to take a note or make a reminder for the next meeting in the office. So he executes a GCF-Enabled memo application, then enter some text. He also add a reminder condition to the memo as a "current location". At that time, the app stores the entered text and its contextual data (e.g., indoor location). Later, when he visits Anind's office, the application will notify the stored memo. Or if he configured the trigger condition as "meet Anind", the application will notify the memo when the app detects Anind's phone.	4	Identity, location	Bluewave Only
Who are in the same room	Determine whether there are another user of this service in the same room by comparing whether the microphone data have include common background sound.	2	Audio	Ad Hoc Request

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Find my meeting?	Meetup, meeting face-2-face with people who you meet online is quite challenging. Well, use the app to tell you if they arrived and stand next to you. Or can be used at any social events, dating app, Meetup app, party app, and more.	2	Location, identity	Bluewave Only
My Food Allergy	Share list of my food allergy to restaurant, air plain, social event that provide food. Restaurant can provide ingredient from their menu that might match allergy list of customers.	2	Identity, preferences	Bluewave Only
Love collecter	The world that full with suffering. Give/revived nice, wonderful, encourage messages from/to people who you walk pass you today. Tell us how many messages you get.	2	Messages	Bluewave Only
Who is the best walker	Share/received steps from people who walk pass you. And compare your rank and theirs.	2	Identity, activity (physical)	Bluewave Only
Let's talk	It's difficult for social anxiety people to go and have conversation in a bar or party. The app shows who available and what topics they like to talk. Send emoji to each other to make conversion starting easier.	2	Identity, preferences	Bluewave Only
Theft tools	Capture every context possible. Using machine learning to identify pattern who is the best candidate for roping.	4	Location, accelerometer, altitude, compass, audio	Relay
Auto Lock/Unlock Computer	The app can unlock/lock your computer if you are going away or close to/from it. Maybe, it can be use with a lab door.	2	Credentials	Bluewave Only
Quick book club	Tell people what book are you reading. Which book that people around you read. Good way to start fun conversion and make new book lover friends.	2	Identity, preferences	Bluewave Only
Where is my cat	Cats love hiding. The app help you to detect your cat collar. It also contain basic information such as its name, owner address, and name of the owner. If your cat lost or found a cat, then you know she is whom.	2	Identity	Bluewave Only
Social Poll	One person can create a poll, and other can response to it. It can be used anywhere (coffee shop, sport events, class room) without the need of internet connection. And the poll can last the entire weeks.	2	Identity, user response	Relay
Air Business card	Imagine, you are in a conference or job fair. Would it be cool to always get your contact from people you meet. The app shows information in business cards or people around them. Users can decide to save some of it or filter only the one they want. (or automatically save everything)	2	Identity	Bluewave Only

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Coffee Shop Auto Connection	Automatically connect to publicly available wifi w/o the need to go through settings and/or ask for passwords.	2	Credentials	Bluewave Only
Simple Photo Sharing	Whenever people take a group photo, they often use more than one camera and still have to harass the people whose camera was used to share the photos. This app would just register everyone who is in a photo and send a copy of the picture to them automatically. The push could be done simply to all the devices in bluetooth range (so the stranger taking the photo might get access to the picture) or use facial recognition/known context about the group to send the photos by another means (so someone who didn't have their phone on them would also get a copy of the picture in their email)	2	Identity	Ad Hoc Request
Car Connection Sharing	Within a car, there are often a limited number of connections to the car's speakers and power. If multiple people are in the car, music and navigation instructions are typically tied to whichever device is connected. If a phone call comes in on that phone, you have to plug in another phone to continue getting directions. It would be nice if, within the car, phone calls could be routed to other phones, gps tracking could be handled by the device that is plugged in (or has the most power), and music queues could be controlled by anyone in the car.	2	Location, activity (phone)	Local Area Request
Which cafeteria is less crowded?	Know which cafeteria is less crowded before you go. This app will ask some users near cafeteria near campus about whether they are crowded around dinner or lunch time. With those data in hands, the app can show the user how crowded it is for each cafeteria so that people can make a wise choice.	2	Location, bluetooth, user response	Relay
Building energy optimizer	Optimize the energy consumption of a building by knowing how many (people/app user) are there in different part of the building at a certain time. By deploying many different bluetooth broadcaster in different part of the building and query people in a certain area within the building's location about whether they can have those bluetooth broadcaster in range, we can tell which part of the building those user are currently located. With those people distribution data in mind, I assume building manager can have a better way to optimize power usage.	4	Bluetooth	Relay

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Cheap tracker	Track your stuff location without the high energy consumption GPS. Stick a crappy phone at the stuff you want to track. The crappy phone can have its GPS turned off. The crappy phone turn itself on occasionally and ask nearby devices if they can share GPS information, and update their GPS information to the server. The user can then track the location of the phone by looking up on the server.	3	Location	Ad Hoc Request
More accurate sensors data	Many sensor reading should have similar result within a certain range. This idea is to use this knowledge improve the accuracy of those sensor data. When a sensor data that are similar within a certain range (e.g. location, environment temperature and volume) is needed, the phone publish its own reading and search if there are other phone sharing the similar data. It then uses the average of all those readings for a more accurate result.	4	Location, audio, temperature, altitude	Ad Hoc Request
More power efficient sensors data	Many sensor reading should have similar result within a certain range. This idea is to use this knowledge to reduce the times that the user uses their own sensors. When a sensor data that are similar within a certain range (e.g. location, environment temperature and volume) is needed, the phone first search if there are other phone sharing the similar data. if no data is found, the phone then uses power up its own sensor to get the data and share it, otherwise, it just use other data instead of wasting more power on similar things.	4	Location, audio, temperature, altitude	Ad Hoc Request
CouponShare	Know if someone in the vicinity has a coupon for the store I'm in or the item I'm interested in and allow people to share coupons.	2	Coupons	Bluewave Only
Shared Activities/Interests	Knowing who, close to me, shares similar interests and allow people to connect (similar to meetup groups, but much more localized and decentralized, increase chance of group forming since people already have point of commonality e.g., work for the same company), potentially even only for a tightly defined duration such as a lunch break	2	Identity, preferences	Bluewave Only
Exercise History	Each piece of equipment in a gym keeps a personalized record of the exercise, number of reps, and weight so that people can easily track their progress and adjust their routines.	2	Identity, activity (physical)	Ad Hoc Request

Application Name	Quick Description	Quadrant	Contexts Used	Implementation Pattern
Bill Sharing	Easy bill sharing, knowing who had food with me and allow us to easily share the check. Many applications allow people to share bills (Venmo, Paypal, etc.), but with larger groups this becomes an exercise of finding the right people who might not even be in my address book. If the application knows who I dined with it can pre-populate the list of attendees.	4	Identity	Bluewave Only
Collaborative Semantic Indoor Localization	Knowing where someone is within a building (office, mall, etc.) is a powerful tool, but also hard to realize. Retrieving semantic labels for locations is crucial, but provides challenges such as data privacy and coverage. Using the phones sensors (esp. WiFi) it is possible to localize someone. By storing the store WiFi fingerprints together with the label at an easily accessible beacon (near the entrance) it is possible to collaboratively label locations. This cuts out the need for an expensive and privacy intrusive server client architecture.	4	Wi-Fi, user response	Relay