Towards Wearable Cognitive Assistance

Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter,
Padmanabhan Pillai[†], and Mahadev Satyanarayanan

December 2013
CMU-CS-13-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Intel Labs

Abstract

We describe the architecture and prototype implementation of an assistive system based on Google Glass devices for users in cognitive decline. It combines the first-person image capture and sensing capabilities of Glass with cloud processing to perform real-time scene interpretation. The system architecture is multi-tiered. It offers tight end-to-end latency bounds on compute-intensive operations, while addressing concerns such as limited battery capacity and limited processing capability of wearable devices. The system gracefully degrades services in the face of network failures and unavailability of distant architectural tiers.

This research was supported by the National Science Foundation (NSF) under grant number IIS-1065336, by an Intel Science and Technology Center grant, by the Department of Defense (DoD) under Contract No. FA8721-05-C-0003 for the operation of the Software Engineering Institute (SEI), and by Bosch, Google, IBM, and Vodafone. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of the funding entities or Carnegie Mellon University.

Keywords: mobile computing, cloud computing, Google Glass, virtual machines, system architecture, cloudlets, face detection, face recognition, object recognition, OCR, activity recognition, motion classification, context awareness, graceful degradation

1 Introduction

Today, over 20 million Americans are affected by some form of cognitive decline that significantly affects their ability to function as independent members of society. This includes people with conditions such as Alzheimer's disease and mild cognitive impairment, survivors of stroke, and people with traumatic brain injury. Cognitive decline can manifest itself in many ways, including the inability to recognize people, locations and objects, loss of short- and long-term memory, and changes in behavior and appearance. The potential cost savings from even modest steps towards addressing this challenge are enormous. It is estimated that just a one-month delay in nursing home admissions in the US could save over $1 billion annually. The potential win is even greater when extrapolated to global scale.

Wearable devices such as Google Glass offer a glimmer of hope to users in cognitive decline. These devices integrate first-person image capture, sensing, processing and communication capabilities in an aesthetically elegant form factor. Through context-aware real-time scene interpretation (including recognition of objects, faces, activities, signage text, and sounds), we can create software that offers helpful guidance for everyday life much as a GPS navigation system helps a driver. Figure 1 presents a hypothetical scenario that is suggestive of the future we hope to create.

An ideal cognitive assistive system should function "in the wild" with sufficient functionality, performance and usability to be valuable at any time and place. It should also be sufficiently flexible to allow easy customization for the unique disabilities of an individual. Striving towards this ideal, we describe the architecture and prototype implementation of a cognitive assistance system based on Google Glass devices. This paper makes the following contributions:

- It presents a muli-tiered mobile system architecture that offers tight end-to-end latency bounds on compute-intensive cognitive assistance operations, while addressing concerns such as limited battery capacity and limited processing capability of wearable devices.
- It shows how the processing-intensive back-end tier of this architecture can support VM-based extensibility for easy customization, without imposing unacceptable latency and processing overheads on high data-rate sensor streams from a wearable device.
- It explores how cognitive assistance services can be gracefully degraded in the face of network failures and unavailability of distant architectural tiers.

2 Background and Related Work

2.1 Assistive Smart Spaces

One of the earliest practical applications of pervasive computing was in the creation of *smart spaces* to assist people. These are located in custom-designed buildings such as the Aware Home at Georgia Institute of Technology [10], the Gator Tech Smart House at the University of Florida [9], and Elite Care's Oatfield Estates in Milwaukie, Oregon [28]. These sensor-rich environments detect and interpret the actions of its occupants and offer helpful guidance when appropriate. They can thus be viewed as first-generation cognitive assistance systems.

Inspired by these early examples, we aim to free cognitive assistance systems from the confines of a purpose-built smart space. Simultaneously, we aim to enrich user experience and assistive value. These goals are unlikely to be achieved by an evolutionary path from first-generation systems. Scaling up a smart space from a building or suite of buildings to an entire neighborhood or an entire city is extremely expensive and time-consuming. Physical infrastructure in public spaces tends to evolve very slowly, over a time scale of decades. Mobile computing technology, on the other hand, advances much faster.

Instead of relying on sensors embedded in smart spaces, we dynamically inject sensing into the environment through computer vision on a wearable computer. The scenario in Figure 1 is a good example

Ron is a young veteran who was wounded in Afghanistan and is slowly recovering from traumatic brain injury. He has suffered a sharp decline in his mental acuity and is often unable to remember the names of friends and relatives. He also frequently forgets to do simple daily tasks. Even modest improvements in his cognitive ability would greatly improve his quality of life, while also reducing the attention demanded from caregivers. Fortunately, a new Glass-based system offers hope. When Ron looks at a person for a few seconds, that person's name is whispered in his ear along with additional cues to guide Ron's greeting and interactions; when he looks at his thirsty houseplant, "water me" is whispered; when he looks at his long-suffering dog, "take me out" is whispered. Ron's magic glasses travel with him, transforming his surroundings into a helpful smart environment.

Figure 1: Hypothetical Scenario: Cognitive Assistance for Traumatic Brain Injury

of this approach. Aided by such a system, commonplace activities such as going to a shopping mall, attending a baseball game, or transacting business downtown should become attainable goals for people in need of modest cognitive assistance.

Using computer vision for sensing has two great advantages. First, it works on totally unmodified environments — smart spaces are not needed. Second, the sensing can be performed from a significant physical distance. The video stream can be augmented with additional sensor streams such as accelerometer readings and GPS readings to infer user context. This enables a user-centric approach to cognitive assistance that travels with the user and is therefore available at all times and places.

2.2 Wearable Cognitive Assistance

The possibility of using wearable devices for deep cognitive assistance (e.g., offering hints for social interaction via real-time scene analysis) was first suggested nearly a decade ago [25, 26]. However, this goal has remained unattainable until now for three reasons. First, the state of the art in many foundational technologies (such as computer vision, sensor-based activity inference, speech recognition, and language translation) is only now approaching the required speed and accuracy. Second, the computing infrastructure for offloading compute-intensive operations from mobile devices was absent. Only now, with the convergence of mobile computing and cloud computing, is this being corrected. Third, suitable wearable hardware was not available. Although head-up displays have been used in military and industrial applications, their unappealing style, bulkiness and poor level of comfort have limited widespread adoption. Only now has aesthetically elegant, product-quality hardware technology of this genre become available. Google Glass is the most well-known example, but others are also being developed. It is the convergence of all three factors at this point in time that brings our goal within reach.

The rest of this paper focuses on Google Glass. This is the wearable device used in our prototype implementation. Today, it is the most widely available wearable device with a first-person video camera and sensors such as an accelerometer, GPS, and compass. Although our experimental results apply specifically to the Explorer version of Glass, our system architecture and design principles are applicable to any similar wearable device. Figure 2 illustrates the main components of a Glass device [14].

3 Design Constraints

The unique demands of cognitive assistance applications place important constraints on system design. We discuss these constraints below, and then present the resulting system architecture in Section 4.

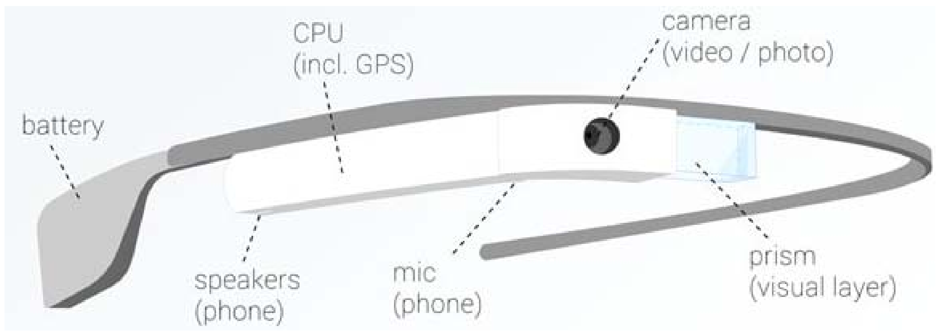


Figure 2: Components of a Google Glass Device (Source: adapted from Missfeldt [14])

3.1 Crisp Interactive Response

Humans are acutely sensitive to delays in the critical path of interaction. This is apparent to anyone who has used a geosynchronous satellite link for a telephone call. The nearly 500 ms round-trip delay is distracting to most users, and leads to frequent conversational errors.

Normal human performance on cognitive tasks is remarkably fast and accurate. Lewis et al. [11] report that even under hostile conditions such as low lighting and deliberately distorted optics, human subjects take less than 700 milliseconds to determine the absence of faces in a scene. For face recognition under normal lighting conditions, experimental results on human subjects by Ramon et al. [22] show that recognition times range from 370 milliseconds for the fastest responses on familiar faces to 620 milliseconds for the slowest response on an unfamiliar face. For speech recognition, Agus et al. [2] report that human subjects recognize short target phrases within 300 to 450 ms, and are able to tell that a sound is a human voice within a mere 4 ms.

Ellis et al [4] report that virtual reality applications that use head-tracked systems require latencies less than 16 ms to achieve perceptual stability. More generally, assistive technology that is introduced into the critical paths of perception and cognition should add negligible delay relative to the task-specific human performance figures cited above. Larger delays will distract and annoy a mobile user who is already attention challenged and in cognitive decline.

3.2 Need for Offloading

Wearable devices are always resource-poor relative to server hardware of comparable vintage [24]. Figure 3, adapted from Flinn [5], illustrates the consistent large gap in the processing power of typical server and mobile device hardware over a 16-year period. This stubborn gap reflects a fundamental reality of user preferences: Moore's Law works differently on wearable hardware. The most sought-after features of a wearable device are light weight, small size, long battery life, comfort and aesthetics, and tolerable heat dissipation. System capabilities such as processor speed, memory size, and storage capacity are only secondary concerns.

The large gap in the processing capabilites of wearable and server hardware directly impacts the user. Figure 4 shows the speed and energy usage of a Glass device for a representative cognitive assistance application (optical character recognition (OCR), described in detail in Section 5.5). In the configuration labeled "Standalone," the entire application runs on the Glass device and involves no network communication. In the configuration labeled "With Offload," the image is transmitted from the Glass device over Wi-Fi 802.11n to a compute server of modest capability: a Dell Optiplex 9010 desktop with an Intel® Core™ i7 4-core processor and 32GB of memory; the result is transmitted back to the Glass device. As Figure 4 shows, offloading gives almost an order of magnitude improvement in both speed of recognition and energy used on the Glass device. The benefit of offloading is not specific to OCR but applies across

	Typical Server		Typical Handheld or Wearable	
Year	Processor	Speed	Device	Speed
1997	Pentium® II	266 MHz	Palm Pilot	16 MHz
2002	Itanium®	1 GHz	Blackberry 5810	133 MHz
2007	Intel® Core™ 2	9.6 GHz (4 cores)	Apple iPhone	412 MHz
2011	Intel® Xeon® X5	32 GHz (2x6 cores)	Samsung Galaxy S2	2.4 GHz (2 cores)
2013	Intel® Xeon® E5	64 GHz (2x12 cores)	Samsung Galaxy S4	6.4 GHz (4 cores)
			Google Glass OMAP 4430	2.4 GHz (2 cores)

Figure 3: Evolution of Hardware Performance (adapted from Flinn [5])

the board to a wide range of cognitive assistance applications.

3.3 Graceful Degradation of Offload Services

What does a user do if a network failure, server failure, power failure, or other disruption makes offload impossible? While such failures will hopefully be rare, they cannot be ignored in an assistive system. It is likely that a user will find himself in situations where offloading to Internet-based infrastructure is temporarily not possible. Falling back on standalone execution on his wearable device will hurt user experience. As the results in Figure 4 suggest, it will negatively impact crispness of interaction and battery life. Under these difficult circumstances, the user may be willing to sacrifice some other attribute of his cognitive assistance service in order to obtain crisp interaction and longer battery life. For example, an assistive system for face recognition may switch to a mode in which it can recognize only a few faces, preselected by the user; when offloading becomes possible again, the system can revert to recognizing its full range of faces. Of course, the user will need to be alerted to these transitions so that his expectations are set appropriately.

This general theme of trading off *application-specific fidelity of execution* for response time and battery life was explored by Noble [18], Flinn [6], Narayanan [16], and others in the context of the Odyssey system. Their approach of creating an interface for applications to query system resource state and to leave behind notification triggers is applicable here. However, assistive applications have very different characteristics from applications such as streaming video and web browsing that were explored by Odyssey.

Metric	Standalone	With Offload
Per-image speed (s)	10.49 (0.23)	1.28 (0.12)
Per-image energy (J)	12.84 (0.36)	1.14 (0.11)

Each result is the mean over five runs of an experiment. Standard deviations are shown in parentheses.

Figure 4: OCR Performance on Glass Device

3.4 Context-sensitive Sensor Control

Significant improvements in battery life and usability are possible if high-level knowledge of user context is used to control the data rate from sensors on a wearable device. For example, consider a user who falls asleep in his chair at home while watching TV. While he is asleep, his Glass device does not have to capture video and stream it for cognitive assistance. In fact, offering whispered hints during his nap might wake him up and annoy him. When he wakes up of his own accord, processing for cognitive assistance should resume promptly. The challenge is, of course, to reliably distinguish between the user's sleeping and waking states. This is the classic problem of activity recognition from body-worn sensor data, on which significant progress has been made in the recent past. These results can be extended to infer context and then use it for adaptive control of sensor streams.

This problem has many subtle aspects. In the above example, if the user falls asleep in a bus or metro rather than in his living room, the cognitive assistance system should give him timely warning of his approaching exit even if it wakes him up from his pleasant nap. In this case, location sensing will need to remain turned on during his nap even though other sensors (such as video) may be turned off. More generally, we see tight coupling of sensing and context inference in a feedback loop as an important capability in a cognitive assistance system.

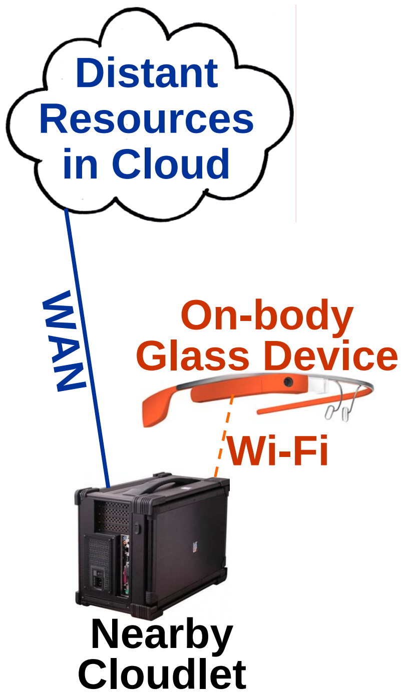
3.5 Coarse-grain Parallelism

Human cognition involves the synthesis of outputs from real-time analytics on multiple sensor stream inputs. A human conversation, for example, involves many diverse inputs: the language content and deep semantics of the words, the tone in which they are spoken, the facial expressions and eye movements with which they are spoken, and the body language and gestures that accompany them. All of these distinct channels of information have to be processed and combined in real time for full situational awareness. There is substantial evidence [21] that human brains achieve this impressive feat of real-time processing by employing completely different neural circuits in parallel and then combining their outputs. Each neural circuit is effectively a processing engine that operates in isolation of others. Coarse-grain parallelism is thus at the heart of human cognition.

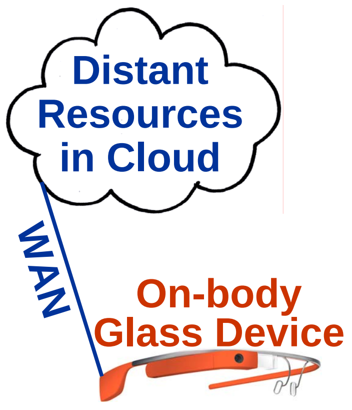
A wide range of software building blocks that correspond to these distinct processing engines exist today: face recognition, activity recognition in video, natural language translation, OCR, question-answering technology such as that used in Watson, and so on. These *cognitive engines* are written in a variety of programming languages and use diverse runtime systems. Some of them are proprietary, some are written for closed source operating systems, and some use proprietary optimizing compilers. Each is a natural unit of coarse-grain parallelism. In their entirety, these cognitive engines represent many hundreds to thousands of person years of effort by experts in each domain. To the extent possible we would like to reuse this large body of existing code. This forces us to avoid automated fine-grain approaches to offloading such as MAUI [3] and COMET [8] that impose significant language and runtime constraints on cognitive engines.

4 Architecture

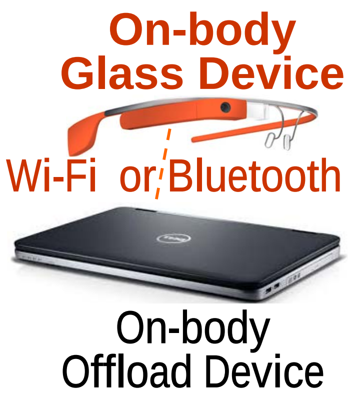
The high-level design of *Gabriel*, our system for wearable cognitive assistance, is strongly influenced by the constraints presented in Section 3. We present an overview of Gabriel here to explain how these constraints are met, and follow with details of our prototype in Section 5.



(a) Normal



(b) Cloud-Direct



(c) Disconnected

Figure 5: Offload Approaches

4.1 Low-latency Offloading

Gabriel faces the difficult problem of simultaneously satisfying the need for crisp, low-latency interaction (Section 3.1) and the need for offloading processing from a wearable device (Section 3.2). The obvious solution of using commercial cloud services over a WAN is unsatisfactory because the RTT is too long. Li et al. [12] report that average RTT from 260 global vantage points to their optimal Amazon EC2 instances is nearly 74 ms, and a wireless first hop would add to this amount. This makes it virtually impossible to meet tight latency goals of a few tens of milliseconds, even if cloud processing takes zero time. There is little evidence that WAN RTTs will improve significantly in the future, since most networking deployments today are working towards improving bandwidth (e.g., for video streaming) rather than lowering latency. Recent work on active control of end-to-end latency shrinks cloud processing time per operation to meet deadlines, and achieves error bounds on the order of 50 ms in an end-to-end latency budget of 1200 ms [23]. Unfortunately, this error bound is comparable to our entire end-to-end latency budget in Gabriel. Further, we strongly believe that degrading cloud processing quality to meet latency bounds should only be a last resort rather than standard operating procedure in cognitive assistance.

Gabriel achieves low-latency offload by using *cloudlets* [26]. A cloudlet is a new architectural element that represents the middle tier of a 3-tier hierarchy: mobile device — cloudlet — cloud. It can be viewed as a "data center in a box" whose goal is to "bring the cloud closer." As a powerful, well-connected and trustworthy cloud proxy that is just one Wi-Fi hop away, a cloudlet is the ideal offload site for cognitive assistance. Although widespread deployment of cloudlets is not yet a reality, commercial interest in cloudlet-like infrastructure is starting. In early 2013, for example, IBM and Nokia-Siemens Networks announced the availability of a "mobile edge computing platform" [1]. Wearable cognitive assistance can be viewed as a "killer app" that has the potential to stimulate investment in cloudlets.

Figure 5(a) illustrates how offload works normally in Gabriel. The user's Glass device discovers and associates with a nearby cloudlet, and then uses it for offload. Optionally, the cloudlet may reach out to the cloud for various services such as centralized error reporting and usage logging. All such cloudlet-cloud interactions are outside the critical latency-sensitive path of device-cloudlet interactions. When the mobile user is about to depart from the proximity of this cloudlet, a mechanism analogous to Wi-Fi handoff is invoked. This seamlessly associates the user with another cloudlet for the next phase of his travels.

4.2 Offload Fallback Strategy

When no suitable cloudlet is available, the obvious fallback is to offload directly to the cloud as shown in Figure 5(b). This incurs the WAN latency and bandwidth issues that were avoided with cloudlets. Since RTT and bandwidth are the issues rather than processing capacity, application-specific reduction of fidelity must aim for less frequent synchronous use of the cloud. For example, a vision-based indoor navigation application can increase its use of dead reckoning and reduce how often it uses cloud-based scene recognition. This may hurt accuracy, but the timeliness of guidance can be preserved. When a suitable cloudlet becomes available, normal offloading can be resumed. To correctly set user expectations, the system can use audible or visual signals to indicate fidelity transitions. Some hysteresis will be needed to ensure that the transitions do not occur too frequently.

An even more aggressive fallback approach is needed when the Internet is inaccessible (Section 3.3). Relying solely on the wearable device is not a viable option, as shown by the results of Figure 4. To handle these extreme situations, we assume that the user is willing to carry a device such as a laptop or a netbook that can serve as an offload device. As smartphones evolve and become more powerful, they too may become viable offload devices. The preferred network connectivity is Wi-Fi with the fallback device operating in AP mode, since it offers good bandwidth without requiring any infrastructure. For fallback devices that do not support AP mode Wi-Fi, a lower-bandwidth alternative is Bluetooth. Figure 5(c) illustrates offloading while disconnected.

How large and heavy a device to carry as fallback is a matter of user preference. A larger and heavier fallback device can support applications at higher fidelity and thus provide a better user experience. With a higher-capacity battery, it is also likely to be able to support a longer period of disconnected operation than a smaller and lighter device. However, running applications at higher fidelity shortens battery life. One can view the inconvenience of carrying an offload device as an insurance premium. A user who is confident that Internet access will always be available during his travels, or who is willing to tolerate temporary loss of cognitive assistance services, can choose not to carry a fallback offload device.

4.3 VM Ensemble and PubSub Backbone

For reasons explained earlier in Section 3.5, a cloudlet must exploit coarse-grain parallelism across many off-the-shelf cognitive engines of diverse types and constructions. To meet this requirement, Gabriel encapsulates each cognitive engine (complete with its operating system, dynamically linked libraries, supporting tool chains and applications, configuration files and data sets) in its own virtual machine (VM). Since there is no shared state across VMs, coarse-grain parallelism across cognitive engines is trivial to exploit. A cloudlet can be scaled out by simply adding more independent processing units, leading eventually to an internally-networked cluster structure. If supported by a cognitive engine, process-level and thread-level parallelism within a VM can be exploited through multiple cores on a processor — enhancing parallelism at this level will require scaling up the number of cores. A VM-based approach is less restrictive and more general than language-based virtualization approaches that require applications to be written in a specific language such as Java or C#. The specific cognitive engines used in our prototype (discussed in Section 5) span both Windows and Linux environments.

Figure 6 illustrates Gabriel's backend processing structure on a cloudlet. An ensemble of *cognitive*

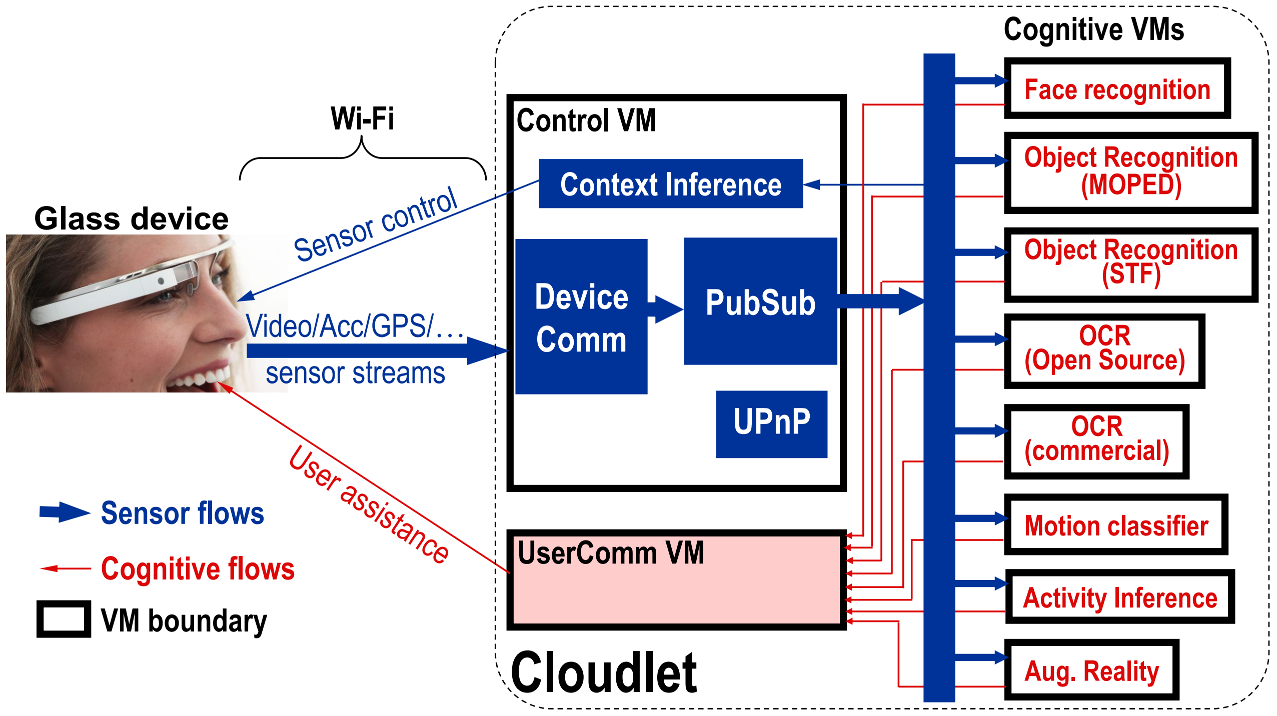


Figure 6: Backend Processing on Cloudlet

VMs, each encapsulating a different cognitive engine, independently processes the incoming flow of sensor data from a Glass device. A single *control VM* is responsible for all interactions with the Glass device. The sensor streams sent by the device are received and preprocessed by this VM. For example, the decoding of compressed images to raw frames is performed by a process in the control VM. This avoids duplicate decoding within each cognitive VM. A PubSub mechanism distributes sensor streams to cognitive VMs. At startup, each VM discovers the sensor streams of interest through a UPnP discovery mechanism in the control VM.

The outputs of the cognitive VMs are sent to a single *UserComm VM* that integrates these outputs and performs higher-level cognitive processing. In this initial implementation of Gabriel, we use very simple rule-based software. As Gabriel evolves, we envision significant improvement in user experience to come from more sophisticated, higher-level cognitive processing in the UserComm VM. From time to time, the processing in the UserComm VM triggers output for user assistance. For example, a synthesized voice may say the name of a person whose face appears in the Glass device's camera. It may also convey additional guidance for how the user should respond, such as "John Smith is trying to say hello to you. Shake his hand."

Section 3.4 discussed the importance of context-sensitive control of sensors on the Glass device. This is achieved in Gabriel through a context inference module in the Control VM. In practice, this module determines context from the external cognitive engines and uses this to control sensors. Referring to the example in Section 3.4, it is this module that detects that a user has fallen asleep and sends a control message to the Glass device to turn off the camera.

5 Prototype Implementation

We have built a prototype of Gabriel around Google Glass and cloudlets, following closely the architecture outlined earlier. Our system uses Google Glass Explorer version based on Android 4.0.4 and a front-end Glass application written using Android SDK.

The cloudlet itself is composed of 4 desktop machines, each with an Intel® Core® i7-3770 and 32 GB memory, running Ubuntu 12.04.3 LTS server. The control VM runs a UPnP server to allow the Glass device to discover it, and to allow the various VMs to discover the control VM and PubSub system. We use OpenStack Grizzly [20] on top of QEMU/KVM version 1.0 as the base software stack of the cloudlet.

5.1 Glass Front-end

A front-end Android application that runs on the Glass device discovers a cloudlet and connects to its Gabriel back-end. Since our implementation was done before the Glass Develpment Kit (GDK) became available, it does not take advantage of some Glass-specific features. Our next version will be GDK-based. After connecting to the Gabriel back-end, the front-end streams video, GPS coordinates, and accelerometer readings over Wi-Fi to the cognitive engines. Each sensor stream uses its own TCP connection. Guidance from UserComm, based on the outputs of cognitive engines, is presented to the user via text or images on the Glass display or as synthesized speech using the Android text-to-speech API. A single Wi-Fi TCP connection is used for this guidance. In the prototype implementation, the output passes through the DeviceComm module of the Control VM since it represents the single point of contact for all interactions with the Glass device.

5.2 Discovery and Initialization

The architecture of Gabriel is based on many software components in multiple VMs working together. Key to making this composition work is a mechanism for the different components to discover and connect with each other.

In our prototype, the control VM is launched first and provides the servers to which Google Glass and the cognitive engine VMs connect. The UserComm VM starts next, followed by the cognitive VMs, which are connected together on a private virtual network using OpenStack Nova-networking's VLAN. This private network can span machine boundaries, allowing sizable cloudlets to be used. The control VM is connected to both this private network and the regular LAN, with a public IP address. The latter is used to communicate with the Google Glass device.

A UPnP server, which provides a standard, broadcast-based method of discovering local services in a LAN, runs in the control VM. This allows the other VMs to find the control VM when they start. More specifically, at launch, each VM performs a UPnP query on the private network. The UPnP server, which is tied to our simplified PubSub system, replies with a list of published data streams that are available and the PubSub control channel. The cognitive VMs subscribe to the desired sensor streams and register their own processed data streams through the PubSub system. At a lower level, this establishes TCP connections between the cognitive VMs and the sensor stream servers in the control VM, and to the UserComm VM that subscribes to all of the processed data streams.

The UPnP server also provides a simple mechanism to let the Glass device discover the Gabriel infrastructure on a LAN. On the public interface of the control VM, the UPnP server replies to queries with the public IP address and port of the Device Comm server. The Glass front end makes multiple TCP connections to this to push its sensors streams and receive cognitive assistance output (passed through from the UserComm VM) as well as sensor control information.

When the Glass device is not within the broadcast domain of a cloudlet running Gabriel, the system

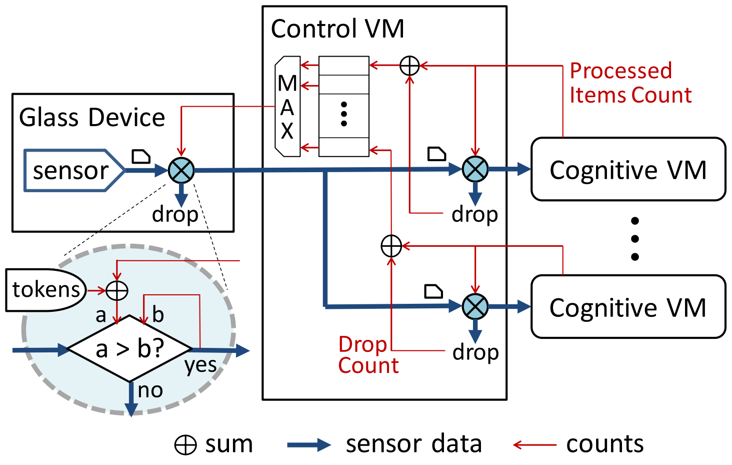


Figure 7: Two level token-based filtering scheme

fails over to a cloud-hosted directory of nearby cloudlets, or to offload to an instance of Gabriel in the cloud. When all else fails, offload to other user-carried devices can be performed, as described in Section 4.2.

5.3 Handling Cognitive Engine Diversity

Our architecture allows cognitive engines to use a variety of programming frameworks and operating systems. For all of these components to work together, we do need them to follow a common communication paradigm. To this end, our system requires a small amount of glue logic to be written for each cognitive engine. This code can be thought of as a wrapper around the cognitive engine, transforming the inputs and outputs to work with our system. It pulls sensor data from the streams and presents it in a manner needed by the cognitive engine (e.g., in the right image format, or as a file). It also extracts results from the engine-specific form and transmits it as a processed data stream. This wrapper is also responsible for discovering and connecting to the PubSub system, subscribing to the required sensor streams, and publishing the outputs. Finally, the wrapper ensures that the cognitive engine processes each data item (e.g., by signaling a continuously running engine that new data is available, or relaunching the engine for each item). All of these functions are simple to implement, and tend to follow straightforward patterns. However, as they are tailored to specific cognitive engines, we expect the wrappers to be packaged with the engines in the cognitive VMs.

5.4 Limiting Queuing Latency

A ramification of our flexible, pluggable architecture is that it results in a set of components that communicate using standard networking protocols (i.e., TCP). Each communication hop involves a traversal of the networking stacks, and can involve several queues in the applications and guest OSs, over which we have little control. The application and TCP buffers can be large, and cause many items to be queued up, increasing latency. To minimize queuing, we need to ensure that the data ingress rate never exceeds the bottleneck throughput. However, the actual bottleneck and throughput can vary dramatically over time – the available network bandwidth can fluctuate between the Glass device and the control VM, and processing times can vary based on data content.

We devised an application-level, end-to-end flow control system to limit the total number of data items in flight at a given time. We use a token-bucket filter to limit ingress of items for each data stream at the Glass device, using returned counts of completed items exiting the system to replenish tokens.

Engine	Source	OS	VCPU*	Sensors
Face Recognition	open source research code based on OpenCV [19]	Windows	2	images
Object Detection (MOPED)	based on published MOPED [15] code	Linux (32-bit)	4	images
Object Detection (STF)	reimplementation of STF [27] algorithm	Linux	4	images
OCR (Open Source)	open source tesseract-ocr [7] package	Linux	2	images
OCR (Commercial)	closed-source VeryPDF OCR product [31]	Windows	2	images
Motion Classifier	action detection based on research MoSIFT [33] code	Linux	12	video
Activity Inference	reimplementation of algorithm from [17]	Linux	1	accelerometer
Augmented Reality	research code from [29]	Windows	2	images

* Number of virtual CPUs allocated to the cognitive engine at experiment

Figure 8: Summary of implemented cognitive engines in our prototype

This provides a strong guarantee on the number of data items in the processing pipeline, limits any queuing, and automatically adjusts ingress data rate (frame rates) as network bandwidth or processing times change.

To handle multiple cognitive engines with different processing throughputs, we add a second level of filtering at each cognitive VM (Figure 7). This achieves per-engine rate adaptation while minimizing queuing latency. Counts of the items completed or dropped at each engine are reported to and stored in the control VM. The maximum of these values are fed back to the source filter, so it can allow in items as fast as the fastest cognitive engine, while limiting queued items at the slower ones. The number of tokens corresponds to the number of items in flight. A small token count minimizes latency at the expense of throughput and resource utilization, while larger counts sacrifice latency for throughput. A future implementation may adapt the number of tokens as a function of measured throughput and latency, ensuring maximum performance as conditions change.

5.5 Supported Cognitive Engines

Our prototype incorporates several cognitive engines based on available research and commercial software. These are summarized in Figure 8 and detailed below.

Face Recognition: A most basic cognitive task is the recognition of human faces. Our face recognition engine runs on Windows. It uses a Haar Cascade of classifiers to perform detection, and then uses the Eigenfaces method [30] based on principal component analysis (PCA) to make an identification from a database of known faces. The implementation is based on OpenCV [19] image processing and computer vision routines.

Object Recognition (MOPED): Our prototype supports two different object recognition engines. They are based on different computer vision algorithms, with different performance and accuracy characteristics. The open source MOPED engine runs on Linux, and makes use of multiple cores. It extracts key visual elements (SIFT features [13]) from an image, matches them against a database, and finally performs geometric computations to determine the pose of the identified object. The database in our prototype is populated with thousands of features extracted from more than 500 images of 13 different objects.

Object Recognition (STF): The other object recognition engine in our prototype is based on machine learning, using the semantic texton forest (STF) algorithm described by Shotton et al [27]. For our prototype, the MSRC21 image dataset mentioned in that work (with 21 classes of common objects) was used as the training dataset. Our Python-based implementation runs on Linux and is single-threaded.

OCR (Open Source): A critical assistance need for users with visual impairments is a way to determine what is written on signs in the environment. Optical character recognition (OCR) on video frames captured by the Glass camera is one way to accomplish this. Our prototype supports two different OCR engines. One of them is the open source Tesseract-OCR package [7], running on Linux.

OCR (Commercial): The second OCR engine supported by our prototype is a Windows-based commercial product: VeryPDF PDF to Text OCR Converter [31]. It provides a command-line interface that allows it to be scripted and readily connected to the rest of our system.

Neither OCR engine is ideal for our purposes. Since they were both intended to operate on scans of printed documents, they do not handle well the wide range of perspective, orientation, and lighting variations found in camera images. They are also not optimized for interactive performance. However, they are still useful as proofs of concept in our prototype.

Motion Classifier: To interpret motion in the surroundings (such as someone is waving to a user or running towards him), we have implemented a cognitive engine based on the MoSIFT algorithm [33]. From pairs of consecutive frames of video, it extracts features that incorporate aspects of appearance and movement. These are clustered to produce histograms that characterize the scene. Classifying the results across a small window of frames, the engine detects if the video fragment contains one of a small number of previously-trained motions, including waving, clapping, squatting, and turning around.

Activity Inference: As discussed in Section 3.4, determining user context is important. Our activity inference engine is based on an algorithm described by Nirjon et al [17] for head-mounted accelerometers. The code first determines a coarse level of activity based on the standard deviation of accelerometer values over a short window. It then uses clues such as inclination to refine this into 5 classes: sit/stand, two variants of lay down, walk, and run/jump.

Augmented Reality: Our prototype supports a Windows-based augmented reality engine that identifies buildings and landmarks in images [29]. It extracts a set of feature descriptors from the image, and matches them to a database that is populated from 1005 labeled images of 200 buildings. The implementation is multi-threaded, and makes significant use of OpenCV libraries and Intel Performance Primitives (IPP) libraries. Labeled images can be displayed on the Glass device, or their labels can be read to the user.

5.6 UserComm VM

Our architecture envisions a sophisticated UserComm process that interprets results from multiple cognitive engines and uses context to intelligently present informative assistance to the user. As the focus of this work is on the architecture and system performance of Gabriel, not cognitive assistance algorithms, our prototype incorporates a rudimentary place holder for a full-fledged UserComm service. Our implementation integrates the outputs of the cognitive engines into a single stream of text, which is then converted to speech output on the Glass device. It filters the text stream to remove annoying duplicate messages over short intervals, but is not context-sensitive. The implementation of a complete, intelligent UserComm service will entail significant future research, and is beyond the scope of this paper.

6 Evaluation

Gabriel is an extensible architecture that provides a plug-in interface for the easy addition of new cognitive engines. These cognitive engines are written by third parties, and are of varying quality. In some cases, these engines are closed source and we have no control except through a few external parameters. Our evaluation in this paper focuses on the intrinsic merits of the Gabriel architecture, rather than the speed, accuracy or other attributes of the cognitive engines or their synthesis into user assistance. Wherever possible, we use real cognitive engines in our evaluation so that the workloads they impose are realistic. In a few cases, we supplement real cognitive engines with synthetic ones to study a particular tradeoff in depth.

We quantify the merits of the Gabriel architecture by asking the following questions:

- How much overhead does the use of VMs impose? Is the complexity of a full-fledged OpenStack cloud computing infrastructure tolerable in the latency-sensitive domain of cognitive assistance? (Section 6.2)
- Are cloudlets really necessary for cognitive assistance? How much benefit do they provide over conventional cloud infrastructure such as Amazon EC2? (Section 6.3)
- Is the implementation complexity of token-based flow control (Section 5.4) necessary? (Section 6.4)
- How easy is it to scale out cognitive engines? Assuming that they are written to exploit parallelism, how well is Gabriel able to provide that parallelism through the use of multiple VMs? (Section 6.5)
- Is Gabriel able to meet latency bounds even when loaded with many cognitive engines processing at full data rate? (Section 6.6)
- Does fallback through fidelity reduction really help in preserving crisp user interaction? What is the tradeoff between loss of fidelity and improvement in response time and battery life? (Section 6.7)

We examine these questions below, in Sections 6.2 to 6.7, after describing the aspects of our setup that are common to all the experiments.

6.1 Experimental Setup

Hardware specifications for our experiments are shown in Figure 9. Experiments using a cloudlet deploy Gabriel inside VMs managed by OpenStack on Linux hosts. For any experiment involving the cloud, we used Amazon EC2 c3.2xlarge VM instances located in Oregon, US. For any experiments involving fallback devices, we run the cognitive engines natively with no virtualization on either a laptop or a netbook. The netbook is comparable in power to a modern smartphone, but it has an x86 processor which means all of our cognitive engines execute unmodified and results are directly comparable with other experiments. The Google Glass device always connects to the cloudlet and cloud via a private Wi-Fi Access Point (AP) which routes over the public university network. For the fallback experiments, we create Wi-Fi hotspots on the offload devices and let the Google Glass directly connect to them.

For reproducibility we use a pre-recorded 360p video to process with the cognitive engines. During experiments, we capture frames from the Glass, but replace them with frames from the pre-recorded video. In this way, our energy measurement will be consistent. We compute energy consumption of Google Glass using the battery voltage and current reported by the OS (in `/sys/class/power_supply/bq27520-0/current_now` and `/sys/class/power_supply/bq27520-0/voltage_now`). We have validated that this measure is consistent based on the battery capacity specifications. For laptops and netbooks, we use a WattsUp Pro

	CPU	RAM
Cloudlet	Intel® Core™ i7-3770 3.4GHz, 4 cores, 8 threads	32GB
Amazon EC2 Oregon	Intel® Xeon® E5-2680v2 2.8GHz, 8 VCPUs	15GB
Laptop - Dell Vostro 1520	Intel® Core™ 2 Duo P8600 2.4GHz, 2 cores	4GB
Netbook - Dell Latitude 2120	Intel® Atom™ N550 1.5GHz 2 cores, 4 threads	2GB
Google Glass	OMAP4430 ARMv7 Processor rev 3 (v7l)	773MB

Figure 9: Experiment hardware specifications

.NET power meter [32] to measure the AC power consumption with batteries removed.

While measuring latency, we noticed anomalous variance during experiments. We found that to reduce discomfort as a wearable device, Glass attempts to minimize heat generation by scaling CPU frequency. Unfortunately, this causes latency to vary wildly as seen in Figure 10. The CPU operates at 300, 600, 800, or 1008 MHz, but uses the higher frequencies only for short bursts of time. Empirically, with ambient room temperature around 70 degrees Fahrenheit, the Glass sustains 600 MHz for a few minutes at a time after, which it drops to its lowest setting of 300 MHz. In Figure 10, we let Glass capture images from its camera, send them over Wi-Fi to a server, and report time from capture to time of server acknowledgment back to the Glass. Whenever the CPU frequency drops we observe a significant increase in latency. To reduce thermal variability as a source of latency variability in our experiments, we externally cool the Glass device by wrapping it with an ice pack in every experiment, allowing it to sustain 600 MHz indefinitely.

6.2 Gabriel Overhead

The Gabriel architecture uses VMs extensively to provide an extensible system with few constraints on the operating systems and cognitive engines used. To quantify the overhead of Gabriel running on a cloudlet, we measure the time delay between receiving sensor data from Google Glass and sending back a result. To isolate performance of the infrastructure, we use a *NULL* cognitive engine, that simply sends a dummy result upon receiving any data. The cognitive VM is run on a separate physical machine from the Control and UserComm VMs. Figure 11 summarizes time spent in Gabriel for 3000 request-response

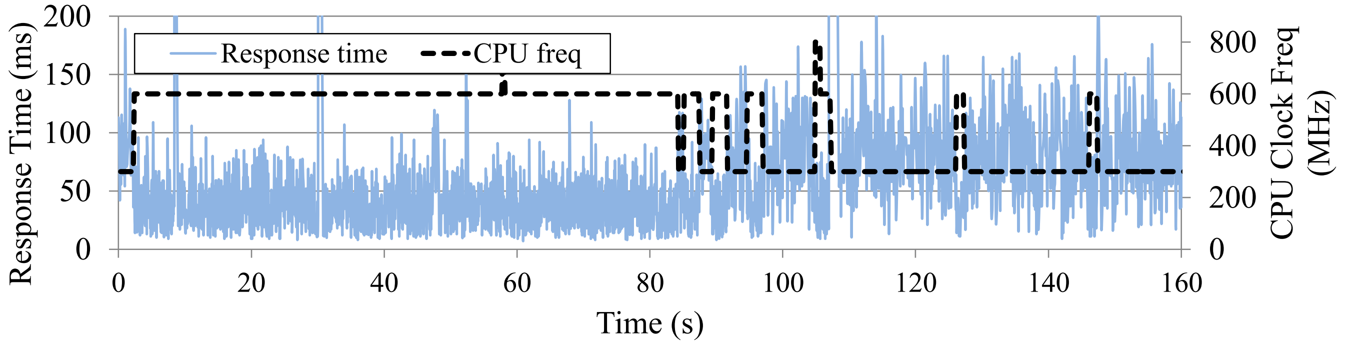


Figure 10: Trace of CPU frequency, response time changes on Google Glass

percentile	1%	10%	50%	90%	99%
delay (ms)	1.8	2.3	3.4	5.1	6.4

The delay between receiving a sensor data and sending a result using *NULL* engine.

Figure 11: Intrinsic delay introduced by Gabriel

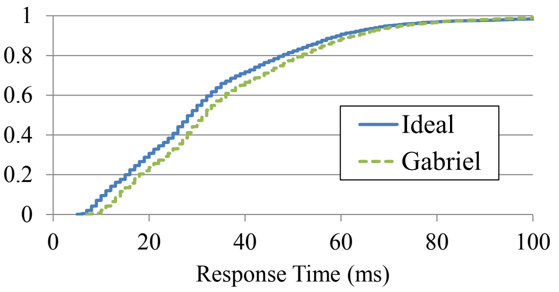


Figure 12: CDF of end-to-end response time for the *NULL* cognitive engine

pairs as measured on the physical machine hosting the Control VM. This represents the overhead of the Gabriel backend framework. At 2–5 ms, this overhead is surprisingly small, considering it includes overheads for the Control, Cognitive, and UserComm VM stacks, and for traversing the cloudlet's internal Ethernet network.

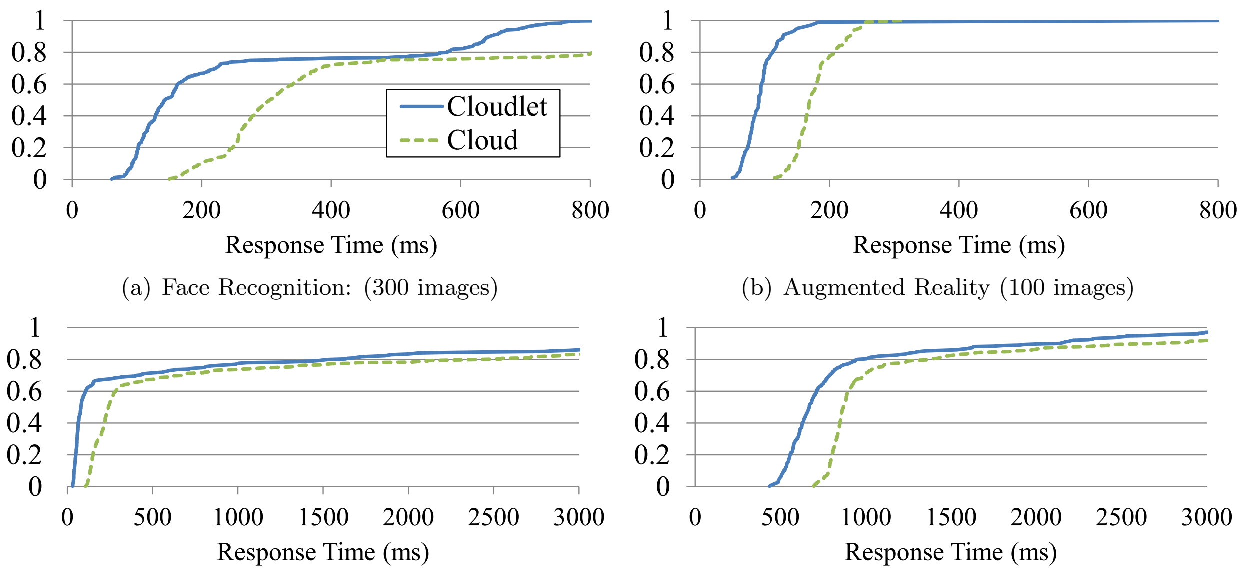
The overall, end-to-end latency measures at Google Glass are shown in Figure 12. These include processing time to capture images on Glass, time to send 6–67 KB images over the Wi-Fi network, the Gabriel backend response times with the NULL engine, and transmission of dummy results back over Wi-Fi. We also compare against an ideal server – essentially the NULL server running natively on the offload machine (no VM or Gabriel framework). Here, Gabriel and the ideal case achieve 33 and 29 ms median response times, respectively, confirming an approximately 3 ms overhead for the flexible Gabriel architecture. These end-to-end latencies represent the minimums that can be achieved with offload of a dummy task. Real cognitive computations will add to these latencies.

6.3 Need for Cloudlets

We compare offload on our cloudlet implementation to the same services running in the cloud (Amazon EC2 Oregon datacenter). Figure 13 shows the distribution of end-to-end latencies as measured at the Glass device for face recognition, augmented reality, and both OCR engines. The results show that we do indeed see a significant decrease in latencies using cloudlets over clouds. The median improvement is between 80 ms and 200 ms depending on the cognitive engine. Much of this difference can be attributed to the higher latencies and transfer times to communicate with the distant data center.

In addition, except for augmented reality, the CDFs are quite heavy tailed. This indicates that the time to process each data item can vary significantly, with a small fraction of inputs requiring many times the median processing time. This will make it difficult to achieve the desired tens of milliseconds responses using these cognitive engines, and indicates more effort on the algorithms and implementations of the cognitive engines is warranted. Although this variation exists for both the cloud and cloudlet, the slopes of the curves indicate that our cloudlet instances are actually faster, despite the fact that we employ desktop-class machines for the cloudlet and use Amazon instances with the highest clock rate and network speed available for the cloud.

Finally, the extra latency for cloud offload will also adversely affect the energy consumed on the Glass device. Figure 14 shows the energy consumed on Glass per query while running these experiments. Overall, cloudlet offload reduces energy per query by 30–40%. Clearly, offload to local cloudlets can produce significant latency and energy improvements.



(a) Face Recognition: (300 images)

(b) Augmented Reality (100 images)

(c) OCR - Open Source (300 images)

(d) OCR - Commercial (300 images)

Figure 13: Cumulative distribution function (CDF) of response times in ms

Application	Cloudlet (Joule/query)	Cloud (Joule/query)
Face	0.48	0.82
AR	0.19	0.32
OCR(open)	2.01	3.09
OCR(comm)	1.77	2.41

Figure 14: Energy consumption on Google Glass

6.4 Queuing Delay Mitigation

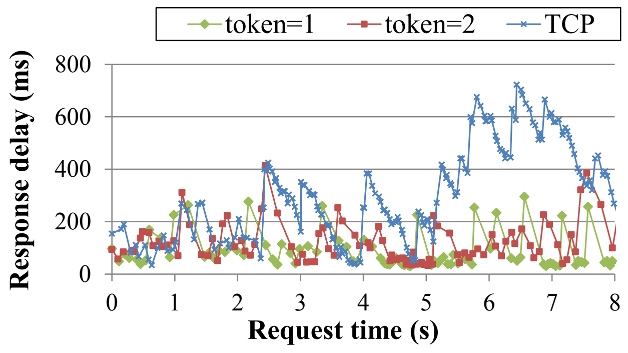
In order to evaluate the need for our application-level, end-to-end flow-control mechanism for mitigating queuing, we first consider a baseline case where no explicit flow control is done. Rather, we push data in a open-loop manner, and rely on average processing throughput to be faster than the ingress data rate to keep queuing in check. Unfortunately, the processing times of cognitive engines are highly variable and content-dependent. Measurements summarized in Figure 15 show that computation times for face recognition and OCR (Open Source) can vary by 1 and 2 orders of magnitude, respectively. An image that takes much longer than average to process will cause queuing and an increase in latency for subsequent ones. However, with our two-level token bucket mechanism, explicit feedback controls the input rate to avoid building up queues.

We compare the differences in queuing behavior with and without our flow control mechanism using a synthetic cognitive engine. To reflect high variability, the synthetic application has a bimodal processing time distribution: 90% of images take 20 ms, 10% take 200 ms each. Images are assigned deterministically

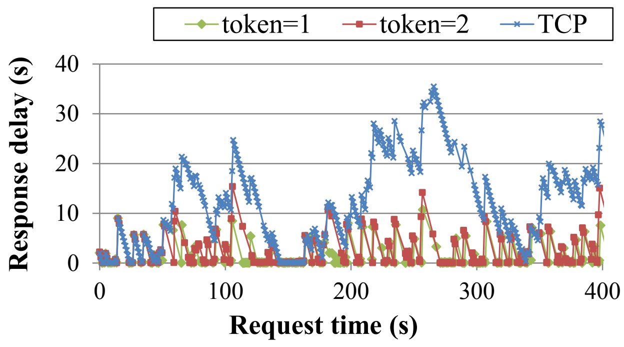
Application	1%	10%	50%	90%	99%
OCR(open)	11ms	13ms	29ms	3760ms	7761ms
Face	21ms	33ms	49ms	499ms	555ms

Figure 15: High variability in response time

to the slow or fast category using a content-based hash function. With this distribution, the average processing time is 38 ms, and with an ingress interval of 50 ms (20 fps), the system is underloaded on average. However, as shown in Figure 16-(a), without application-level flow control (labeled TCP), many frames exhibit much higher latencies than expected due to queuing. In contrast, the traces using our token-based flow control mechanism with either 1 or 2 tokens reduce queuing and control latency. There are still spikes due to the arrival of slow frames, but these do not result in increased latency for subsequent frames. We repeat the experiment with a real cognitive engine, OCR (Open Source). Here, the mean processing time is just under 1 second, so we use an offered load of 1 FPS. The traces in Figure 16-(b) show similar results: latency can increase significantly when explicit flow control is not used (TCP), but latency remains under control with our token-based flow control mechanism.



(a) Synthetic Cognitive Engine (20 FPS ingress)



(b) OCR (1 FPS ingress)

Figure 16: Response delay of request

The tight control on latency comes at the cost of throughput – the token based mechanism drops frames at the source to limit the number of data items in flight though the system. Figure 17 shows the number of frames that are dropped, processed on time, and processed but late for the OCR experiment. Here, we use a threshold of 0.5 s as the "on-time" latency limit. The token-based mechanism (with tokens set to 1) drops a substantial number of frames at the source, but the majority of processed frames exhibit reasonable latency. Increasing tokens (and number of frames in flight) to 2 reduces the dropped frames slightly, but results in more late responses. Finally, the baseline case without our flow-control mechanism attempts to process all frames, but the vast majority are processed late. The only dropped frames are due to the output queues filling and blocking the application, after the costs of transmitting the frames from

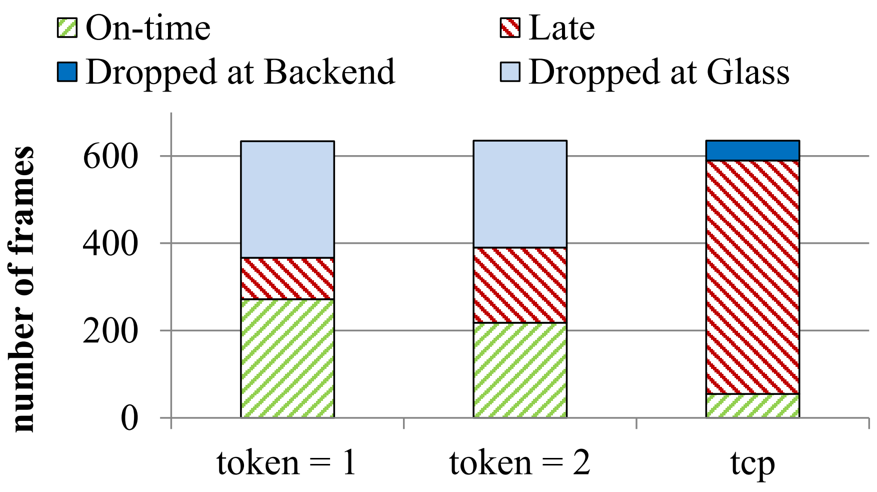


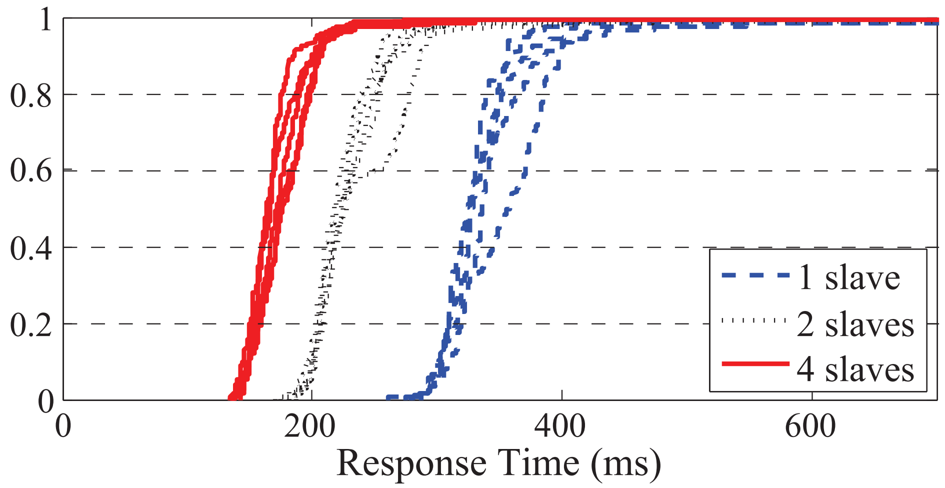
Figure 17: Breakdown of processed and dropped frame for OCR (500ms for on-time response)

the Glass front-end have been incurred. In contrast, the token scheme drops frames in the Glass device before they are transmitted over Wi-Fi. These results show that our token-based flow control mechanism is effective and necessary for low latency as well as saving energy.

6.5 Scale-Out of Cognitive Engines

We use the Motion Classifier cognitive engine (Section 5.5) to explore how well Gabriel supports parallelization across multiple VMs. We modified this application to use a master-slave scale-out structure. Feature extraction and classification are both done in parallel on slaves, and pipelined for higher throughput. Feature extraction is parallelized by splitting every image into tiles, and having each slave extract features from a different tile. Classification is parallelized by having each slave receive the full feature vector, but only doing an SVM classification for a subset of classes. The master and each slave are mapped to different VMs. The master receives sensor data from the control VM, distributes it to the slaves, gathers and merges their results, and returns the final result back to the control VM.

Since the final classification is done based on a window of frames, we define the latency of this offload engine as the time between when the last frame in the window is captured, and when the classification result is returned. Figure 18 shows how latency drops as we use more slave VMs. The 90^{th} percentile latency decreases from about 390 ms with one slave, to about 190 ms with four slaves. Throughput also improves. Using one, two, and four slave VMs, we get frame rates of 9.8 fps, 15.9 fps, and 19.0 fps.



Each family of CDFs shows five experimental runs with the same number of slaves. The tight clustering within each family shows that the observed improvement in latency across families is significant: i.e., increasing the number of slave VMs does indeed reduce latency.

Figure 18: Latency Reduction by VM Scale-Out

6.6 Full System Performance

Thus far we have presented experimental results examining an individual tradeoff or technique employed by Gabriel to bound latency. In this section we investigate the performance of the entire system with the cognitive engines executing together. We use a different pre-recorded video for this experiment which includes footage recognizable by the cognitive engines such as signs with text for OCR, a Coke can for object recognition, human faces for face recognition, and waving for motion recognition.

Figure 19 shows the performance of each offloading engine during this full system benchmark. Frame rate per second (FPS) and response time is measured at the Glass. Because each cognitive engine's compute time varies drastically based on content, this experiment is not comparable to the previous

Engine	FPS	Response time (ms)				
		1%	10%	50%	90%	99%
Face	4.4	196	389	659	929	1175
MOPED	1.6	877	962	1207	1647	2118
STF	0.4	4202	4371	4609	5055	5684
OCR(Open)	14.4	29	41	87	147	511
OCR(Comm)	2.3	394	435	522	653	1021
Motion	0.7	126	152	199	260	649
AR	14.1	48	72	126	192	498

Figure 19: FPS and latency of cognitive engines

microbenchmarks. But, the results reflect that the cognitive engines operate independently within their VMs. Most importantly, the overall system is not limited to the slowest cognitive engine. The cognitive engines that can process frames quickly operate at higher frame rates than the slower cognitive engines, which is precisely what Gabriel promises. In addition, the quickest cognitive engines maintain end-to-end latency of tens of milliseconds.

6.7 Impact of Reducing Fidelity

To study the effect of reducing fidelity, we focus on the most heavyweight cognitive engine in our suite. The response times in Figure 19 indicate that STF object recognition (described in Section 5.5) is very slow even on a cloudlet. On a fallback device, the performance would be even worse.

For object recognition, the resolution of individual video frames is a natural parameter of fidelity. The highest resolution supported by a Glass device is 1080p. Lower resolutions are 720p, 480p and 360p. When resolution is lowered, the volume of data transmitted over Wi-Fi is reduced. This lowers transmission time, as well as the energy used for transmission. Lower resolution also reduces the processing time per frame on a fallback offload device.

	Response Time	Glass energy per frame	Glass power	Fallback device power
Cloudlet				
1080p	12.9 (1.2) s	23 J	1.8 W	NA
Laptop				
1080p	27.3 (1.7) s	55 J	2.0 W	31.6 W
720p	12.2 (0.4) s	25 J	2.0 W	31.7 W
480p	6.3 (0.6) s	13 J	2.0 W	31.4 W
360p	4.3 (0.8) s	9 J	2.1 W	31.3 W
Netbook				
480p	32.9 (1.0) s	59 J	1.8 W	14.3 W
360p	20.5 (0.4) s	41 J	2.0 W	14.4 W

Each experiment was run for about 20 minutes to get at least 36 frames processed by STF. Numbers in parentheses are standard deviations of response times across frames. These measurements used token setting 1 to get the best response time possible. The monitor on the fallback device was turned off to save energy.

Figure 20: Energy & Response Time vs. Fidelity

Figure 20 shows the measured response time and energy usage on the Glass device and on the offload

	body	building	car	chair	dog
1080p	2408	875	122	22	1004
720p					
False neg.	56	5	4	0	39
False pos.	19	137	52	1	14
480p					
False neg.	124	19	11	2	83
False pos.	24	219	136	2	25
360p					
False neg.	223	39	14	3	122
False pos.	23	273	176	5	35

These numbers indicate the quantity of objects in each class detected at different resolutions of the test data set. For each class, the number detected at 1080p is ground truth.

Figure 21: Fidelity versus Accuracy

device, as fidelity is reduced. The response time (12.9 s) and Glass energy per frame (23 J) at 1080p with cloudlet offload are baseline values. At 1080p with laptop offload, these values increase to roughly twice their baselines. At 720p with laptop offload, these values are comparable to their baselines. At lower fidelities with laptop offload, these values are below their baselines. The netbook is much too slow at 1080p and 720p, so Figure 20 only shows values for 480p and 360p. Even at these fidelities, response time and Glass energy per frame are both much higher than baseline.

The fourth and fifth columns of Figure 20 show that power consumption on the Glass device and the fallback device are quite stable, regardless of fidelity. This is because the higher frame rate at lower fidelity compensates for the lower energy per frame. At these power draws, a rough estimate of battery life using data sheets gives the following values: 1 hour for Glass device, 1.5 hours for laptop, and 2 hours for netbook. These are long enough to cover Internet coverage gaps in the daily life of a typical user.

Although reducing fidelity hurts the accuracy of object recognition, the relationship is not linear. In other words, relative to the improvement in response time and energy usage, the loss of accuracy due to fidelity reduction is modest. This is illustrated by Figure 21, which shows results based on the processing of 3583 video frames from 12 different video segments. The STF object classifier is trained on the 21 classes of objects in the MSRC21 data set described by Shotton et al. [27]. For brevity, we only show results for 5 representative object classes ("body," "building," "car," "chair," and "dog"). The results for the other 16 classes follows this general pattern. We treat the classifier output at 1080p to be ground truth. At this resolution, there are 2408 instances of "body," 875 instances of "building" and so on.

As fidelity is reduced, classifier performance suffers both through false negatives (i.e., objects that are missed) and through false positives (i.e., detected objects that don't really exist). Figure 21 shows that there are relatively few false negatives. Even in the worst case (for "chair" at 360p), only 3 out of 22 objects (i.e., 13.6%) are missed. The picture is murkier with respect to false positives. The "building" and "car" classes show a high incidence of false positives as fidelity is reduced. However, the "body," "chair," and "dog" classes only show a modest incidence of false positives as fidelity is reduced. For use cases where false negatives are more harmful than false positives, the data in Figure 21 suggests that the wins in Figure 20 come at an acceptable cost.

7 Future Work and Conclusion

Sustained progress in foundational technologies has brought the decade-long dream of mobile, real-time cognitive assistance via "magic glasses" much closer to reality. The convergence of mobile and cloud computing, the increasing sophistication and variety of cognitive engines, and the widespread availability of wearable hardware are all coming together at a fortuitous moment in time. The work presented here is an initial effort towards understanding this domain in depth and identifying areas where improvements are most critical.

Most urgently, our results show that significant speed improvements are needed in virtually all cognitive engines. The measurements reported in Section 6 indicate typical speeds of a few hundred milliseconds per operation, rather than the desired target of a few tens of milliseconds. In a few cases such as STF object recognition, the speed improvement has to be even larger. Exploiting cloudlet parallelism can help, as shown by the results of Section 6.5. Restructuring the internals of cognitive engines to exploit parallelism and to make them more efficient is a crucial area of future effort.

Our results also show that significant improvements are needed in wearable hardware. The Google Glass devices used in this paper have many of the right attributes in terms of weight, size, and functionality. Their computing power is adequate when combined with cloudlet offloading, although their thermal sensitivity (discussed in Section 6.1) limits their usefulness. However, their battery life (less than two hours on the workloads reported here) is unacceptably short for real-world deployment as assistive devices. A factor of four improvement in battery life on cognitive workloads, without compromising other attributes, would be highly desirable. Even longer battery life would, of course, be welcome.

Our results reinforce the need for cloudlet infrastructure to attain the low end-to-end latencies that are essential for real-time cognitive assistance. Although the need for cloudlets is appreciated by the mobile systems community, there are no commercial deployments of such infrastructure yet. Identifying the business models that can sustain cloudlet deployments, and then catalyzing such deployments in the real world are important prerequisites for success in this domain. Our measurements suggest that cloudlets have to be substantial computational engines, involving many multi-core processors with large amounts of memory each. They essentially have to be server-class hardware, stationed at the edges of the Internet. This is quite different from the early industry thinking about cloudlets, that tends to see them as modest adjuncts of Wi-Fi access points [1].

Ultimately, real progress will come from experimental deployments involving users who are genuinely in need of cognitive assistance. Only live use will reveal HCI, usability, robustness and scalability issues that will need to be addressed for a full solution. The Gabriel prototype represents a humble first step, providing an open-source foundation for exploring this exciting new domain.

References

[1] IBM and Nokia Siemens Networks Announce World's First Mobile Edge Computing Platform. http://www.ibm.com/press/us/en/pressrelease/40490.wss, 2013.

[2] T. Agus, C. Suied, S. Thorpe, and D. Pressnitzer. Characteristics of human voice processing. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, Paris, France, June 2010.

[3] E. Cuervo, A. Balasubramanian, D. Chok, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, San Francisco, CA, June 2010.

[4] S. R. Ellis, K. Mania, B. D. Adelstein, and M. I. Hill. Generalizeability of Latency Detection in a Variety of Virtual Environments. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 48, 2004.

[5] J. Flinn. *Cyber Foraging: Bridging Mobile and Cloud Computing via Opportunistic Offload*. Morgan & Claypool Publishers, 2012.

[6] J. Flinn and M. Satyanarayanan. Energy-aware Adaptation for Mobile Applications. In *Proceedings of the Seventeenth ACM Symposium on Operating systems Principles*, 1999.

[7] Google. tesseract-ocr. http://code.google.com/p/tesseract-ocr/.

[8] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code Offload by Migrating Execution Transparently. In *Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation*, Hollywood, CA, October 2012.

[9] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. The Gator Tech Smart House: A Programmable Pervasive Space. *IEEE Computer*, 38(3):50–60, 2005.

[10] C. D. Kidd, R. Orr, G. D. Abowd, C. G. Atkeson, I. A. Essa, B. MacIntyre, E. D. Mynatt, T. Starner, and W. Newstetter. The Aware Home: A Living Laboratory for Ubiquitous Computing Research. In *CoBuild '99: Proceedings of the Second International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture*, pages 191–198, 1999.

[11] M. B. Lewis and A. J. Edmonds. Face detection: Mapping human performance. *Perception*, 32:903–920, 2003.

[12] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *Proceedings of the 10th annual conference on Internet measurement*, 2010.

[13] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[14] M. Missfeldt. How Google Glass Works. `http://www.brillen-sehhilfen.de/en/googleglass/`, 2013.

[15] MOPED. MOPED: Object Recognition and Pose Estimation for Manipulation. `http://personalrobotics.ri.cmu.edu/projects/moped.php`.

[16] D. Narayanan and M. Satyanarayanan. Predictive Resource Management for Wearable Computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, San Francisco, CA, 2003.

[17] S. Nirjon, R. F. Dickerson, Q. Li, P. Asare, J. A. Stankovic, D. Hong, B. Zhang, X. Jiang, G. Shen, and F. Zhao. Musicalheart: A hearty way of listening to music. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, 2012.

[18] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[19] OpenCV. OpenCV Wiki. `http://opencv.willowgarage.com/wiki/`.

[20] OpenStack. `http://www.openstack.org/software/grizzly/`, April 2013.

[21] M. Posner and S. Peterson. The Attention System of the Human Brain. *Annual Review of Neuroscience*, 13:25–42, 1990.

[22] M. Ramon, S. Caharel, and B. Rossion. The speed of recognition of personally familiar faces. *Perception*, 40(4):437–449, 2011.

[23] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling User-perceived Delays in Server-based Mobile Applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Farminton, PA, 2013.

[24] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada, 1996.

[25] M. Satyanarayanan. Augmenting Cognition. *IEEE Pervasive Computing*, 3(2):4–5, April-June 2004.

[26] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), October-December 2009.

[27] J. Shotton, M. Johnson, and R. Cipolla. Semantic texton forests for image categorization and segmentation. In *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*, June 2008.

[28] V. Stanford. Using Pervasive Computing to Deliver Elder Care Applications. *IEEE Pervasive Computing*, 1(1), January-March 2002.

[29] G. Takacs, M. E. Choubassi, Y. Wu, and I. Kozintsev. 3D mobile augmented reality in urban scenes. In *Proceedings of IEEE International Conference on Multimedia and Expo*, Barcelona, Spain, July 2011.

[30] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

[31] VeryPDF. PDF to Text OCR Converter Command Line. `http://www.verypdf.com/app/pdf-to-text-ocr-converter/index.html`.

[32] WattsUp. .NET Power Meter. `http://wattsupmeters.com/`.

[33] M. yu Chen and A. Hauptmann. MoSIFT: Recognizing Human Actions in Surveillance Videos. Technical Report CMU-CS-09-161, CMU School of Computer Science, 2009.