Scaling Task Management in Space and Time:
Reducing User Overhead in Ubiquitous-Computing Environments

João Pedro Sousa

CMU-CS-05-123
March 28, 2005

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh PA

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

This research was sponsored by the US Office of Naval Research under contract no. N66001992891806, the US Army Research Office under contract no. DAAD190110485, the US Air Force Research Laboratory under contract no. F306020020616, and the National Science Foundation under grant nos. CCR0205266 and CCR0113810. The author's graduate studies at Carnegie Mellon University were supported in part by the Portuguese Ministry of Science and Technology, under grant Praxis XXI/BD/15634/98, and by the Portuguese-American Foundation for the Development (FLAD). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: task management, task-driven computing, software engineering, software architecture, ubiquitous computing, mobile computing, self-configurable systems, adaptive systems, model-based adaptation, modeling user preferences, utility-theoretic models, user attention, Project Aura.

Abstract

Advances in computing and networking are prompting users to change their expectations about the availability of computing. Instead of making primary use of a single machine, users may expand their computer-supported tasks across multiple locations, and they may work on some tasks for days or even months. It is well known that such tasks typically involve several applications and information resources, making it a chore to rebuild the state of devices and software for resuming a task interrupted somewhere else or sometime ago.

Unfortunately, current systems offer little support for scaling task management in space and in time, and consequently users are torn between taking advantage of increasingly pervasive computing systems, and the price (in attention and skill) that they have to pay for using them.

This dissertation describes a new approach to the scalability of task management in space, across heterogeneous environments, and in time, allowing users to recover tasks interrupted long ago. The approach is based on high-level models of what users need from the computing environment for each of their tasks. Such models are exploited at run-time by an infrastructure that automatically configures the computing environment, on demand, on behalf of users.

We present an architectural framework that grounds our approach, and that embodies new system design principles that hold independently of the particular infrastructure implementing the framework. As part of the framework, we present a utility-theoretic model that enables finding the best match between user needs and the capabilities and resources in the environment.

We evaluate our research from three perspectives. First, from a user's perspective, we validate that the infrastructure: (a) delivers the capabilities for scaling task management in space and in time; (b) that it reconciles the competing requirements of sparing users from routine configuration chores, while enabling them to take full advantage of the surrounding computing environments; and (c) that it is usable by non-experts. Second, from a software architect's perspective, we evaluate the benefits and limitations of the architectural framework supporting our approach. And third, from a systems perspective, we validate that the infrastructure exhibits a performance that makes it usable on a daily basis.

To Afonso
(*b*. 1996)

Alegria da Criação
(*Joy of Creation*)

Plantei a semente da palavra
antes da cheia matar o meu gado
ensinei a meu filho a lavra e a
colheita num terreno ao lado.

A palavra rompeu
cresceu como a baleia
no silêncio da noite
há lua cheia
vi mudar estações
soprar a ventania
brilhar de novo o Sol
sobre a baia.

Fui um bom engenheiro
um bom castor
amei a minha amada
com amor
de nada me arrependo
só a vida
me ensinou a cantar
esta cantiga.

José Afonso
(1929 – 1987)

Acknowledgements

Standing on the brink of completing this dissertation, I can't help reflecting about the people that contributed to the path that brought me to this point, and more deeply, to who I am.

I wish to acknowledge:

David, for deconstructing the center for articulated thought in my brain and rebuilding it from scratch (a work in progress). David did more than showing me technical tools of the trade: he was a true teacher in the sense that he gave me tools that shaped the way I look at problems and, no less important in engineering, solutions.

Satya, Reid and Gregory, both for asking hard questions and for their valuable contribution in scoping an interesting, but manageable, problem.

Vahe, Bradley and Rajesh, for shoring up my research with their own, and for selflessly contributing to the results in this dissertation.

The faculty of the School of Computer Science, for building a truly supportive environment for research and learning.

José Granado (can't do without a last name here), for being an agent of change, constantly opening doors leading to bright places.

Amílcar, for introducing me to scientific honesty and passion.

My parents, Lídio and Maria Augusta, for unquivering support, even when they had no idea what I was up to.

My sister, Isabel, for challenging me to grow up.

Ana, for being the tree where the swallow stops.

My friends, who keep teaching me what being human is about.

Table of Contents

CHAPTER 1 INTRODUCTION **1**

THE PROBLEM 2

How Existing Solutions Fall Short *3*

THESIS 4

CHALLENGES 5

Modeling User Tasks *5*

Infrastructure for Task Management *7*

Describing and Operating on User Tasks *8*

SUMMARY OF THE RESEARCH 9

Describing and Operating on Tasks *10*

Models of User Tasks *12*

Infrastructure for Task Management *13*

Thesis Validation *14*

Contributions *15*

PLAN OF DISSERTATION 16

CHAPTER 2 RELATED WORK **17**

USER TASKS 17

Task Management *18*

Assistance with Tasks *19*

Task Modeling *20*

ADAPTIVE SYSTEMS 21

Service Awareness *21*

Resource Awareness *22*

Context Awareness *22*

CHAPTER 3 MODELING TASKS 24

COORDINATED USE OF SERVICES 24
Identifying Services 25
State Snapshot 26
Interconnecting Services 26
Specification 28
USER PREFERENCES 30
Configuration Preferences 31
Supplier Preferences 32
QoS Preferences 33
Specification 35
TASK IDENTITY 36

CHAPTER 4 INFRASTRUCTURE FOR TASK MANAGEMENT 39

TWO SOURCES OF CHANGE 40
Changes in User Tasks 42
Changes in the Environment 43
HETEROGENEITY AND DISTRIBUTION 43
Heterogeneity 44
Distribution 44
Verifying the Protocols of Interaction 46
FINDING THE BEST MATCH 46
ADAPTATION AT THREE LEVELS 48
Integrating Adaptive Applications 49
Coordinating Policies 50
IMPLEMENTATION 51
Prism 51
Suppliers 52
Evaluation 53

CHAPTER 5 DESCRIBING AND OPERATING ON TASKS 56

OPERATING ON TASKS 57
Tasks at a Glance 57
Finding Tasks 58
Task Life-cycle 60
DESCRIBING TASKS 61
Coordinated Use of Services 61
Preferences 62
Task Identity 64

CHAPTER 6 THESIS VALIDATION **66**

SCALABILITY IN SPACE 67
SCALABILITY IN TIME 68
TAKING FULL ADVANTAGE 69
REDUCING THE OVERHEAD 70
Coordinated Use of Services *71*
Scalability in Space *75*
Scalability in Time *76*
Overall Reduction *77*

CHAPTER 7 DISCUSSION **78**

MODELING USER TASKS 78
Sophistication of Task Models *79*
Task Recurrence *80*
Service Naming and Substitutability *80*
Service Decomposition *82*
Service Interconnection *83*
Supplier Preferences *83*
QoS Tradeoffs *84*
INFRASTRUCTURE FOR TASK MANAGEMENT 86
User Mobility vs. Privacy *86*
Software Engineering of Service Suppliers *88*
Representation of Service State *89*
Interconnection of Service Suppliers *90*
DESCRIBING AND OPERATING ON TASKS 90
Incremental Benefit for Incremental Effort *91*
Finding Past Tasks *91*

CHAPTER 8 CONCLUSION AND FUTURE WORK **93**

CONTRIBUTIONS 95
Contributions of the Approach *95*
Contributions of the Architectural Framework *96*
Contributions of the Infrastructure *97*
FUTURE WORK 97
Delegation *98*
Scoping the Environment *99*
Proactive Suppliers *100*
Guidance on Tasks *101*
Learning Task Models *102*
EPILOGUE 103

REFERENCES **105**

APPENDIX A **110**

PRISM – EM PROTOCOL 110
PRISM – SUPPLIERS PROTOCOL 113
PRISM 114
PROTOCOL EM – SUPPLIERS 115
EM 116

Chapter 1

Introduction

It is well known that computer users may simultaneously handle several tasks, such as preparing presentations, writing reports, or answering email, constantly shifting their attention between those tasks. This fact was observed twenty years ago [11], and it certainly holds today [25].

One important property of such tasks is that they typically involve several applications and information assets. For instance, for preparing a presentation, a user may edit slides, refer to a couple of papers on the topic, check previous related presentations, and browse the web for new developments. Existing work on desktop management has addressed this property, from early work on Rooms [18], through recent work such as the GroupBar [84].

Another, increasingly important property of user tasks is that they may span multiple locations. Advances in ubiquitous computing are prompting people to change their expectations towards the availability of computing [2]. Rather than being bound to a specific device, users may desire to take full advantage of the computing systems accessible to them, much as they take advantage of the furniture in each physical space. In the example above, a user may start working on the presentation while in his or her office, continue at the office of a collaborator, and pick the task up later at home. Ideally, the user should not have to carry a machine around, just as people don't have to carry their own chairs. If they so desire, users should be able to resume their tasks, on demand, with whatever computing systems are available.

Yet another important property of user tasks is their duration and recurrence. Users may work on some tasks for days or even months. Tasks may need to be referred back to or restarted after the user thought they were done. For instance, a user may need to find a presentation given last year, so that it can be updated with the latest data and developments for an upcoming meeting. And note that this is not a question of finding one file that resulted from the task, but a question of finding the task (definition) itself, so that it can be reactivated or used as a template to create a similar task. In fact, some tasks recur periodically; or to be more precise, users may periodically carry out distinct instances of the same kind of task. For instance, if a user prepares monthly reports, although such tasks share some characteristics, each has its own identity and may diverge from the common pattern: in July the user might need to include some slides for top management, but not in August.

The Problem

Unfortunately, current computer systems offer little support for the properties above, and users carry the burden of configuring the computing *environment*[1] every time they resume a task interrupted somewhere else, or sometime ago. Users have to deal with finding and starting suitable hardware and software components; and they have to deal with accessing the relevant information.

In the example of preparing a presentation, introduced above, suppose that the user wants to pick up the task at home using the local computing capabilities, after having interrupted it at the office. It will be up to the user to:

– Recall which services were required from the environment for preparing the presentation (edit slides, review related materials, browse the web, etc.).
– Map those services into the set of applications available at home, which may be distinct from the ones at the office, for instance if different operating systems are involved.
– Recall which files/web addresses were being consulted and manipulated, and ensure access to those files.
– Activate the necessary applications and recover their user-perceived state: open files, recover the working position within those files, recover the layout of windows, recover important application settings, etc.

To make matters worse, users are required to manage resources and dynamic change in mobile computing environments. Mobile computing is increasingly available; however, wireless networks and mobile devices expose users to wide variations of resources, such as battery and bandwidth. To obtain the desired level of quality of service, users need to be aware of the demand that alternative computing modalities pose on limited resources. Moreover, a setup that corresponds to the user’s expectations at some point may be unacceptable a few moments later: for example, in heavily networked environments, remote servers constantly change their response times and even availability.

[1] The computing environment is the set of devices, applications, services and resources that are accessible to a user at a particular location. The accessible devices and software determine the environment’s capabilities (what it can do), while the resources (things like bandwidth and battery) influence how well it can carry out those capabilities.

A consequence is that users are torn between taking advantage of the increasingly pervasive computing systems, and the price (in attention and skill) that they have to pay for using them.

Ideally, users should have easy access to their tasks, no matter where or when those tasks were interrupted before. Furthermore, a good solution should satisfy two competing requirements:

- Users should be able to take full advantage of the computing environment at each location, if they so desire, and as required by their tasks.
- To the extent possible, users should be spared from the routine chores of configuring and reconfiguring the computing environment to support their tasks.

Take the example of resuming at a different location the task of preparing a presentation, above. Ideally the user would be able to browse his currently pending tasks, no matter where they were defined, and upon indicating that he wished to resume preparing the presentation, the required applications would be automatically identified in the local environment, activated, and their user-perceived state reconstructed according to the last time the user worked on that task.

How Existing Solutions Fall Short

Several approaches have been developed to assist users with carrying out their tasks. However, these approaches fall short with respect to simultaneously addressing the scalability of task management in space and time, and the competing requirements of taking full advantage of available capabilities while saving user overhead. Specifically, this subsection outlines the limitations of five existing technologies: desktop managers, remote access using thin clients, mobile devices, mobile code, and internet suspend-resume. Chapter 2 elaborates on related work.

Current desktop managers enable users to associate several applications with one task by keeping the applications within a specific workspace area, and to quickly switch between tasks by flipping from one workspace area to another (e.g., [28,50,58]). Existing desktop managers assume the primary use of one machine, and therefore do not address user mobility. Nevertheless, it would be tempting to combine desktop managers with a solution for user mobility, such as the ones discussed below.

Thin clients enable remote access to the user's computing server (e.g., [12,66]). Unfortunately, this approach has two serious drawbacks. First, it relies on a stable, fairly high-bandwidth, connection – something that is frequently not available to a mobile user. Second, thin clients fail to take advantage of local resources, and, in particular, of the ever-increasing capabilities of mobile devices, smart spaces, etc.

The use of mobile devices such as laptops and PDAs grants ubiquitous access to a user's personalized devices and software. However, while promoting the self-sufficiency of users, this approach offers no support for taking advantage of other devices and computing capabilities in the users' vicinity. For example, a user that joins a teleconference using his wireless PDA while walking down the hall has no automatic support for taking advantage of the large display, wired connectivity, and sophisticated teleconferencing application at the office he just entered.

Another approach to user mobility is based on mobile code, which is capable of migrating between devices (e.g., [13,67]). Mobile code examines the characteristics of each environment it is migrated to, and using internal logic, chooses appropriate interaction modalities. However, this approach relies on the compatibility of the virtual machines in all devices where the code will be migrated to. Furthermore, it relies on the ability of mobile applications to recognize and handle the characteristics of each device and other software components in the environment. Unfortunately, there is no guarantee that a mobile application will be able to run on every device that the user chooses to utilize. Even if it does run, there is no guarantee that it will provide a quality of service comparable to a local, custom built application. Furthermore, this approach doesn't enable users to take full advantage of the wealth of existing software, since it requires the development of all new applications equipped for mobility, and often requires significant extensions to operating systems for distributed access to data and services.

An approach for mobility that requires no changes to existing software is based on migrating the contents of the whole virtual memory between machines: internet suspend-resume [54]. This approach, however, is limited to cases where users move only among machines with compatible hardware, and therefore fails to enable users to take advantage of diverse environments.

In summary, the approaches above do not address simultaneously the requirements of (a) scaling task management in space and time, (b) enabling users to take full advantage of the capabilities around them, and (c) saving users from routine configuration chores. Some approaches do a good job at addressing some of the requirements, but do a poor job at addressing the others.

Thesis

This dissertation describes a new approach to the scalability of task management in space and in time, for the class of tasks that can be found in an office. The approach is based on high-level models of what users need from the computing environment for each of their tasks. Such models are exploited at run-time by an infrastructure that automatically configures the environment, on demand, on behalf of users. Specifically, this dissertation shows that:

> *High-level models of user tasks can be used to address the scalability of task management in space, across heterogeneous environments, and in time; while simultaneously (a) enabling users to take full advantage of the capabilities and resources accessible in each environment; and (b) relieving users from routine chores associated with configuring and managing those environments.*

The model of each task includes the definition and the state of the task. The task definition represents the collection of services involved in that task. For instance, for preparing a monthly report, the user may edit the report text, work on a data spreadsheet, and prepare accompanying slides for presenting the highlights. The state of the task captures a snapshot of the *user-perceived state* of each service: things such as layout of windows, files being worked on, cursor positions and application settings.

The scalability of task management in space and time is addressed as follows:

- By adopting a model of tasks that represents what users need for each task, we can reactivate a task at any time, and at any location, matching the user's needs with the services available in each environment.
- By making such a model independent of specific applications, we can reactivate the task on different platforms. For instance, for the task of preparing a report, we capture the fact that the user needs to *edit a text document*, not that MS Word is involved in the task.
- By using the programming interfaces (APIs) exposed by applications, we capture a high-level model of the user-perceived state. Those APIs are used later to reconstruct the user-perceived state of the task, as needed.
- By giving tasks a persistent semantic identity (which goes beyond a name, or a set of currently active applications) we enable users to find and manipulate tasks, regardless of where or how long ago those tasks were defined.

The competing requirements of enabling users to take full advantage of computing environments, while reducing the associated configuration overhead, are addressed as follows:

- By finding the available components that best fit the user's needs for each task, the infrastructure enables users to take full advantage of diverse environments. Furthermore, by keeping track of user preferences with respect to quality of service, of resource demand posed by alternative computing modalities, and of resource availability, the infrastructure can select the optimal configuration and carry out dynamic adaptation to changes in the environment.
- By automatically configuring environments, on demand, and by continuously adapting to changes in the environment, the infrastructure reduces the burden of routine configuration and reconfiguration chores for the user.

Challenges

Demonstrating the thesis above entails three top-level challenges: first, we need models of user tasks that serve the goal of this research. Second, we need to build an infrastructure that exploits such models for configuring and continuously adapting the environment to the requirements of ongoing user tasks. And third, we need to incorporate mechanisms into the infrastructure that enable users to describe and operate on their tasks. The following subsections discuss these challenges in turn, and Chapters 3, 4, and 5 discuss in depth how each is addressed.

Modeling User Tasks

Considerable work has addressed modeling user tasks with goals such as (a) to assist or direct the user during complex, multi-step tasks (e.g., [6,36,81,100]), (b) to predict what the user may be doing next (e.g., [4,44]), and (c) to assist designers in analyzing and building computer systems (e.g. [47,60,96,99] – see Chapter 2 for related work).

In this dissertation, the goal is different: task models are used to reconstruct a configuration of services in the environment, and to dynamically adjust that configuration in the face of changes in resource availability. Although some research in Human Computer Interaction has exploited user models to build self-configurable computer systems, its focus has been on the usability of individual applications, specifically on automatically adapting user interfaces to diverse device characteristics (e.g., [16,52,89]).

The main role of task models in this research is to provide users with a notion of task that involves the coordinated use of a set of services in an environment, and that can be suspended and resumed as a whole. In this context, task models must be detailed enough to support the automatic configuration of the environment, reconstructing, to the extent possible, the user-perceived state of the services involved in the task. However, since different environments may have very different applications and interaction models, task models should not attempt to capture detailed models of *how* the user interacts with concrete applications. In other words, task models must be abstract enough to be environment-independent, but precise enough to allow for the instantiation of the task using the concrete capabilities of the current environment.

Additionally, to address the heterogeneity of ubiquitous computing environments, and the fact that their resources are subject to frequent variation, task models should capture user preferences relative to functionality and to Quality of Service (QoS). Preferences relative to functionality play a key role in choosing among alternative configurations for supporting a task. For instance, if a task involves editing a text document and two distinct text editors are available in the environment, which should be activated? The answer depends on how the user appraises the functionality of each editor for the task at hand. For editing natural language, the user may prefer an editor with automatic spell checking, but for editing a computer program that feature may hinder more than help.

Preferences relative to QoS play a key role in guiding the policies for adaptation to resource variation. For that, however, we need to take into account that QoS is seldom expressed along a single dimension. For instance, suppose that the user is watching a video over a network link and that the bandwidth suddenly drops. Should an adaptive video player reduce the frame rate, or the image quality? The answer depends on the user's preferences for the current task. If the user is watching a sports event, he may prefer frame rate to be preserved at the expense of image quality. For watching a documentary on painting, the opposite might be preferable. Task models should capture the QoS tradeoffs that are appropriate for each task.

Furthermore, user preferences should be expressive enough to represent which reconfigurations are safe to be carried out automatically, without user intervention, and which are potentially disruptive and should be run by the user.

Finally, to address scalability in time, task models must establish an enduring identity that enables users to find (and resume, if necessary) tasks long after they are gone from the list of currently active tasks. Furthermore, task models should enable task browsing based on fuzzy remembrances, rather than expecting users to remember a specific task identifier, its precise classification, or when exactly the task was carried out. For instance, if a user needs to find "the report on the trip to Pittsburgh that I wrote last year," he may not remember the name of the file (Pittsburgh may not be mentioned in the name) and may have no idea under which directory it was filed. However the user may remember incidental facts about the report, such as the topic and who he collaborated with, and should be able to easily find the report based on whatever he remembers.

In summary, we need to define a computable model of user tasks that (a) describes the coordinated use of a set of services, in an environment-independent fashion; (b) captures user preferences relative to features and QoS; and (c) supports finding tasks after they disappear from the list of currently active tasks.

Infrastructure for Task Management

The purpose of the infrastructure is to automatically configure the environment, on demand, on behalf of users. Specifically, given a task model (and the user's intention to resume that task) the infrastructure is responsible for (a) determining the set of devices and software components that best match the user's needs for the task; (b) activating those components, reconstructing the user-perceived state, possibly adjusting resource allocation among them to meet the QoS requirements for the task; and (c) continuously monitoring for changes in the environment that would prompt a reconfiguration, and enacting reconfigurations, as appropriate.

In the context of daily task management, users constantly switch between tasks, incrementally change their needs for the ongoing task (e.g., by adding or removing services) and may change their QoS preferences midway through a task. For instance, during a task that involves automatic translation of natural language, a user may prefer faster responses over accuracy of translation during the introductory part of the conversation, but may prefer the opposite once the conversation becomes more involved. Of course, such changes in the user task may prompt the infrastructure to perform a reconfiguration, just as changes on the environment would.

We need to extend existing software frameworks for adaptation, to account for adaptation to changes originating both in the environment, and in user tasks. In fact, current research on adaptive software systems has focused on adapting to changes in the environment (fault tolerance, or adaptation to changing resources, such as battery and bandwidth) and most have assumed that the user's needs and preferences are relatively static.

Scaling task management in space implies dealing with heterogeneity of devices and software. Allowing users to take advantage of the computing systems accessible to them in different locations implies that the infrastructure has to accommodate devices ranging from mobile phones, to personal computers, to smart rooms; and has to deal with the plethora of software that comes with those devices. The design of the infrastructure needs to make it easy to accommodate existing, out-of-the-box software, while avoiding becoming locked into proprietary frameworks of interaction between components.

The fact that computing environments are increasingly distributed implies that the infrastructure must handle variations of resources and service availability as a normal situation, rather than exposing it to the user as exceptional (faulty) behavior. The increasing pervasiveness of smart spaces is causing a shift in the paradigm of computer use: from single-device, tightly integrated interaction (e.g., desktop and laptop computers), to multiple-device, loose interaction. For instance, for the task of preparing a review of a video clip, a user may watch the clip on a large wall-mounted display, dictate notes into a mobile phone, and transcribe the notes using speech recognition software running on a remote server. For such distributed setups, the infrastructure needs to monitor and proactively address situations like the depletion of the battery in the mobile phone, or the connectivity to a particular remote server.

While research on fault tolerance adopted a binary model of faults (either a component is fine, or there is a fault) some research on resource-adaptive applications adopted utility-theoretic models for comparing alternative computation strategies. Fault tolerant systems often have a system-wide view of the status of components, but unfortunately that view normally misses finer aspects of QoS. In contrast, resource-adaptive applications address the QoS aspects, but normally confine their view to the internal computing tactics of a single application. For instance, an adaptive natural-language translator running on a handheld may execute sophisticated algorithms on a remote server when bandwidth is plentiful, but may have to rely on simple local algorithms when the connection is flaky. In either case, the computation is “correct” in the sense that a translation is provided, but the quality (accuracy) of the translation is likely to be much better in the first case.

To find the best match between the user’s needs and what the system has to offer, we need to devise a model of QoS that supports both a system-wide view and local fine-grain adaptation to resource changes. Such a model plays a key role in guiding both the initial configuration of the system and ongoing adaptation to changes.

Once we devise such a model of QoS, we need a design for the infrastructure that exploits it consistently, while, if possible, taking advantage of existing results in system-wide adaptation and resource-adaptive applications (e.g. [55,62,64]).

In summary, the infrastructure needs to support (a) adaptation to dynamic changes both in the environment and in task requirements; (b) heterogeneity and distribution of devices and software components in the environment; (c) a model of QoS for guiding configuration and adaptation; and (d) an architectural framework for integrating the model of QoS with system-wide adaptation and resource-adaptive applications.

Describing and Operating on User Tasks

To make the infrastructure usable by non-expert users on a daily basis, it needs to be endowed with mechanisms that enable users to describe and operate on their tasks. The design of such mechanisms needs to consider the requirements targeted by this research, as elaborated below.

Users may wish to work on their tasks in environments with very different capabilities. Therefore, it will be important for users to obtain an indication, at a glance, of the feasibility of each of their tasks in the current environment. To avoid overwhelming the users with information that is likely to be irrelevant, an initial perspective should be limited to the tasks a user may want to work on at the present time and location. Of course, users should always be able to find and recover their tasks, whether or not those tasks are listed on the initial perspective. Furthermore, users should be able to use past tasks as templates for creating new, similar ones.

Once a user decides which task(s) to work on, it should be easy for him to resume the task as a unit, activating the configuration of all services involved in the task, and bringing up all the relevant materials. Likewise, once a user decides to suspend work on a task, it should be easy for him to capture a snapshot of the task and deactivate all the involved services.

The mechanisms for describing user tasks should be simple to use, yet powerful enough to capture the task models discussed above. There is a range of options, from offering users a programming language for describing tasks, to relying on artificial intelligence techniques for learning tasks by observation. In choosing such mechanisms, two aspects need to be considered: the effort required of users for describing their tasks, and how much knowledge about those tasks can captured by the infrastructure. If little knowledge can be acquired, there is little the infrastructure can do on the users' behalf. If the mechanisms for acquiring that knowledge are too cumbersome, users may get discouraged and not use the infrastructure at all.

These mechanisms should have a low entry cost for users and deliver incremental benefits for incremental effort. If the mechanisms for describing tasks demanded a significant initial investment, using the infrastructure to manage simple tasks would not pay off. Ideally, users should be able to reap some of the benefits of task management, even with very little effort put into describing their tasks. The more effort users are willing to put in describing their tasks, the better job the infrastructure for task management can do.

These mechanisms should also be explicit about the assumptions they make, and they should make it easy for users to correct incorrect assumptions. To provide a low cost of entry, we need to employ defaults to fill in models that would otherwise be incomplete. However, every default embodies assumptions about what the user intended. Rather than trying to second-guess users, those assumptions need to be made clear, and it must be easy for users to clarify what they really want.

In summary the mechanisms for operating on user tasks should (a) be simple to use, yet powerful enough to capture the task models defined above; (b) have a low entry cost and deliver incremental benefits for incremental effort; and (c) be explicit about the assumptions they make, and make it easy for users to make adjustments if those assumptions are at odds with the users' actual desires.

Summary of the Research

For this dissertation, I implemented an infrastructure that scales task management in space and in time. This research is part of a wider initiative in ubiquitous computing: Project Aura [35].

The infrastructure supports task management in the sense that it supports suspending and resuming user tasks as a coordinated set of services in the environment. Scalability in space is supported by allowing users to suspend tasks in one location and to resume them at another location, provided an installation of the infrastructure is available (see Figure 1.1). Scalability in time is supported by allowing users to browse their tasks regardless of how long ago those tasks were defined, or completed, and to resume them, if appropriate.

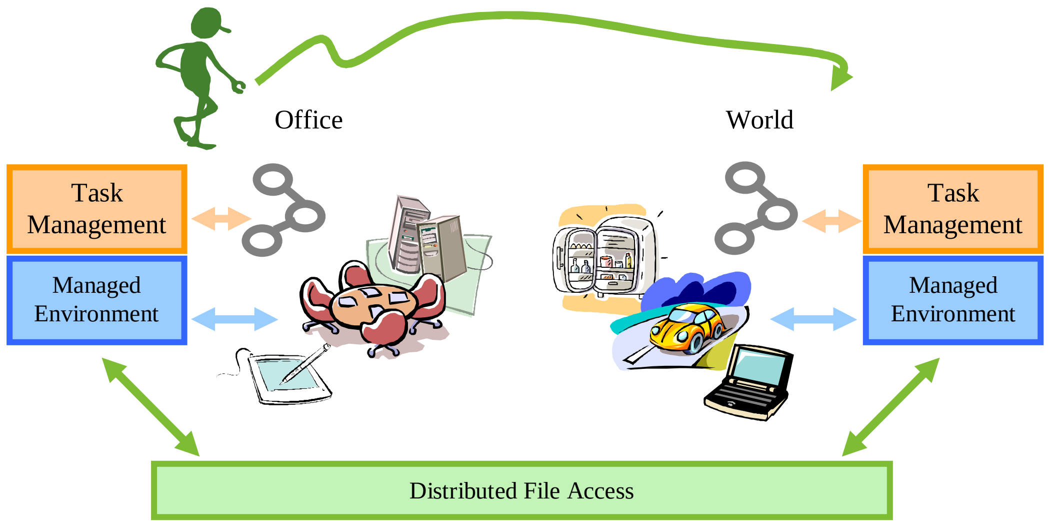


Figure 1.1 Scaling task management in space.
Users move from one location to another and may want to resume their tasks with the local capabilities. The Task Management layer keeps track of what users need from the computing environment for each task. The Managed Environment layer keeps track of the available capabilities and coordinates their configuration. The models of user tasks and personal information resources are shared via distributed file access mechanisms.

The approach for scalable task management described in this dissertation reduces the users' overhead with configuring computing environments, while enabling users to take full advantage of those environments.

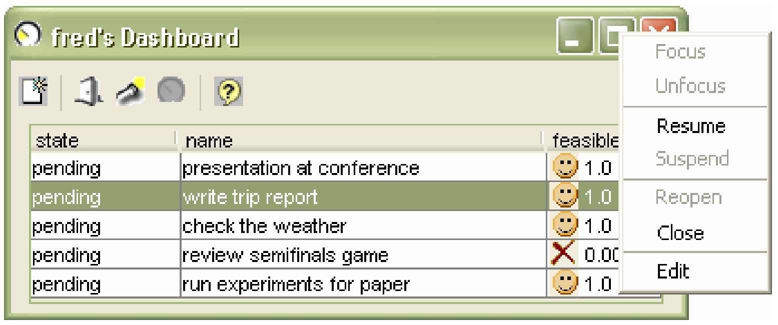
The following subsections summarize my research directed at each of the top-level challenges enumerated in the previous section, namely: (a) the interfaces for users to interact with the infrastructure for describing and operating on their tasks; (b) modeling user tasks for supporting scalable task management; and (c) designing and building an infrastructure for the automatic configuration of computing environments. These subsections also summarize the strategy for validating the thesis, as well as the contributions of this dissertation.

Describing and Operating on Tasks

Task management promotes user tasks to first class entities in computer systems, and thus enables users to operate directly on their tasks. Such operations treat as a unit the set of services (e.g. provided by applications) and materials (e.g. files) involved in a task. For instance, a user may *suspend* a task at the office and *resume* it in a coffee shop. However, to enable users to operate on their tasks, the infrastructure needs to be made aware of what is involved in each task.

The infrastructure includes interfaces for users to (a) describe their tasks, (b) operate on their tasks, and (c) find and recover their tasks. The interfaces implemented for supporting each of these aspects are summarized below.

The model of interaction for describing what is involved in a task is grounded in the widespread metaphor of drag-and-drop. For instance, suppose that a user, Fred, is about to start writing a



state	name	feasible
pending	presentation at conference	☺ 1.0
pending	write trip report	☺ 1.0
pending	check the weather	☺ 1.0
pending	review semifinals game	✗ 0.0
pending	run experiments for paper	☺ 1.0

Figure 1.2 Example task dashboard.

paper. Once Fred indicates he wants to create a new task description, a dedicated window is associated with that task, where Fred may consult and update the task description.

Fred may incrementally drop files that are relevant to the task into the description window. By dragging a file from standard file browsers and dropping it into the task description window, the infrastructure creates a reference to the file, but the file itself is not moved or copied.

When a file or internet shortcut is dropped, a default service is associated with it, for instance *edit spreadsheet*, if the file is a spreadsheet. Fred can change the service associated with each file, and he can also add services independently of files. For instance, Fred may add an *edit text* service for the paper he wants to write (but for which he has no file yet).

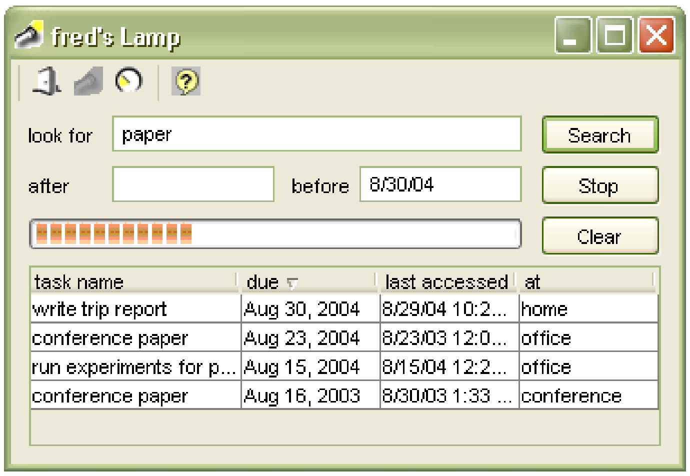
Users can operate on tasks either through the task description window or through the task *dashboard*, which shows the equivalent of a to-do list (see Figure 1.2). The task dashboard shows each user the list of tasks on which he may want to work on, and offers control over which tasks to access at which environments. For instance, a user may choose not to resume a task involving sensitive information at an untrusted environment.

The dashboard also shows an evaluation of how feasible is each task in the current environment. For evaluating a task's feasibility, the infrastructure identifies candidate applications and devices for supporting the services involved in the task, and estimates the availability of required resources (such as battery and bandwidth).

When users indicate their intention of resuming a task, the infrastructure finds and activates appropriate applications to supply the services and handle the materials involved in the task. When users indicate their intention of suspending a task, the infrastructure saves the files and deactivates the applications involved in the task (more on this in the subsection on the Infrastructure for Task Management, below).

Users may find their tasks based on circumstantial facts such as the purpose of the task, who collaborated on it, when was it due, etc. For that, and in addition to describing the services involved in a task, users may associate circumstantial facts with their tasks. We developed a task browser, called *lamp*, that employs a metaphor similar to web browsing (see Figure 1.3).

Chapter 5 elaborates on these interfaces and on how they address the specific challenges identified in the previous section. In the interest of scoping this dissertation, we restricted the research to tasks carried out by a single user (cooperative work is not addressed).



task name	due	last accessed	at
write trip report	Aug 30, 2004	8/29/04 10:2...	home
conference paper	Aug 23, 2004	8/23/03 12:0...	office
run experiments for p...	Aug 15, 2004	8/15/04 12:2...	office
conference paper	Aug 16, 2003	8/30/03 1:33 ...	conference

Figure 1.3 Example search for tasks.

Models of User Tasks

The role of task models in this research is to enable the automatic configuration of computing environments. Task models capture what users need from the computing environment for each of their tasks. These models are exploited at run-time by the infrastructure for automatically configuring the environment on behalf of users.

Task models enumerate which services are needed for each task, and whether and how those services should be interconnected. To account for the diversity in computing environments, the types of services required by a task are represented in abstract terms: *play video*, *edit text*, *edit slides*, etc. For instance, suppose that a user, Fred, is preparing a review of a video clip. While Fred works on his laptop, which runs MS Windows, viewing the clip is supported by Windows Media Player. However, when Fred reaches his office, he would like to take advantage of the large screen on his powerful desktop, which runs Linux. If Fred's task model includes the fact that he needs Media Player, the task cannot be instantiated on Fred's desktop. However, there is no good reason why, if Fred so desires, the video couldn't be played by whatever media player is available on Fred's desktop. By modeling tasks in abstract terms, whenever Fred reaches an environment, these services required for his task can be dynamically mapped to available suppliers that have the capability to provide them.

To address heterogeneity and resource variations in the environment, task models represent the user's preferences with respect to alternative ways to carry out the task and preferred quality of service tradeoffs. In this research, user preferences are expressed formally as *utility functions*. Then, finding the best match between what the user needs and what the environment has to offer corresponds to maximizing a utility function, where the environment's capabilities and available resources act as constraints in the maximization process. To make preferences easier to both elicit and process, we split them into three parts: first, *configuration preferences* capture preferences with respect to the set of services to support a task. Second, *supplier preferences* capture the desired features of the individual service suppliers; and third, *QoS preferences* capture the acceptable levels of quality of service and preferred tradeoffs.

To reduce the overhead for users in configuring the environment for their tasks, task models include a snapshot of the user-perceived state of each task. The user perceived-state includes things such as layout of windows, files being worked on, cursor positions and application settings. For example, for the *edit text* service, the snapshot would include the current *cursor position*, whether *spell checking* is currently activated, etc. Similarly to naming services, the snapshot of the user-perceived state is represented in abstract, application-independent terms.

An installation of the infrastructure can automatically configure the environment for user tasks, as long as the corresponding models (and information resources, such as user files) are available. In the current implementation, installations of the infrastructure access task models and information resources via a distributed file access system (see Figure 1.1).

Chapter 3 elaborates on the form and function of the task models adopted in this dissertation.

Infrastructure for Task Management

The role of the infrastructure for task management is to support the notion of user task as a coordinated set of services. For that, the infrastructure exploits task models for automatically configuring the environment on behalf of users.

To automatically configure the environment, first, the infrastructure needs to know *what* to configure for; that is, what users need from the environment to carry out their tasks. Second, the infrastructure needs to know *how* to best configure the environment: it needs to know which capabilities and resources are available in the environment, and it needs mechanisms to optimally match those to the user needs.

My research introduces an architectural framework, the Aura framework, where each of these two problems is addressed by a distinct software layer: (1) the Task Management layer, called Prism, determines *what* users need from the environment at a specific time and location; and (2) the Managed Environment layer determines *how* to best configure the environment to support user needs (see Figure 1.1).

Prism caters to user needs and preferences, capturing the corresponding task models. The interfaces described in the previous subsection are the visible part of Prism. When a user accesses a new environment, Prism coordinates accessing all the information related to the user tasks, and cooperates with the Managed Environment layer to find the best match for the user needs. When users indicate their intention to resume or suspend tasks, Prism coordinates reconstructing the user-perceived state of the resumed task, or saving the state of the suspended task, as appropriate.

The Managed Environment layer (ME) is responsible for monitoring the availability of suppliers and resources, and for optimally matching the incoming requests from Prism to the available alternatives. Upon resuming a task, the ME maps the service requests to the concrete suppliers that best match the user preferences. While tasks are being carried out, the ME monitors the environment for failures and opportunities for improvement. Should an alternative configuration become more attractive, the ME is the first to reason whether to replace one or more suppliers to reach the desired configuration. A cost of change is factored into this reasoning, since users may perceive a cost whenever they are interacting directly with a supplier targeted for replacement.

The Aura framework takes advantage of the knowledge about user tasks and preferences captured by Prism to guide adaptation policies inside resource-aware applications. Designs that rely on ad hoc mechanisms inside applications to capture knowledge about user tasks make it hard to have a consistent view across applications, and to transfer that knowledge to a different set of applications when a task is resumed in another environment. In contrast, the Aura framework promotes a consistent system-wide awareness of user needs and preferences, and makes it easy to disseminate that knowledge to wherever it is needed.

Such design of the Aura framework made it easier to implement the ME layer, allowing for resource-aware applications coming out of complementary research to be cleanly integrated and controlled [9,10]. Implementing the ME layer also benefited from dovetailing with complementary research in algorithms for maximizing of utility functions [70], and from the collaboration of several students, under the coordination of Bradley Schmerl, who wrapped existing applications to serve as service suppliers in the Aura framework.

The infrastructure was tested on Windows and Linux, including the migration of user tasks between the two. The delay introduced by the infrastructure during the automatic configuration of the environment is typically less than 1 second, and therefore mostly imperceptible when coupled with starting up applications.

Chapter 4 elaborates on the Aura architectural framework, on the infrastructure that implements the framework, and on the evaluation of the infrastructure from a systems perspective.

Thesis Validation

A key component of evaluating this research is validating the thesis stated above. This entails demonstrating the following premises: that the proposed models of user tasks can be used to (i) scale task management in space, and (ii) in time; that an infrastructure that exploits such models (iii) enables users to take full advantage of computing environments, and that using that same infrastructure (iv) poses less overhead to users than configuring the environment themselves.[2]

To validate that the proposed approach supports scaling task management I built an infrastructure that does it. Specifically, with respect to the scalability in space, the infrastructure enables users to suspend tasks in one environment and to resume those same tasks in another environment, while addressing the challenges of heterogeneity and distribution. With respect to the scalability in time, the infrastructure enables users to access tasks long after their completion.

Validating that users are enabled to take full advantage of the surrounding computing

[2] The overhead of using the infrastructure is compared against users configuring the environment themselves because that represents the state of the art for scaling task management in space across heterogeneous environments (see Chapter 6).

environment is demonstrated by construction. The infrastructure for task management is endowed with the capability to find the best match between the user's needs and the available components and resources. Furthermore, the optimality of that match is continuously maintained in the face of dynamic changes both in user tasks and in the environment. Such optimality includes mechanisms for guiding fine-grain adaptation to resource variation, such as those provided by resource-adaptive applications.

Finally, validating that the infrastructure reduces the overhead for users is demonstrated by comparing the overhead of interacting with the infrastructure against the overhead of interacting with the raw environment. Specifically, we analyze the overhead for defining tasks, and for suspending and resuming those tasks, both with and without the assistance of the infrastructure.

Chapter 6 addresses validating the thesis in detail.

Contributions

This dissertation offers contributions at three levels: an *approach* to scaling task management in space and time, an *architectural framework* that supports the approach, and an *infrastructure* that implements the framework. Chapter 8 elaborates on the contributions summarized below.

The main contribution of my research is demonstrating that high-level models of user tasks can be exploited to address user mobility beyond traditional office environments, with significant advantages over other current approaches (see How Existing Solutions Fall Short, above). Specifically, the proposed *approach* provides a number of innovative and important capabilities:

– Scales in space, across heterogeneous environments, allowing users to suspend a task in one environment and resume it on another environment running a different set of applications and devices.
– Scales in time, allowing users to find and recover tasks defined or completed long ago.
– Reconciles two competing requirements: enabling users to take full advantage of the environments around them at different locations, while simultaneously reducing the overhead incurred by users when configuring computing environments.
– Offers users control over which tasks to access at which environments, thus enabling control over which environments are allowed to manipulate sensitive materials, and saving resources by activating only the services required by the tasks that users intend to work on.
– Accounts for user preferences with respect to alternative ways of carrying out their tasks, and with respect to quality of service tradeoffs.

The *architectural framework* that supports our approach clarifies the responsibilities and interaction protocols between the components of an infrastructure for task management. This architectural framework provides a number of innovative and important features:

– Defines a new software layer dedicated to gathering knowledge about user tasks, thus promoting system-wide awareness of user tasks and preferences.
– Defines a component dedicated to managing the environment based on abstract models of user tasks and on a global view of the environment, thus promoting coherent system-wide configuration and adaptation decisions.

- Optimally matches user needs and preferences to environment capabilities, using a utility-theoretic framework.
- Coordinates dynamic adaptation at three levels: fine-grain adaptation policies within resource-aware applications; adaptation to changes in the capabilities of an environment; and adaptation to changes in the requirements of user tasks.

The *infrastructure* that implements this framework accomplishes a number of important goals:

- Demonstrates the capabilities of the approach and the features of the architectural framework by providing a working implementation of an infrastructure for task management that is usable on a daily basis.
- Demonstrates that non-expert users can understand and manipulate the functionality delivered by the infrastructure.
- Provides a foundation for research that supports affordable integration of legacy applications, and that imposes a low buy-in cost to extensions to the infrastructure.

Plan of Dissertation

Chapter 2 compares the research presented in this dissertation with related work in desktop/task management, and more broadly with other work that represents and exploits models of user tasks. Since task management involves the automatic configuration and reconfiguration of the computing environment, we also compare our research with work in self-configurable systems, resource-aware systems, and context-aware systems.

The next three chapters address in turn the challenges associated with modeling user tasks, with building an infrastructure for scalable task management, and with providing that infrastructure with mechanisms for users to describe and operate on tasks. Chapter 3 focuses on the formal aspects of modeling user tasks as a coordinated set of services; of modeling user preferences with respect to the way their tasks will be supported in the environment; and of giving tasks a persistent semantic identity that supports browsing. Chapter 4 focuses on the part of the infrastructure facing the environment, and dedicated to configuring it. This chapter describes the architectural framework underlying the infrastructure: the responsibilities of the principal components, as well as their coordination. Additionally, this chapter describes and evaluates the current implementation of the infrastructure. Chapter 5 focuses on the part of the infrastructure facing the user, specifically on the mechanisms for users to describe and operate on their tasks.

Chapter 6 demonstrates how the proposed approach supports the thesis put forth in this dissertation. For that, it takes each of the premises underlying the thesis and argues how they are satisfied by the solutions presented in the previous chapters. Chapter 7 discusses important design and engineering decisions that we tackled during our research. It also discusses some important points that were only partially addressed, or left out of this dissertation, because of scoping considerations. Finally, Chapter 8 summarizes the contributions of this research and points at directions for future work.

Chapter 2

Related Work

This dissertation relates to work in two broad areas: first, research on representing and exploiting knowledge about user tasks; and second, research on self-configurable (adaptive) systems.

The first section below focuses on research related to user tasks. Specifically, we compare our work with other research on desktop/task management; with research on the related problem of assisting users to carry out complex tasks; and more broadly with research that represents and exploits models of user tasks.

The second section focuses on research on adaptive systems. Specifically, we compare our work with research on adaptation (a) to changes in service availability and quality of service, (b) to resource variability, and (c) to changes in the physical context surrounding the user.

User Tasks

This section compares our work with other research that employs models of user tasks, in one form or another. Specifically, the first subsection below focuses on task management, taken in the sense adopted in this dissertation, that is, of suspending and resuming tasks.

The second subsection focuses on the related problem of providing assistance to users on their tasks. The third subsection takes a broader perspective of the research areas where task models have been used, and compares the forms that such models take in each of those areas.

Task Management

The idea of desktop management was launched with Rooms, in 1987 [18]. Users aggregate sets of applications and information resources in virtual areas of the desktop, called “rooms.” To work on different tasks, users relocate between “rooms.” This idea originated in the world of desktop personal computers (workstations), where an underlying assumption is that users carry out their computer-supported tasks primarily on one machine. An additional limitation of this approach is that it does not easily scale to large numbers of tasks over extended periods. Busy users may intermittently touch on dozens of different tasks over the course of a workweek, and this strategy keeps all the applications consuming resources and cluttering the workspace presented to the user.

Early work in ubiquitous computing experimented with the idea that users are mobile, and may utilize available devices in their vicinity. That work uses OS-level mechanisms driven by location-sensing components to automatically “teleport” (make accessible) a user’s desktop to the nearest display within a smart space, such as an augmented home. Examples of this are 1994’s work on teleporting X Windows desktops [74]; and Microsoft’s Easy Living project, with results published in 2000 [15], where a set of smart rooms senses the location of users and migrates PC desktops, or simple tasks such as listening to music.

These two ideas, desktop management and user mobility, came together in 2000 with work that treated user tasks as a set of applications that is independent of a particular device. Examples of this are Georgia Tech’s project Kimura [57], where collections of applications migrate across displays within a smart room; and early work in Carnegie Mellon’s Project Aura [94], that targets the migration of user tasks across machines at different locations.

The following two years saw the publication of work that addressed making smart spaces amenable to cooperative tasks. Some of this work targeted generic office-like domains, for example Stanford’s ICrafter [71], and University of Illinois’ project Gaia [75]; while others targeted more specific domains, such as University of Washington’s Labscape project, which addresses conducting biology experiments in a lab [6], or University of Aarhus’ Pervasive Healthcare project, which addresses the work of healthcare professionals in a hospital [21].

This research shares with ours the goal of supporting task management for mobile users, where tasks may involve several services in the environment, and environments may contain heterogeneous devices.

However, this research supports user mobility by migrating applications, which are mapped to local devices. In contrast, our research introduces a new approach for supporting user mobility based on migrating high-level descriptions of user tasks, which are mapped to local applications.

Furthermore, this research either rebuilds operating systems from the ground up to support user tasks [75], or custom-builds, or at least significantly extends existing applications to work over custom-built infrastructures for distributed data exchange and code mobility [6,21,71]. In contrast, our research supports task management as a new software layer on top of existing operating systems, and accommodates the integration of legacy applications.

At the apex of minimizing changes to existing operating systems and applications is the Internet Suspend-Resume (ISR) project at Intel Research [54]. Here, one extension at the operating system level enables the migration of the contents of the whole virtual memory from one machine to another. This approach targets cases where users don't carry a personal computer around, but can use available "community" machines. However, such a solution is limited to cases where user tasks are supported by a single machine, and where users move only among machines with compatible hardware. In contrast, our approach makes no assumptions on the homogeneity of devices or software.

Furthermore, the ISR approach supports user tasks at a very coarse grain: everything that is active in a user's machine constitutes *the* user task, which is then migrated, as a whole, to the next machine. Although representing user tasks at such coarse grain requires no involvement from the user in discriminating which activities pertain to which task, it also offers no discriminating power on what the user actually wants to resume working on at the new location.

In contrast, our research explicitly represents which services and information resources (such as files) are involved in each task. Once that is known, our infrastructure automatically tracks the user-perceived state of those services (window sizes, cursors, etc.). Such capability enables users to discriminate which tasks they wish to resume at each location, and to swiftly switch among tasks (by activating/deactivating the corresponding services) at the same location.

Assistance with Tasks

In this dissertation, we focused on task management as the ability to suspend and resume tasks as a unit. A problem related to that, and that extends the scope of task management as we treated it herein, is providing assistance to users on their tasks.

Broadly, research on task assistance can be divided in two groups: one where the system guides or facilitates users in carrying out their tasks; and another where the system additionally may carry out tasks, or parts of tasks, on behalf of users.

An example of research on guiding users through tasks is the Adtranz train repair system [81], published in 1998, where the system guides technical staff through diagnosing problems, loads and presents relevant schematics, and facilitates communication with experts, as necessary. More recent research addresses daily life, often focusing on the needs of special groups, such as elderly people, or people in debilitated health situations. Examples are Georgia Tech's Smart Home [1], MIT's House_n [46], and the University of Rochester's Smart Medical Home [83].

Research on automated agents took task assistance one step further by enabling systems to carry out tasks on behalf of users. Examples of this are the RETSINA framework, with applications in domains such as financial portfolio management, ecommerce and military logistics [88]; and more recently Carnegie Mellon's RADAR project, which focuses on the office domain, automating such tasks as processing email, scheduling meetings, and updating websites [72].

For scoping reasons, providing assistance to users on the flow of complex tasks was not addressed in this dissertation (see Chapter 8, Future Work).

Task Modeling

A variety of research areas have addressed the broad problem of task modeling, with some work focusing on the modeling aspects proper (what to model), and other work on the process of capturing task models.

Research on task models can broadly be grouped into three categories. In the first category, tasks represent plans of actions to be carried out by an automated system, where an action corresponds to a computation or actuation of a mechanical device. Examples can be found in robotics, in distributed systems, and agent-based systems (e.g., [29,65,82]).

In the second category, tasks are carried out by humans, or by a mix of human and computers. Here, task descriptions play the role of guiding users along complex tasks. Examples can be found in the workflow modeling of business processes, and in some agent-based systems, where the description of the actions is in a form suitable to be interpreted by humans, such as “fill out form *x*” or “schedule a team meeting” (e.g., [6,36,81,100]). In these models, beyond the type and plan of actions there is little or no content to be exploited by a computer system. Examples can also be found in human computer interaction, where designers employ software usage models for analyzing and building computer systems, and often to constrain the interaction sequences allowed at run-time (e.g., [47,60,96,99]).

In the third category, task models represent the expectations and/or needs of users with respect to computational support for their tasks, which is an orthogonal problem to modeling task plans (first and second categories, above). Research in task management, namely this dissertation, focuses on this perspective of user models. Such models are either interpreted by automated tools at development-time to generate task-specific systems, or by computer systems at run-time to generate task-specific configurations.

The most significant related work in this third category is research in applications that automatically adapt their user interfaces to the characteristics of the available devices (e.g., [16,52,89]). Like work in tools for developing user interfaces [60], this work focuses on deciding which interaction widgets to bring up and how to place them on the screen. In contrast, our work focuses on deciding which high-level services (such as *editing text*) should be activated and which applications are best to supply them. Furthermore, while the success criteria of configuring user interfaces is the usability of the interfaces (which face tough competition from interfaces configured by human designers), in our work the success criteria is reducing the overhead of end-users in activating applications for their tasks (see the evaluation in Chapter 6).

With respect to the process of capturing task models, research falls into two groups. In the first group, task models are explicitly defined by a human. The research cited so far in this subsection, including our own, belongs to this group. Here, the human defining the models is typically an expert; except in task management, where end-users define their own tasks.

In the second group, the system includes functionality to learn models of user tasks, or to infer when users are carrying out specific tasks. Examples of this research can be found in the fields of Artificial Intelligence and Human Computer Interaction (e.g., [4,44,80,90,98]).

For scoping reasons, learning models of user tasks was not addressed in this dissertation (see Chapter 8, Future Work).

Adaptive Systems

This section compares our work with research on adaptive systems. The infrastructure for task management we developed for this dissertation is an adaptive system in the sense that it adapts to changes in user tasks and in the capabilities and resources in the computing environment.

The first subsection below focuses on systems that adapt to changes in service availability and quality of service (QoS). Examples of these are fault-tolerant and load-balancing systems. The second subsection focuses on systems that adapt to changes in resources, such as battery and bandwidth. Finally, the third subsection focuses on systems that adapt to changes in the physical context of users, such as user location and focus of attention.

Service Awareness

Fault-tolerant and load-balancing systems represent the earliest form of service-aware systems. Fault-tolerant systems react to component failure, compensating for errors using a variety of techniques such as redundancy and graceful degradation (e.g., [24,43]). Such systems have been prevalent in safety-critical systems or systems for which the cost of off-line repair is prohibitive (e.g., space systems, telecom, power control systems, etc.). Here the primary goal is to prevent or delay large-scale system failure.

Load balancing systems use quantitative models of QoS (typically response time) to dynamically assign requests to a pool of known servers (e.g., [14]).

More recent research in model-based adaptation puts these two ideas together: quantitative models of QoS to guide the adaptation policies, and the ability to hot-swap components. Typically, as in load balancing, the pool of available components is known at deployment time, or it is updated with human intervention. These systems use global system models, such as architectural models, as a basis for system reconfiguration (e.g., [20,37]).

Our research builds on work in model-based adaptation, adding the problem of dynamic discovery of services (and components that supply those services). Work in distributed systems and ubiquitous computing has addressed mechanisms for service discovery. Examples of such mechanisms are MIT’s INS, Sun’s Jini, and IETF’s SLP [3,5,79].

With the popularization of systems based on web services, there have been multiple efforts to normalize the way of describing and accessing web services. Examples of such efforts are DAML/OWL, UDDI, and WSDL [26,92,97]. In this work, however, the description of physical properties, such as the location of the component supplying a web service, is irrelevant.

In contrast, the location of a service supplier may be crucial in task management, if the user needs to interact physically with such a supplier. Our research shares with previous work in ubiquitous computing the requirement of describing physical properties of service suppliers [3].

However, the problem of representing and enforcing strategies to scope the search for service suppliers based on physical properties remains an open problem (see Chapter 8, Scoping the Environment).

Resource Awareness

Resource-aware systems react to resource variation: components adapt their computing strategies so they can function optimally with the current set of resources (bandwidth, memory, CPU, power, etc.). Many of these systems emerged with the advent of mobile computing over wireless networks, where resource variability becomes a critical concern [8,30,55,62,64].

While most of this research focuses on one component at a time, in contrast, our work tackles the problem of multi-component integration, configuration, and reconfiguration.[3]

Furthermore, the adaptation policies in such resource-aware systems are typically determined internally by each component, and often hard-coded. However, which adaptation policies are appropriate at each moment depends on the user preferences for the current task. And user preferences change dynamically, as users switch between tasks, or even in the middle of a task.

In contrast, our research associates user preferences to each task, and provides mechanisms to communicate those preferences dynamically to all the components supporting the task. Such preferences determine the appropriate resource allocation and adaptation policies.

For that, we build on previous work in mechanisms to determine the optimal resource scheduling and allocation among competing components, based on the requirement for the task (e.g., [33, 48,56,63,73]). Specifically, separate but complementary research to this dissertation developed the algorithms that we use [70], which in turn are based on Knapsack algorithms [69].

We also build on research on resource-aware systems that support programming interfaces to dynamically configure the appropriate adaptation policies (e.g., [9]).

Context Awareness

Context-aware systems react to variations in the physical context around users. Examples of observed variables are: user location, attention focus, physical activity (sitting, driving…), emotional state (relaxed, working, in distress…), privacy (who else is in the vicinity), etc. There is a considerable body of work in sensing such variables (e.g., [7,42,68,78]).

Typical context-aware systems represent such awareness as collections of interpreted rules (clauses of the form "if *context* then *action*"), or embedded logic in the code. Examples are found in research in ubiquitous computing (e.g., [17,41,95]).

[3] Although somewhat related, this kind of automatic configuration is distinct from the automatic configuration being investigated in other research [53]. There, configuration is taken in the sense of *building and installing* new applications into an environment, whereas here, it is taken in the sense of *selecting and controlling* applications so that the user can go about his tasks with minimal disruption.

While the architectural framework that we define accommodates context-aware applications, task management itself can be made context-aware. For instance, tasks can be suspended automatically when a user leaves the room; or a desired task, such as navigation assistance, can be resumed automatically when a user enters his car.

In our architectural framework, we allow for the definition of rules that constrain which tasks should be carried out on which environments (see Chapter 5, Tasks at a Glance). However, for scoping reasons, we did not specify the representation of such rules, nor incorporated mechanisms for context awareness in the infrastructure. For that, we would build on complementary research on mechanisms for delivering context information (e.g., [49]), which in turn build on research on context sensing (cited above).

Chapter 3

Modeling Tasks

The role of task models in this research is to enable the automatic configuration of computing environments. Task models capture what users need from the computing environment for each of their tasks (see also the discussion in Chapter 7, Sophistication of Task Models). These models are exploited at run-time by an infrastructure that automatically configures the environment on behalf of users.

This chapter focuses on the internal representation of task models, which is exploited by the infrastructure, while Chapter 5 addresses how these models may be viewed and constructed by users.

The task models adopted in this work address the three fundamental properties of scalable task management discussed in Chapter 1. First, task models provide a handle for the coordinated use of a set of services in the environment. Second, to address scalability in space, and in particular heterogeneity and resource variations in the environment, task models represent the user's preferences relative to features and Quality of Service (QoS). Third, to address scalability in time, task models establish an enduring identity that enables users to find tasks long after they are gone from the list of currently active tasks. The following sections discuss these properties in turn.

Coordinated Use of Services

A fundamental property in task management is that user tasks typically involve several applications and information resources. For instance, for preparing a presentation, a user may

task	An everyday activity such as preparing a presentation or writing a report. Carrying out a task may require obtaining a *configuration* of *services* from an *environment*, and accessing several *materials*.
configuration	Set of possibly interconnected *services* that together support a *task*.
service	Either (a) a service type, such as editing text, or (b) the occurrence of a service proper, such as editing a given document. For simplicity, we will let these meanings be inferred from context.
environment	The set of *suppliers*, *materials* and *resources* accessible to a user at a particular location.
supplier	A component (application and/or device) in the *environment* offering *services* – e.g. MS Word.
material	An information asset such as a file or data stream.
resource	What is consumed by *suppliers* while providing *services*. Examples are: CPU cycles, memory, battery, bandwidth, etc.
context	Set of human-perceived attributes such as physical location, physical activity (sitting, walking…), or social activity (alone, giving a talk…).
user-perceived state of a task	User-perceived set of properties in the *environment* that characterize the support for the *task*. Specifically, the user-level settings (preferences, options) associated with each of the *services* supporting the task, the *materials* being worked on, user-interaction parameters (window size, cursors…), and the *user preferences* for the task.
user preferences	*Task*-specific preferences with respect to alternative *configurations* for supporting the task, alternative *suppliers* to support a *service*, and user expectations towards quality of service (*QoS*).
QoS	Evaluation of properties (*QoS dimensions*) of a *service* perceived by a user while performing a *task*.
QoS dimension	An aspect of *QoS*, such as response time, accuracy, image resolution, frame rate, etc.

Figure 3.1 Summary of the terminology used in this dissertation.

edit the slides, refer to a couple of papers on the topic, check previous related presentations, and browse the web for new developments. Sometimes a task is supported by a loose collection of services; other times services may need to be interconnected, for instance to pipe data between them.

To provide a handle for the coordinated use of a set of services in the environment, task models need to represent three aspects: the set of services to be used, how they are interconnected, and the *user-perceived state* of the task (refer to Figure 3.1 for a summary of the terminology used in this dissertation).

The main challenge in representing these aspects is establishing the level of abstraction: task models must be abstract enough to be environment-independent, but precise enough to allow for the instantiation of the task using the concrete capabilities of the current environment. The following three subsections discuss the appropriate level of abstraction for the three aspects. The fact that the chosen level of abstraction is precise enough to support the automatic configuration of environments is addressed in Chapter 4. The fourth subsection presents a formal specification for the internal representation of these aspects of task modeling.

Identifying Services

Suppose that a user, Fred, is preparing a review of a video clip. While Fred works on his laptop, which runs MS Windows, viewing the clip is supported by Windows Media Player. However, when Fred reaches his office, he would like to take advantage of the large screen on his powerful desktop, which runs Linux. If Fred's task model includes the fact that he needs Media Player, the task cannot be instantiated on Fred's desktop. However, there is no good reason why, if Fred so desires, the video couldn't be played by whatever video player is available on Fred's desktop.

To account for the diversity in computing environments, the types of services required by a task are represented in abstract terms: *play video*, *edit text*, *edit slides*, etc. In this example, Fred's task would require two services: *play video* on the video clip, and *edit text* on Fred's notes. Whenever Fred reaches an environment and expresses his desire to resume the video review task, these services can be dynamically mapped to available suppliers that have the capability to provide them. Naturally, the infrastructure needs to share a vocabulary of service names, or otherwise be able to resolve name equivalences (see also the discussion in Chapter 7, Service Naming and Substitutability, and related work in Chapter 2, Service Awareness).

State Snapshot

Similarly to identifying services, the snapshot of the user-perceived state for each service is represented in abstract, application-independent terms. The user perceived-state includes things such as layout of windows, files being worked on, cursor positions and application settings. For example, for the *edit text* service, the snapshot would include the current *cursor position*, whether *spell checking* is currently activated, etc.

Furthermore, the representation of the user-perceived state must be such that it can be processed by applications with different degrees of sophistication. For instance, while finding a text editor that supports spell checking in a rich environment may not be a problem, a basic text editor running on a small platform might not support that feature, or even be aware of what *spell checking* means. Therefore, the format of the representation must be such that a given service supplier is able to extract the information it can recognize, without being thrown off by information it does not know how to handle.

The requirements of descriptive service names and of accommodating different levels of sophistication are addressed by adopting an XML-based representation of task models.

The modular nature of task models allows for independent dictionaries of terms to be maintained. Specifically, the component of the infrastructure that deals with mapping services to suppliers, needs to be aware of service names, but not of the terms used to describe the state snapshot of each service. Suppliers that offer, for example, a text editing service need to be aware of their own service name (*edit text*) and of the terms used to describe the state snapshot of text editing (*spell checking*, etc.), but not of the terms used to describe any other service.

Interconnecting Services

Although tasks are supported by a coordinated set of services, in many cases those services need not communicate with each other. For instance, in the task of preparing a presentation, each of the services interacts with the user, but not among each other: it is up to the user to manually transfer relevant snippets of information among them, as necessary.

However, in other cases the services need to be interconnected for the task to be adequately supported. For example, suppose that a user, Fred, needs to talk to a foreign speaker using real-time automatic translation. For that, Fred's task involves three services: speech recognition, (text) translation, and speech synthesis. The output of the speech recognizer needs to be piped to the input of the translator, and the output of the translator to the input of the speech synthesizer.

Task models support the establishment of (point-to-point) *connections* between service *ports* (in this context, just a generic term for input or output). This concept can be used while being oblivious of the concrete mechanisms used by the service suppliers: opening a session on a point-to-point *connector* (in a software-architectural sense), activating a publish-subscribe mechanism over an event bus (multi-point connector), etc. Under the premise of leaving the connector-specific knowledge with the infrastructure, task models represent connections by identifying the services to be connected and the port *types* on those services (see also the discussion in Chapter 7, Service Interconnection).

In this chapter we use a variant of BNF customized for the XML representation of task models. A specification in BNF is structured in rules. A rule defines the syntactic form of a symbol. The symbol being defined is called a *non-terminal*, and appears to the left of the `::=` sign. The syntactic form is characterized by the expression to the right of the `::=` sign. A syntactic symbol that is not further defined by a rule is called *terminal*. In BNF, alternative is denoted by a vertical bar, |, and parenthesis are used for grouping (the standard mathematical interpretation). Square brackets, [], and curly brackets, {}, are also grouping operators, with the added meaning that anything inside square brackets is optional, and anything inside curly brackets can be repeated zero or many times.

To simplify reading the specification, we drop the convention of surrounding non-terminal symbols with angle brackets. In the variant adopted herein, whether a symbol is a terminal or non-terminal is established by context (see below). Furthermore, since the task models are built on top of XML syntax, we augment the operators of BNF with the following:

```
E ::= t: A; C
```

defines a type `E` of XML elements with tag `t`, attributes `A`, and children `C`, where `t` is a terminal symbol, `A` is an expression containing only terminals (the attribute names), and `C` is an expression containing only non-terminals (the child XML elements). So, for instance the rule

```
Book = book: year ISBN; Title {Author}
```

allows the following as a valid element:

```
<book year="2004" ISBN="123">
  <title>...</title>
  <author>...</author>
  <author>...</author>
</book>
```

where the non-terminals `Title` and `Author` (with contents elided above) would have their own definition rules in the grammar, with `title` and `author` as the corresponding XML tags.

Figure 3.2 Summary of the variant of Backus-Naur Form used in this dissertation.

Specification

While the three subsections above discussed *what* goes into task models for representing the coordinated use of services, this subsection focuses on *how* we represent those models. Specifically, we introduce a grammar that defines the syntax of such models. We also provide an example model representing the the task of reviewing a video clip.

For the specification of the task models used in this dissertation, we follow a variant of the Backus-Naur Form (BNF, see for instance [45]) summarized in Figure 3.2. In this setting, a particular task model is a sentence allowed, or generated, by the grammar.

Figure 3.3 shows the grammar for modeling tasks as a set of possibly interconnected services. A task (model) is an XML element with tag `auraTask`, with one `id` attribute, and with one `Prefs` child, followed by an arbitrary number of `ServiceSnapshot`, `MaterialSnapshot`, and `Config` children. The id of a task is unique for each user. A task may be carried out using one of several alternative service configurations – see section User Preferences, below. Configuration names are local to each task model.

Services stand for concepts such as *edit text*, or *browse the web*, and materials are files and data streams manipulated by the services. A service may manipulate zero or many materials; for instance, text editing can be carried out on an arbitrary number of files simultaneously. That relationship is captured by the `Uses` clauses within the `Service` element. Service ids are local to each task model. Materials are given an enduring identity, which includes an id, unique for each user, and information such as where to find that material – a path in the file system, or a URL. More on this in section Task Identity, below.

```
Task   ::= auraTask: id;
       Prefs {ServiceSnapshot | MaterialSnapshot | Config}

ServiceSnapshot ::= service: id type;
       Settings
MaterialSnapshot ::= material: id;
       State

Config ::= configuration: name weight;
       { Service | Connection }

Service ::= service: id;
       {Uses}
Uses   ::= uses: materialId;

Connection ::= connection; id type;
       Attach QoSPrefs
Attach ::= attach: ;
       From To
From   ::= from: serviceId port;
To     ::= to: serviceId port;
```

Figure 3.3 Grammar for specifying task models.
The italicized symbols *`Settings`* and *`State`* are service- and material-specific, and are not defined herein. The symbols `Prefs` and `QoSPrefs` are defined in the next section.

The snapshot of the user-perceived state of the task is captured in the `Settings` and `State` elements. The `Settings` element captures the state that is specific to a service, and shared by all materials manipulated by that service, while the `State` element captures the state that is specific to each material. The boundaries between which attributes of the state snapshot are represented within each of these elements are somewhat arbitrary. Both these elements are not further defined in this specification, since their content is specific to the type of service or material. Such content is treated as a black box by all components of the infrastructure, except by those that provide the particular type of service, say text editors for *edit text*.

```
<auraTask id="34">
  <preferences>
    <service template="default" id="1"/>
    <service template="default" id="2"/>
  </preferences>
  <service type="play Video" id="1">
    <settings mute="true"/>
  </service>
  <material id="11">
    <state>
      <video state="stopped" cursor="0"/>
      <position xpos="645" ypos="441"/>
      <dimension height="684" width="838"/>
    </state>
  </material>
  <service type="edit Text" id="2">
    <settings>
      <format overtype="0"/>
      <language checkLanguage="1"/>
    </settings>
  </service>
  <material id="21">
    <state>
      <cursor position="31510"/>
      <scroll horizontal="0" vertical="7"/>
      <zoom value="140"/>
      <spellchecking enabled="1" language="1033"/>
      <window height="500" xpos="20" width="600" mode="min" ypos="100"/>
    </state>
  </material>
  <configuration name="all" weight="1.0">
    <service id="2">
      <uses materialId="21"/>
    </service>
    <service id="1">
      <uses materialId="11"/>
    </service>
  </configuration>
  <configuration name="only video" weight="0.7">
    <service id="1">
      <uses materialId="11"/>
    </service>
  </configuration>
</auraTask>
```

Figure 3.4 Example task model for reviewing a video clip.

The `Connection` element represents service interconnection as discussed in the previous subsection. Note that `Prefs` and `QoSPrefs` are defined in the section User Preferences.

Figure 3.4 shows an example task model for reviewing a video clip. This example was captured while running the infrastructure described in Chapter 4. The user defined two alternative configurations for this task: one including both playing the video and taking notes, the other, playing the video alone. Both services use a single material: *play video* uses a video file, with material id 11, and *edit text* uses a text file, with material id 21.

The user perceived state of the task is represented as the current service settings, under each service, and the current state of each material. For instance, the state of the video includes the fact that the video is stopped at the beginning (the cursor is set to 0 time elapsed), and it indicates the position and dimensions of the window showing the video.

Whenever a specific material is manipulated in more than one task, each task model keeps its own representation of the material's state. For instance, a spreadsheet with experiment results may be manipulated during the task of realizing the experiments, as well as during the task of writing a paper on the results. In the event of both tasks being active simultaneously, coordinating the access to the shared material is handled by the service suppliers, since that coordination depends on the type of access (read only, read-write…) as well as on the semantics of the service (for instance, on whether it makes sense to open separate views of the material).

User Preferences

An important property of user tasks is that they may span multiple locations. Prompted by advances in computing and networking, people have increased expectations towards the availability of computing. Furthermore, users may like to take full advantage of the computing environments accessible to them, much like they take advantage of the furniture in each space.

However, such computing environments may be very diverse. For instance, in a smart room at the office, a user may be surrounded by powerful devices running sophisticated applications; but at a coffee shop, the user may have to rely on his handheld and a flaky wireless connection.

The infrastructure can do a much better job at automatically configuring the environments on a user's behalf, if it knows what the user prefers in different circumstances. This section focuses on the internal representation of user preferences, which is exploited by the infrastructure to find the best match between what the user needs for a given task, and what the environment has to offer. Chapter 5 discusses how these models may be viewed and constructed by users

In this work, user preferences are expressed formally as *utility functions*. This enables turning the problem of finding the best match between what the user needs and what the environment has to offer into a maximization problem. Of course, that maximization is constrained by the availability of capabilities and resources in the environment.

To make preferences easier to both elicit and process, we split them (and their formal reification, utility functions) into three parts: first, *configuration preferences* capture preferences with respect to the set of services to support a task. Second, *supplier preferences* capture the desired features of the individual service suppliers; and third, *QoS preferences* capture the acceptable levels of quality of service and preferred tradeoffs.

Furthermore, we make a simplifying assumption: these aspects are modeled independently of each other. That is, the function for each aspect captures the user's preferences for that aspect independently of the others. For instance, a user, Fred, may state that he prefers response time to be under 3 seconds, and that he prefers supplier *A* to supplier *B*. However, under the independence assumption, Fred cannot express that he would like response times to be under 1 second whenever supplier B is used, and that he would be willing to wait up to 3 seconds when supplier A is used. This assumption has important simplification properties, both for the maximization algorithms, and for the elicitation of preferences (see also Chapter 5, Preferences and the discussion in Chapter 7, QoS Tradeoffs).

The possible values of utility, the *utility space*, provide a formal representation of how useful each alternative environment configuration is with respect to a specific task. In other words, utility is a measure of user happiness with respect to the possible outcomes of the configuration process. We encode utility in the interval [0, 1] of the real numbers, where 0 utility corresponds to the configuration being unacceptable for the task; and 1 corresponds to user satiation, in the sense that increasing the capabilities of the environment will not improve the user's perception of usefulness for the specific task.

The overall utility is given by the product of the three parts above. In semantic terms, if the user considers any of the utility parts to be inadequate (value close to 0) the overall utility will reflect the inadequacy. For the user to be satisfied (overall utility close to 1) all three parts need to be satisfactory. Note that by encoding each of the parts in the interval [0, 1], the overall utility falls within the same interval. The following three subsections address these three parts in turn, and the fourth specifies how these are represented in the task models adopted herein.

Configuration Preferences

Users may be willing to use different sets of services in different circumstances. For instance, suppose that a user, Fred, wants to watch the video broadcast of a sports event, but the network connection at the current location is especially poor. Fred may be willing to forsake playing the video in favor of allotting the meager bandwidth to playing only the audio with a fair quality. Suppose further that Fred needs to take notes on the video that he is watching. If there is a convenient microphone and speech recognition software, Fred prefers to dictate the notes, but he is otherwise willing to type the notes. During automatic configuration, configuration preferences play a key role in choosing among the alternative sets of services for supporting the task.

A task model may include more than one alternative set of services for supporting the task. In the example above, considering the video vs. audio option, and the dictating vs. writing option, there are four alternatives for supporting Fred's task.

Configuration preferences represent user happiness with respect to each alternative. Formally, let C_t denote the set of alternative service configurations for task t; for instance, the set containing *video&dictate*, *video&write*, etc. for Fred's task above. The configuration preferences are a discrete mapping between C_t and the utility space $U \cong [0,1]$:

Definition 3.1 $$h_{Config} : C_t \to U$$

In the example above, Fred may signal that he will be happy watching the video and dictating the notes, ok with writing them, and so forth, by setting h_{Config}(*video&dictate*)=1, h_{Config}(*video&write*)=0.7, etc. Eliciting configuration alternatives and their assigned preference is a topic for Chapter 6. Note that the mapping h_{Config} is given by the attributes `Config.weight` in Figure 3.3.

Supplier Preferences

Users may be willing to use different service suppliers in different circumstances. For instance, suppose that Fred starts reviewing the video at home, using MS Word as a supplier for editing his notes, and decides to resume that task at the office, where he has a desktop running Linux. If only Linux native text editors are available, say Emacs and Vim [38,93], Fred may prefer using Emacs to Vim (or vice-versa). During automatic configuration, supplier preferences play a key role in choosing among alternative components to provide a given service.

A task model includes supplier preferences for every service in the task. Supplier preferences may discriminate as many or as few specific suppliers as the user wishes, and must include the user happiness for a supplier *other* than those discriminated. This strategy covers the corner cases where the user may wish to specify that only one specific supplier is acceptable, or that *any* supplier will be acceptable. Supplier preferences also include the user happiness in the case that no supplier is found to provide the service. This supports comparing the utility of degraded modes of operation – when some of the services in the task cannot be supported in the current environment. (See also the discussion in Chapter 7, Supplier Preferences.)

Formally, let P_s denote the set of suppliers that the user cared to discriminate for service s, augmented with the values *other* and *none*. For instance, $P_{edit\ text}$ for Fred's task above might include MS Word, Emacs, Vim, *other* and *none*. The supplier preferences[4] for service s are a weighted discrete mapping[5] between P_s and the utility space $U \cong [0,1]$:

Definition 3.2 $$h_{Supp}^{w_s} \quad \bullet \quad h_{Supp} : P_s \to U, \quad w_s \in [0,1]$$

In the example above, Fred may signal that he clearly prefers Emacs over Vim by setting h_{Supp}(Emacs)=1 and h_{Supp}(Vim)=0.3. He may also signal that he is open to try other suppliers by setting h_{Supp}(*other*)=0.5 – in fact, that means that he prefers to try a non-discriminated supplier

[4] This framework can easily be extended to include preferences with respect to supplier warm-up time (the user may prefer a supplier that will be available sooner) and with respect to the cost of changing a supplier in the middle of a task [85].

[5] In the definitions throughout this dissertation read the large dot as "where," or "such that."

than to use Vim (of course, the opposite might be represented by flipping these values). Fred may also signal that taking notes is desirable but not crucial for his task by setting h_{Supp}(*none*)=0.2. Note that a value of 0.2 means that Vim will be activated if there is no other alternative. If the value for *none* were higher than the one for Vim, having no supplier would be marked as preferable to having Vim.

The weight w_s reflects how much the user cares about the choice of supplier for *s*. These weights are a convenient instrument to scale the preferences with respect to the choice of supplier for each service. By assigning w_s a low value (close to 0) the overall utility is desensitized to the choice of supplier for service *s*. Eliciting supplier preferences is a topic for Chapter 6.

QoS Preferences

Users may prefer to have different QoS tradeoffs for a given service in different tasks. For instance, suppose that Fred is watching a video over a network link and that the bandwidth suddenly drops. Should an adaptive video player reduce the frame rate, or the image quality? The answer depends on Fred's preferences for the current task. If Fred is watching a sports event, he may prefer frame rate to be preserved at the expense of image quality. For watching a documentary on painting, the opposite might be preferable.

Furthermore, the preferred QoS tradeoffs may change during the task. For instance, during a task that involves automatic translation of natural language, Fred may prefer faster responses over accuracy of translation during the introductory part of the conversation, but he may prefer the opposite once the conversation gets more involved.

While the simplest form of expressing a tradeoff is to indicate which dimension is preferred, this form has very limited expressive power. For instance, a user might indicate that response time is preferred over accuracy of translation. However, how short of a response time will satiate the user? And even if accuracy is less important, what if it degrades so much that the translations become unusable? A clearly more powerful form is to express the thresholds that characterize a tradeoff. For example, if Fred requires highly accurate translations, he may be willing to wait up to 30 seconds for an answer.

In this work, QoS tradeoffs are set by expressing user happiness with the level of quality provided along each QoS dimension. In the example above, when Fred prefers faster responses over accuracy of translation, the QoS preferences set stricter happiness thresholds for response time and looser thresholds for accuracy. The tradeoff can be reversed by relaxing the thresholds on response time and tightening the thresholds on accuracy.

QoS preferences play a key role in guiding the adaptation policies within resource-adaptive applications while users carry out their tasks [10]. Additionally, during automatic configuration, QoS preferences play a role in choosing among alternative service suppliers, and in determining the optimal resource allocation among the several service suppliers involved in a task [70].

$$F(x) = \frac{1}{1 + e^{\frac{good+bad-2x}{good-bad}}}$$

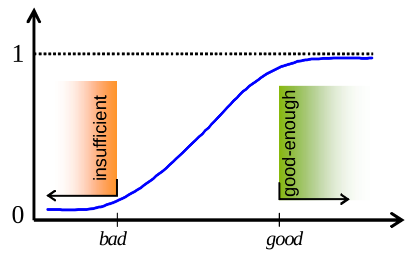


Figure 3.5 Generic Sigmoid shape

To play those roles, task models need to include the QoS preferences for the user-perceived QoS dimensions in every service and connection in the task. (Eliciting these is prevented from becoming a daunting chore by using templates – more on this in Chapter 6).

To make preferences easier to both elicit and process, we make two simplifying assumptions (see also the discussion in Chapter 7, QoS Tradeoffs). First, QoS preferences are modeled independently for each QoS dimension. Second, QoS preferences fall into two categories: those characterized by enumeration, and those characterized by numeric values. For QoS dimensions with an enumerated domain, for instance translation *accuracy*, with values *high*, *medium* and *low*, user preferences are encoded as a discrete mapping to the utility space.

For QoS dimensions with a numeric domain, for instance *response time*, user preferences are encoded using a predefined vocabulary of functions. The question then becomes which vocabulary of functions to choose in a continuum between generic mathematical functions, such as multiplication, exponentiation, etc., and a reduced set of functions. Supporting an arbitrary function has the advantage of being expressive. However it has two strong disadvantages: first, parsing and evaluating the functions are harder the more generic the vocabulary; second, and most importantly, it is very hard to elicit which arbitrary function represents user preferences. Choosing a restricted set of high-level functions makes both these aspects easier, but it brings up the research question of choosing an appropriate vocabulary.

As a working hypothesis, for QoS dimensions with a numeric domain, we distinguish two intervals: one where the user considers the quantity to be good enough for his task; the other where the user considers the quantity to be insufficient. Sigmoid functions characterize such intervals and provide a smooth interpolation between the limits of those intervals (see Figure 3.5). Sigmoids are easily encoded by just two points: the values corresponding to the knees of the curve; that is, the limits *good* of the good-enough interval, and *bad* of the insufficient interval.[6] The case of when less-is-better, e.g. response time, is just as easily captured as the case where more-is-better, e.g. accuracy, (as in Figure 3.5) by flipping the order of the *good* and *bad* values.

[6] Also amenable to this working hypothesis would be a piece-wise linear function *F*, where *F*(*x*) takes an arbitrarily small positive value for $x \leq bad$, $F(x)=1$ for $x \geq good$, and follows a linear interpolation between those two points.

```
Prefs ::= preferences: ;
      {ServicePrefs}
ServicePrefs ::= service: template id;
      [SupplierPrefs QoSPrefs]
SupplierPrefs ::= supplier:;
      Table
QoSPrefs ::= utility: combine;
      {QoSDimensionPrefs}
QoSDimensionPrefs ::= QoSdimension: name;
      Function
Function ::= Table | Sigmoid
Table ::= function: type weight;
      {Entry}
Entry ::= entry: x f_x;
Sigmoid ::= function: type weight;
      Thresholds
Thresholds ::= thresholds: good bad unit;
```

Figure 3.6 Grammar for specifying user preferences.

Formally, let Q_s denote the set of user-perceived QoS dimensions for service *s*. For instance, $Q_{translation}$ for Fred's task above would include *response time* and *accuracy*. Let *Dom*(*d*) denote the domain of QoS dimension $d \in Q_s$; for instance, *Dom*(*response time*) is the set of positive real numbers, scaled in seconds. The QoS preferences for service *s* are given by a family of weighted functions (one for each $d \in Q_s$):

Definition 3.3 $$h^{w_d}_{QoS\,d} \quad \bullet \quad h_{QoS\,d} : Dom(d) \rightarrow U, \quad w_d \in [0,1]$$

Where $h_{QoS\ d}$ is either a discrete mapping or a sigmoid, as discussed above. The weights w_d reflect how much the user cares about the quality along dimension *d*. These weights are a convenient instrument to emphasize the preferred QoS: by assigning w_d a low value (close to 0) the overall utility is less affected by variations of quality along dimension *d*.

Specification

Figure 3.6 shows the grammar for modeling user preferences (see also Figure 3.3). The top element, tagged `preferences`, contains a child for the preferences of each service in the task. Service preferences are identified by (service) `id`, and in their simplest form they mention the `template` to be applied and have no children. If the value of the `template` attribute is `custom`, then service preferences will have both `SupplierPrefs` and `QoSPrefs` children.

Supplier preferences are defined as a table (discrete mapping) with an arbitrary number of entries: pairs <`x`, `f_x`>, with `x` in P_s, and `f_x` in *U*. This table defines h_{Supp} in Definition 3.2. The attribute `weight` corresponds to w_s, and `type` is necessarily `table`.

QoS preferences have as many `QoSDimensionPrefs` children as QoS dimensions for the service, corresponding to the family of functions $h_{QoS\ d}$ in Definition 3.3. Note that `Function` can either be a `Table` or a `Sigmoid`, which is indicated by the value of the attribute `type`, and that the attribute `weight` corresponds to w_d.

```
<preferences>
  <service template="custom" id="1">
    <supplier>
      <function type="table" weight="1">
        <entry x="RealPlayer" f_x="0.7"/>
        <entry x="MediaPlayer" f_x="1"/>
        <entry x="none" f_x="0.001"/>
        <entry x="other" f_x="0.5"/>
      </function>
    </supplier>
    <utility combine="product">
      <QoSdimension name="frameRate">
        <function type="sigmoid" weight="1">
          <thresholds good="20" bad="5" unit="fps"/>
        </function>
      </QoSdimension>
      <QoSdimension name="compression" type="float">
        <function type="sigmoid" weight="0.5">
          <thresholds good="80" bad="30" unit="percent"/>
        </function>
      </QoSdimension>
      <QoSdimension name="audio">
        <function type="table" weight="1">
          <entry x="on" f_x="1"/>
          <entry x="off" f_x="0.001"/>
        </function>
      </QoSdimension>
    </utility>
  </service>
  <service template="default" id="2"/>
</preferences>
```

Figure 3.7 Example custom preferences for service *play video* in Figure 3.4.

Figure 3.7 shows an example of custom preferences (not defined by a template) for the service *play video* in Figure 3.4. Note that the supplier preferences discriminate two suppliers, RealPlayer and Media Player, with a preference for the latter. Note also that the QoS preferences discriminate three dimensions: frame rate and video compression, defined as sigmoids, and audio quality, defined as a table.

A task model such as the one above, defines configuration preferences, supplier preferences, and QoS preferences, with semantics given by Definition 3.1 through Definition 3.3. This model is the basis for calculating the utility of each alternative configuration of services (more on this in Chapter 4, Finding the Best Match).

Task Identity

In addition to spanning multiple locations, another important property of user tasks is their duration and recurrence. Users may work on some tasks for days or even months, and tasks may need to be resumed after users thought they were done. For instance, a user may need to prepare periodic reports, or he may need to find a specific report that he wrote last year.

```
TaskInfo ::= task: state id;
       Description History Links

Description ::= description: ;
       Name Notes Collaborators
Name   ::= name : ;
Notes  ::= notes : ;
Collaborators ::= collaborators: ;

History ::= history: due created;
       {Accessed}
Accessed ::= accessed: at stop start;

Links  ::= links: ;
       {Link}
Link   ::= link: label tId;
```

Figure 3.8 Grammar for specifying information about a task.

Users should be able to refer to their tasks after they are gone from their desktop. Note that this is different than finding a file that resulted from carrying out a particular task. Users should be able to find a task itself, so that it can be reactivated, if necessary, or used as a template to create a similar task. For example, a user, Fred, may want to update a report that he prepared the previous year with some new data. For that, Fred finds and reopens that task. By doing that, Fred not only has access to the file containing the report proper, but also to all the services and sources of information that Fred originally used for preparing the report, laid out in the exact same way as Fred last used them.

However, users should not be expected to remember a specific task identifier, a precise classification, or when exactly the task was carried out. Today, if a user needs to find a file containing “the report on the trip to Pittsburgh that I wrote last year,” he may not remember the name of the file and may have no idea under which directory it was filed. However, modern operating systems support scanning the file system based on a partial file name, or based on some piece of file content. In fact, the metaphor of finding information based on pieces of that same information is being explored with great success by the World Wide Web.

In this work, task models carry information *about* the task. This may include a name, due date, the purpose or goals for the task, who collaborated[7] on it, links to other tasks, etc. Task links are references to other tasks, and, just like hyperlinks in a document, allow the quick navigation between related tasks. None of these items are mandatory or have to be unique, including the task name. Users are free to enter as much or as little information about their tasks as they feel appropriate: for short-lived tasks unlikely to be referred to after completion, users may enter no information at all. In addition to user-provided information, task models also gather automatically harvested information such as the log of when and where the user worked on each task.

[7] Like all other aspects of the information, this is purely informative: recall that in this dissertation we are not addressing the coordination issues of Computer-Supported Cooperative Work (CSCW).

```
<task state="pending" id="34">
  <description>
    <name>review semifinals game</name>
    <notes>commentary on the European Soccer Championship games
           for the company newsletter</notes>
    <collaborators>Barney;</collaborators>
  </description>
  <history due="2/07/04" created="30/06/04 9:51 PM">
    <accessed at="home" stop="30/06/04 10:32 PM" start="30/06/04 9:53 PM"/>
    <accessed at="office" stop="1/07/04 10:03 AM" start="1/07/04 9:27 AM"/>
  </history>
  <links>
    <link label="team A previous game" tId="27"/>
    <link label="team B previous game" tId="23"/>
  </links>
</task>
```

Figure 3.9 Example information about the task of reviewing a video clip in Figure 3.4.

Storing such information about tasks enables users to search for their tasks later on. Much like an Internet search, users can search for their tasks based on *anything* they remember about those tasks. One can think of each term in the information about a task as enabling one classification scheme. Each such classification scheme distinguishes all the tasks referring to that term from the tasks that don't. This is in stark contrast with the single hierarchical classification scheme currently offered by the directory structure in file systems, and much closer to the approach used by web search engines. Eliciting information about tasks, and searching based on that information is covered in Chapter 5.

Figure 3.8 shows the grammar for specifying information about a task. The two attributes are the task `state` (either *pending* or *closed*, i.e. completed) and the task's internal `id`, for cross-referencing (the same as in Figure 3.3 and in the task links, below). The task description holds three free form text elements containing the optional task name, notes and collaborators (a semicolon-terminated list of names). The `Links` element contains user-specified labeled links to related tasks: the `label` pertains to the directional link, i.e., to the relationship between the present task and the referenced one. The `History` element contains mostly automatically harvested information: the timestamp of task creation and the log of accesses (when and where the user worked on the task). The due date is user-specified.

Figure 3.9 shows an example of information about the task of reviewing a video clip, which appeared in Figure 3.4. The task represented in this example would match queries such as "soccer semifinals," or "newsletter July 04." See Chapter 5, Finding Tasks for details about expressing queries.

Chapter 4

Infrastructure for Task Management

The role of the infrastructure for task management is to support the notion of user tasks as coordinated sets of services. In the previous chapter, we introduced task models that capture what users need from the computing environment for each of their tasks.[8] These models are exploited at run-time by the infrastructure for automatically configuring and reconfiguring the environment on behalf of users.

To play this role adequately, the infrastructure needs to address the challenges discussed in Chapter 1. First, in the context of task management, users constantly switch between tasks, incrementally change their needs for the ongoing task (e.g., by adding or removing services) and may change their Quality of Service (QoS) preferences in the middle of a task. Therefore, the infrastructure needs to address adaptation to dynamic changes both in the environment and in the ongoing user tasks.

Second, scaling task management in space implies dealing with distribution and heterogeneity of devices and software in the environment. Allowing users to take advantage of the computing systems accessible to them in different locations implies that the infrastructure has to accommodate devices ranging from mobile phones, to personal computers, to smart rooms; and has to deal with the plethora of software that comes with those devices. Furthermore, the distribution of computing environments implies that the infrastructure needs to handle variations of resources and service availability as a normal situation, rather than exposing it to the user as exceptional (faulty) behavior.

[8] Refer to Figure 3.1 for terminology.

Third, to enable users to take full advantage of the capabilities of the environment, the infrastructure needs to find the best match between the user's needs and what the environment has to offer. To accomplish that, it is not enough to have a binary model of whether applications are working correctly or there is a fault. The infrastructure needs a quantitative framework to evaluate the alternative ways to configure the environment for supporting a task, both during the initial configuration of the environment, and during the ongoing adaptation to changes.

Fourth, the infrastructure needs to be designed in such a way that the quantitative models are exploited consistently to drive both system-wide configuration and adaptation, and local adaptation policies within resource-aware applications.

The rest of this chapter discusses an architectural framework for building infrastructures for task management: the Aura framework. The four sections below walk through the architectural decisions that address each of the challenges above, and that are embodied in this framework. The fifth section presents and evaluates a working implementation of the Aura framework.

Two Sources of Change

The computing environment around mobile users changes constantly. Each time users resume their tasks in a new location, they may find a different set of devices and software to work with. Even when users remain on the same location, resources may change. For example, in heavily networked environments, remote servers constantly change their response times and even availability. When using mobile devices, resources such as battery and bandwidth may fluctuate widely.

In addition, users' needs change as they switch between tasks, update the required set of services and materials, or adjust their QoS preferences for an ongoing task. For example, while browsing an e-commerce site over a poor connection, the user may want to skip loading pictures in favor of faster response times, but he may be willing to wait for the pictures to load once he reaches the page with the desired product. An example of adding services: as a task of preparing a report progresses, the user may add working on a spreadsheet, or he may add browsing the web for newly discovered sources of data.

The infrastructure for task management can be seen as an adaptive system. Typical adaptive systems hold a model of the universe of discourse. Those systems continuously monitor that universe, and act on it in order to optimize some goal function. Here, the universe of discourse has two parts that evolve independently of each other and that the infrastructure monitors: user tasks and environment. The purpose of the infrastructure is to maximize the utility of the environment with respect to the user preferences (see Finding the Best Match, below) by acting on (configuring) the environment.

To automatically configure the environment, first, the infrastructure needs to know *what* to configure for; that is, what users need from the environment to carry out their tasks. Second, the infrastructure needs to know *how* to best configure the environment: it needs to know which

layer	*mission*	*roles*
Task Management (Prism)	***what*** does the user need	– monitor the user's task, context and preferences – map the user's task to requests for services in the environment – represent complex tasks: decomposition, plans, context dependencies
Managed Environment	***how*** to best configure the environment	– monitor environment capabilities and resources – map service needs, and user-level state of tasks to available suppliers – continuously optimize the utility of the environment relative to the task
Environment	support the user tasks	– monitor relevant resources – manage fine grain QoS/resource tradeoffs

Figure 4.1 Software layers of the Aura framework.

capabilities and resources are available in the environment, and it needs mechanisms to optimally match those to the user needs.

In the Aura framework, each of these two problems is addressed by a distinct software layer: (1) the Task Management layer determines *what* users need from the environment at a specific time and location; and (2) the Managed Environment layer determines *how* to best configure the environment to support user needs.

Figure 4.1 summarizes the roles of these software layers and also shows a third layer, the Environment, which contains the applications and devices that support user tasks. Configuration issues aside, these applications interact with the user in the same way as they would without the presence of the infrastructure. The infrastructure steps in only to automatically configure those applications on behalf of the user.

Figure 4.2 shows the top-level components within these layers and the connectors[9] between them. In the Task Management layer, *Prism* acts as a user proxy, coordinating the suspending and resuming of user tasks. The *Context Observer* monitors the physical context of the user (location, etc.) and reports relevant events back to Prism, the *Environment Manager* (EM), and context-aware applications. In the Managed Environment layer, the EM offers the mechanisms to configure the services required by user tasks. The *Suppliers* offer the abstract services that tasks are composed of: *text editing*, *video playing*, etc.

The main focus of this dissertation is on Prism and its interactions with the Environment Manager and the Suppliers. For scoping reasons, incorporating context-awareness into task management was only marginally addressed in this dissertation (see Chapter 2, Context Awareness and also Chapter 5, Tasks at a Glance).

The following subsections elaborate on the roles of the components in Figure 4.2 and on the rationale for the interactions between them. Appendix A provides concrete scenarios of interaction and a formal specification of the protocols of interaction.

[9] In architectural terms, a *connector* is a model of the interaction protocols and a set of design constraints on the mechanism of communication (e.g., synchronous vs. asynchronous).

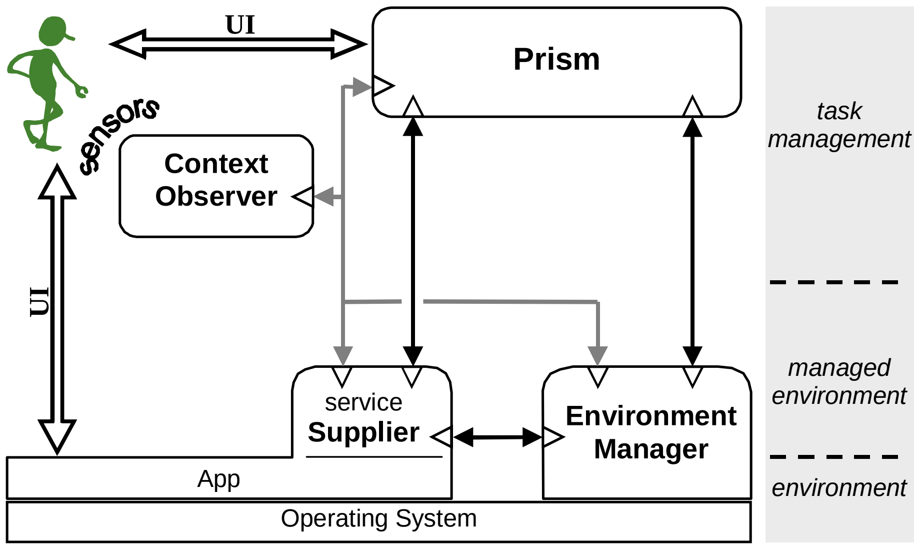


Figure 4.2 Component and connector view of the Aura architectural framework.
The layers on the right-hand side correspond to the software layers introduced in Figure 4.1.

Changes in User Tasks

Prism plays the main role in adapting to changes in user tasks and preferences. Prism holds knowledge about user tasks and preferences in the form of the models presented in Chapter 3. Such knowledge is used to coordinate the configuration and reconfiguration of the environment upon changes in user needs.

For instance, when a user is authenticated in a new environment, Prism coordinates accessing all the information related to the user tasks, and cooperates with the Managed Environment layer to find the best match for the user needs. Prism also monitors indications from users to know when a user intends to resume a task, or to suspend a task being carried out.

Upon getting indication to suspend a task, Prism captures the user-perceived state of the task for later use. When the user indicates that a task should be resumed, Prism coordinates reconstructing the user-perceived state of the task. Likewise, when a user modifies the set of services involved in an ongoing task, Prism saves or reconstructs the state of the dismissed or added services, as appropriate. Furthermore, Prism communicates the user's QoS preferences to resource-aware service suppliers, so that they can enforce the appropriate adaptation policies (see Adaptation at Three Levels, below).

The Task Management layer may also capture more complex representations of user tasks including task decomposition (e.g., task A is composed of subtasks B and C), plans (e.g., C should be carried out after B), and context dependencies (e.g., the user can do B while sitting or walking, but not while driving). However, for scoping reasons discussed in Chapter 7, Sophistication of Task Models, complex representations of user tasks such as these are not covered in this dissertation.

Changes in the Environment

The Managed Environment (ME) layer plays the main role in adapting to changes in the environment. The ME layer is responsible for monitoring the availability of suppliers and resources, and for optimally matching the incoming requests from Prism to the available alternatives. Upon resuming a task, the ME maps the service requests to the concrete suppliers that best match the user preferences (see Finding the Best Match, below).

While a task is being carried out, an alternative configuration may come to offer a better match than the current configuration. This may happen either because (a) resource variations degraded the observed QoS, or some supplier failed (which can be thought of as degrading the QoS all the way to zero); or because (b) some suppliers became accessible or resource variations made them more attractive in terms of forecast QoS.

Whenever an alternative configuration becomes more attractive, the ME layer is the first to reason about whether to replace one or more suppliers to reach the desired configuration. A cost of change is factored into this reasoning, since users may perceive a cost whenever they are interacting directly with a supplier targeted for replacement. Of course, if the supplier in question failed, that cost is unavoidable, and the ME layer should proceed to activate the best alternative supplier anyway.

If, on the other hand, the current configuration is functional but the alternative is attractive despite the cost of change, the ME layer may coordinate with Prism on whether and when to perform the swap. For instance, if the supplier playing a video is about to be replaced, the user may wish to finish viewing the current scene; or if a supplier supporting taking notes is about to be swapped, the user may wish to finish his train of thought. Prism would take these decisions either based on its knowledge of the user tasks or by prompting the user for a decision. See [85] for details about representing reconfiguration policies and cost of change.

Heterogeneity and Distribution

To support user mobility, the framework must accommodate the distribution and heterogeneity of computing environments. Allowing users to take advantage of the computing environments in different locations implies that the infrastructure has to interact with diverse software and devices. Furthermore, the increasing pervasiveness of smart spaces is causing a shift in the paradigm of computer use: from single-device, tightly integrated interaction (e.g., on a desktop computer), to multiple-device, loose interaction.

The designs of the Managed Environment (ME) layer and of the connectors represented in Figure 4.2 play a key role in addressing the heterogeneity and distribution of environments. The following subsections describe how these challenges are addressed in the Aura framework.

Heterogeneity

The ME layer holds abstract models of the environment. These models provide a level of indirection between the user's needs, expressed in environment-independent terms, and the concrete capabilities of each environment. This indirection is used to address heterogeneity: when a user needs a service, such as *speech recognition*, the ME layer finds and configures a supplier for that service among the ones available in the environment. Note that by virtue of these abstract models, Prism is aware of *which* services are available in each environment, but not *how* (i.e., by which applications) those services will be delivered.

The role of mapping user needs into the concrete capabilities of the environment is played by a generic (environment-independent) component within the ME layer: the Environment Manager. Specifically, the Environment Manager (EM) constructs abstract models of the environment and interacts with Prism for matching user needs with the available suppliers. (See Verifying the Protocols of Interaction, below.)

The ME layer also translates the environment-independent models of user-perceived state and QoS preferences issued by Prism into the specific capabilities of each supplier. While holding abstract knowledge about the environment is a generic role, interfacing with the specific capabilities of each supplier is, of course, supplier-specific.

The supplier-specific translation of task models is made by the *Supplier* components (see Figure 4.2). In each environment, the ME layer holds many Suppliers, corresponding to the applications and devices in the environment.

In practice, most Suppliers are implemented by wrapping existing applications to conform to the infrastructure's APIs. Rather than requiring writing a new portfolio of applications, this approach makes it easy to integrate legacy applications into the infrastructure. For instance, in a Unix-based environment, Emacs, and Vim may each be wrapped to become a supplier of *text editing* services; in a Windows-based environment, MSWord and Notepad may each be wrapped to supply the same service (see Chapter 7, Software Engineering of Service Suppliers).

The capitalized term *Supplier* refers to the wrapper code (residing in the ME layer). That code presents the infrastructure with a normalized way to access all the functionality necessary to configure the specific service supplier: to activate and deactivate the service, to capture and reconstruct the user-perceived state, and to enforce the resource-adaptation policies that derive from the QoS preferences.

Distribution

Computing environments are increasingly distributed. The Suppliers, especially, may be scattered across different devices, some of which may be remote to the user's location. Connectivity may vary widely, from high-speed wired connections, to fluctuating wireless (radio or infrared) connections.

Because of distribution, the design of the connectors in Figure 4.2 is crucial to determine the infrastructure's resilience and ability to be proactive. Specifically, we targeted three goals: the responsiveness of components should not be hindered by blocking on communication; whenever a component generates information that is relevant to others, it should pass it with no need to wait for a request; and to facilitate proactive reconfiguration, the EM should keep models of the environment that are as up-to-date as possible.[10]

The responsiveness of the infrastructure is critically influenced by the choice for the modality of communication: synchronous vs. asynchronous, client-server, etc. In synchronous communication, the originating (calling) component blocks on the reply of the target (called) component. However, in our case, each component needs to play its role, in real-time, doing the best it can with the available information and without blocking on another component's reply.

For example, the EM should not stop monitoring the capabilities of the environment, or replying to requests of Prism on account of being blocked on the reply of a remote Supplier – which may have become disconnected. Likewise, Prism should not stop responding to changes in the user's task when waiting for the reply of some other component.

Therefore, all communication between the components in Figure 4.2 is asynchronous (non-blocking).

Furthermore, whenever a component generates information that is relevant to others, it should have the ability to communicate it immediately without having to wait for a request. For example, when Prism first needs to resume a task, it requests the EM to find the best match of Suppliers in the environment. However, if later the environment changes in a way that justifies a reconfiguration, the EM may take the initiative of coming back to Prism suggesting the reconfiguration, or just informing Prism that it performed the reconfiguration, depending on what was agreed for the particular Supplier.

Rather than relying in push or pull models with strict timings, the communication between the components in Figure 4.2 is peer-to-peer (any component may take the initiative).

Additionally, to facilitate proactive reconfiguration, the EM needs to be aware of the state of the environment. Specifically, the EM needs to make sure that the Suppliers actively supporting user tasks are up and running.

The Aura framework puts the burden of proof on the Suppliers: they must emit a "heart-beat" signal to the EM reporting on the level of QoS being achieved with the current resources. In the absence of "heart-beat" (discounting network delays and losses) the EM assumes that the Supplier failed and proceeds to reconfiguration. The QoS reports are also used by the EM to periodically evaluate alternatives against the current configuration (more in Finding the Best Match, below).

[10] Defining meaningful strategies for scoping the environment is a crucial and open problem: see Chapter 8, Future Work.

Verifying the Protocols of Interaction

An important step towards assuring that the infrastructure behaves as intended is verifying the protocols of interaction between the components in Figure 4.2. Because the protocols follow an asynchronous, peer-to-peer modality (see above) it is important to ensure that they are deadlock free, and also to verify liveness conditions expressing that the components are able to recover from faults.

Specifically, we specified the interactions between Prism and the EM, between Prism and the Suppliers, and between EM and the Suppliers using Finite Sequential Processes, FSP [59], a process algebra similar to Hoare's CSP. We then used the LTS automatic checker [59] to verify the desired properties. Process algebras such as FSP allow for the verification of the sequence of interactions but fail to capture complex state and how it is affected by the interactions.

To reduce chattiness, the communication between Prism and the EM is session oriented. Specifically, for each task that the user may want to work on, Prism starts a session with the EM. Each session keeps as state the service definitions and user preferences, as defined in Chapter 3. Naturally this state can be updated incrementally, as users add or remove services from their tasks. Thanks to the state kept by the EM for each session, Prism may issue budget requests for alternative configurations (see Finding the Best Match, below), or reconfiguration requests, just identifying the services by id thus avoiding repeating the service definitions and preferences in each request.

To clarify the state shared by Prism and the EM for each session, and how it is affected by each interaction, we specified it using the Zed specification language and verified its consistency using the Zed checker [87]. These models proved a valuable tool during the low-level design and implementation of the EM.

See Appendix A for both the FSP and Zed specification of the protocols.

Finding the Best Match

To enable users to take full advantage of the capabilities of the environment, the infrastructure needs to find the best match between the user's needs and what the environment has to offer. However, the set of services to be configured in the environment is not always uniquely determined. In fact, users may have several tasks on which they are willing to work.

For instance, a user, Fred, may be willing to take notes on a promotional video; but if the video cannot be played with adequate fidelity, maybe because of insufficient bandwidth, Fred may be willing to work on his weekly report instead. Additionally, each task may have more than one way of being supported. For instance, Fred may dictate, type, or write on a pad for taking notes on the video.

To assist users in deciding which task to work on, and to find the best alternative configuration to support that task, the infrastructure performs a quantitative evaluation of all the alternatives. The two top layers described in Figure 4.1 cooperate in this analysis.

Prism generates the alternatives for *what* a user may want, while the EM evaluates *how well* the environment can support each alternative. For each alternative configuration within each possible task, Prism generates a *budget* request to the EM. That request contains the model of the configuration, in the form defined in Figure 3.3, and of the user preferences, in the form defined in Figure 3.6. The quantitative evaluation of each alternative is supported by the notion of *utility* (see Chapter 3, User Preferences).

The utility of a configuration *c* depends on the supplier assigned for each service, and on the levels of quality for each QoS dimension. For instance, in Fred's video review example (see Chapter 3, Supplier Preferences) the configuration *video&write* will have a different utility depending on whether Emacs or Vim are chosen as the supplier for the *edit text* service. Also, the utility, as Fred perceives it, depends on the levels of quality observed at each moment. If, because of fluctuating bandwidth, the video player reduces the frame rate below Fred's happiness threshold (the *good* value of the sigmoid), the utility for *c* decreases.

Formally, let S_c denote the set of services and connections in configuration *c*, and P_c denote the union of the sets of possible suppliers P_s for each $s \in S_c$. Let $p{:}S_c \rightarrow P_c$ denote one particular supplier assignment for each $s \in S_c$.[11] Also, let D_c denote the union of the sets of QoS dimensions D_s for each $s \in S_c$, and Q_c denote the union of the quality domains $Dom(d)$ for each $d \in D_c$. Let $q{:}D_c \rightarrow Q_c$ denote an observation of the levels of quality for each $d \in D_c$. The overall utility of the environment for configuration *c* is given by:

Definition 4.1
$$U(c|p,q) \mathrel{\hat{=}} \prod_{s\in S_c} h_{Supp}^{w_s}(p(s)) \cdot \prod_{d\in D_c} h_{QoS\,d}^{w_d}(q(d))$$

where the user preferences *h* are given by Definition 3.2 and Definition 3.3. Combining the user preferences by multiplication corresponds to an *and* semantics: overall utility is good, only if each and every preference can be met satisfactorily.

To maximize the utility of the environment, the EM explores all possible supplier assignments to the services in the task, and all possible quality levels that are achievable with the current resources. Formally, given a budget request for configuration *c* and a forecast of the available resources in the environment, the EM determines the supplier assignment, $\hat{p}$, and the forecast levels of QoS, $\hat{q}$, that maximize the utility:[12]

Formula 4.2
$$\underset{\substack{p:S_c\rightarrow P_c \\ q:D_c\rightarrow Q_c}}{\arg\max} \; U(c|p,q)$$

The algorithms involved in solving Formula 4.2 come from research by Vahe Poladian that is separate, but complementary to this dissertation [70]. Other research addresses forecasting available resources (e.g., [62]).

---

[11] As a technicality, if no supplier preferences are elicited for connections, as it is the case with the present syntactic form, that is, $P_s=\varnothing$ if *s* is a connection, set h_{Supp} trivially to 1.

[12] Read the vertical bar as "given." For instance $U(c|p,q)$ is read: the utility of configuration *c*, given a supplier assignment *p* and forecast levels of QoS *q*.

level	*adapts to*	*time scale*
Task Management	– changes in user tasks and preferences	minutes
Managed Environment	– changes in service availability and trends in QoS	seconds
Applications	– changes in resources	milliseconds

Figure 4.3 Adaptation role of each level.

The last step in computing the *feasibility* of a task belongs to Prism, which weighs the utility of each alternative configuration of services, by the user's preference for that configuration. Formally, the achievable utility of the environment for configuration *c*, $U\left(c \middle| \hat{p}, \hat{q}\right)$, is returned to Prism, which then computes the preferred configuration, $\hat{c}$, among the alternatives C_t for task *t*, using the configuration preferences in Definition 3.1:

Formula 4.3
$$\arg\max_{c \in C_t} \quad h_{Config}(c) \cdot U\left(c \middle| \hat{p}, \hat{q}\right)$$

Prism then advises the user (more on this in Chapter 5) on the feasibility of each task t, given the current conditions of the environment. The feasibility of a task *t* is defined as:

Definition 4.4
$$F(t) \hat{=} \quad h_{Config}(\hat{c}) \cdot U\left(\hat{c} \middle| \hat{p}, \hat{q}\right)$$

As discussed in the previous sections, the EM periodically evaluates the utility of the current configuration, feeding the levels of QoS reported in "heart-beat" messages from the Suppliers into Definition 4.1. The EM then runs the maximization in Formula 4.2 over the alternative supplier assignments and resource allocations, and decides whether a reconfiguration should be considered. More on this in the next section.

Adaptation at Three Levels

An important problem is to coordinate the adaptation policies enforced within resource-aware applications with the system-wide configuration and reconfiguration carried out at the Task Management and Managed Environment layers of the infrastructure.

Existing sophisticated applications are able to change their internal behavior to make the most of the available resources. For instance, a virtual reality application with strict timing constraints may use sophisticated graphics rendering algorithms when CPU is plentiful, but simpler algorithms when CPU is scarce. Furthermore, adaptation strategies may include the dynamic reconfiguration of distributed components. For instance, an adaptive natural language translator running on a handheld may run sophisticated algorithms on a remote server when bandwidth is plentiful, but may have to rely on simpler local algorithms when the connection is flaky.

Integrating such adaptive applications into the Aura framework brings up two questions. First, where should we draw the line between the kinds of adaptations managed internally by the applications, and the kinds managed by the EM and Prism? Second, how can we coordinate the adaptation policies enforced by the applications with the policies at the Task Management and Managed Environment levels? Below we review the roles of Prism and EM concerning adaptation, summarized in Figure 4.3, and the following two subsections address these questions, in turn.

At the Task Management level, changes in user tasks cause Prism either to adjust the service composition of currently active configurations, or to activate or deactivate whole configurations. For that, Prism interacts with the EM to evaluate how well alternative service configurations can be supported in the environment, and once a decision is reached, Prism requests the EM to carry out a specific reconfiguration in the environment. Reconfiguration at this level is triggered by human actions and occurs at a human time-scale (minutes).

At the Managed Environment level, reconfiguration consists of swapping suppliers for services that were requested by Prism. This is triggered whenever the configured set of suppliers in the environment no longer offers the best utility for the requested set of services. Broadly, there are two causes for that: first, a change in the capabilities of the environment, such as current suppliers failing or becoming disconnected, or new suppliers becoming available. For instance, suppose that a user initiated a teleconference using his handheld while walking down the hall: new suppliers become available to the user when he enters an office with a large screen and good teleconferencing capabilities. The second cause for reconfiguration is significant resource variation, which may result in dropping the QoS offered by the currently active suppliers below what is possible to achieve from another set of suppliers.

Based on periodic “heart-beat” messages (see previous sections) the EM periodically evaluates the current environment configuration against possible alternatives. If a better alternative is found for a currently active supplier, the EM may proactively swap the supplier, or it may coordinate with Prism on whether and when to swap it (see Changes in the Environment, above). This kind of evaluation takes place at a time-scale of a few seconds.

Below, we discuss the interplay between the adaptation at the two levels above, and the adaptation at the level of applications.

Integrating Adaptive Applications

Recent research on adaptive applications introduced mechanisms for the dynamic reconfiguration of distributed components in response to resource changes (e.g., [8,31]). For example, an application for translating natural language running on a handheld may run sophisticated algorithms on a remote server when bandwidth is plentiful, and simpler local algorithms when the connection is poor. Typically such applications evaluate alternatives and perform reconfigurations at a time-scale of hundreds of milliseconds, or less.

To support integrating such adaptive applications into the Aura framework we need to answer questions like: should the identification and configuration of remote components be managed by the EM, or internally by the applications? Is there a rigid line of responsibility, or is there room for hybrid solutions?

Complex applications may take advantage of the mechanisms offered by the EM to find and configure distributed components (see Chapter 7, Service Decomposition). However, if off-the-shelf applications include customized mechanisms to configure their own distributed components, that should not be an impediment for their integration into the framework.

The current design of the Aura framework accommodates the integration of applications, regardless of their use of internal mechanisms for adaptation. However, to enable the EM's role with respect to resource allocation, such applications should expose a model of their QoS behavior to the EM (see [85] for details). Based on that model, the EM views, activates, and manages the corresponding Supplier as a unit: all internal behavior is treated as a black box by the EM. Furthermore, adaptive applications should be amenable to have their adaptation policies determined externally (by Prism) and passed dynamically, as appropriate (see below).

Coordinating Policies

A persistent problem for adaptive applications is to determine the adaptation policies that users would like to see enforced. This problem would be easier if QoS were expressed along a single dimension: whenever resources are plentiful, make the QoS "better." For instance, a media player playing a video stream over a network connection can adjust the fidelity of the video depending on the available bandwidth. If the bandwidth improves, it can increase the video fidelity.

Unfortunately, QoS is seldom expressed along a single dimension. In the example of playing a video, above, when bandwidth improves should the media player increase the frame rate, the image quality, or both? The answer depends on the user preferences for the current task. If the user is watching a sports event, he may prefer frame rate to be privileged at the expense of image quality. For watching a documentary on painting, the opposite might be preferable.

To make matters worse, resource adaptation policies should be coordinated among the several applications supporting a task. For example, suppose that the user is watching the video on a PDA, and that he wants to take notes on the video using speech recognition. Suppose also that, when bandwidth is plenty, the (adaptive) speech recognizer may ship the utterances to a remote server and receive the results of the recognition. If the media player aggressively uses the available bandwidth, it may render the speech recognizer inoperative, or helplessly slow. One-size-fits-all fairness policies enforced by the operating system or networking levels may not result in the resource allocation that delivers the best results for the user's task.

Clearly, determining the appropriate QoS tradeoffs and optimal resource allocation among the several applications supporting a task is a hard problem to solve at the level of applications.

The Aura framework addresses this problem as follows. First, the EM calculates the optimal resource allocation among the suppliers, as part of the maximization in Formula 4.2. Second, Prism holds the QoS preferences that drive the preferred QoS tradeoffs for the task (see Definition 3.3 and Figure 3.6). These are passed to the Suppliers upon activation of a service and whenever there are changes: for instance, if the user preferences change in the middle of a task.

Implementation

This section describes the implementation of the Aura framework constructed for this dissertation. For practicality, this implementation makes the following assumptions (see also the discussion of design and engineering decisions in Chapter 7, as well as Chapter 8, Future Work):

- Each task is accessed by a single user (cooperative tasks between multiple users is beyond the scope of this dissertation).
- The user interacts with a single instance of the infrastructure at any given time and location. This assumption will have to be dropped to account for situations such as the user carrying around a laptop with an instance of the infrastructure, and entering a location containing another instance of the infrastructure, say his office. Presumably, the user will expect the two infrastructures to cooperate so that he can access all the capabilities seamlessly.
- A distributed file system is available wherever the user may want to access his tasks. For situations where this option is not practical, the infrastructure can easily be extended for using other file access mechanisms, such as https.
- The Suppliers handle issues of data format compatibility. For instance, a Supplier of text editing services should recognize alternative document formats and perform the appropriate transformations, as necessary.

The current version of the infrastructure includes Java implementations of Prism and the EM, as well as implementations of several Suppliers.

The following two subsections describe the implementation of Prism and the Suppliers, while the implementation of the EM was carried out by Vahe Poladian [70], according to the specifications described herein. The third subsection below presents an evaluation of the performance of the infrastructure.

Prism

Figure 4.4 shows a sketch of the internal composition of Prism. The interactions among the components of Prism are realized as (Java) method calls, and the interactions with the distributed file system use the standard (Java) file system API. The asynchronous, peer-to-peer interactions corresponding to the connectors in Figure 4.2 are implemented as the exchange of XML messages over TCP/IP.

The Speakeasy component handles the identification and authentication of users, as well as obtaining the encryption keys for accessing the task models and personal materials of each user.

Once a user is authenticated with the infrastructure, a Task Manager component is created for him or her. A Task Manager contains two subcomponents: first, a Dashboard lists the tasks that the authenticated user may wish to work on. Second, Lamp supports browsing the past and present tasks, based on the persistent task identity defined in Chapter 3.

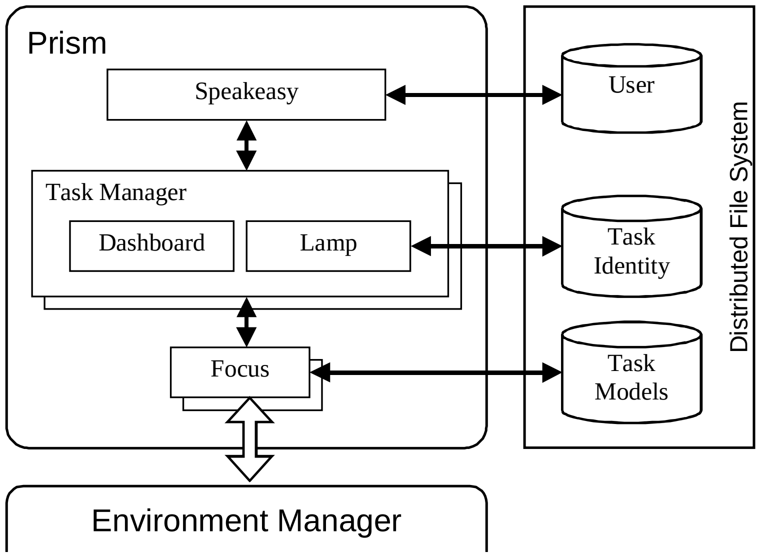


Figure 4.4 Internal composition of Prism.

The Dashboard creates a Focus component for each task listed. An instance of Focus interprets and manipulates the corresponding task model, as defined in Chapter 3. Each Focus starts a session with the EM, following the Prism-EM protocol mentioned in the previous sections.

Chapter 5 elaborates on the functionality of the Dashboard, Lamp, and Focus.

Suppliers

A number of students coordinated by Bradley Schmerl collaborated in implementing suppliers by wrapping BabelFish (web-based translator), Excel (spreadsheet), Festival (speech synthesizer), Internet Explorer, GNU Emacs (text editor), Media Player, MSWord (text editor), PowerPoint (slide editor), Sphinx (speech recognizer), and Xanim (media player). Each of the suppliers was developed using the most convenient language to access the application's APIs, ranging from C/C++, to Java, to Lisp. We have tested the infrastructure on Windows and Linux platforms, including the migration of user tasks between the two.[13]

Chapter 7, Software Engineering of Service Suppliers, discusses what we learned from our experience in implementing the suppliers mentioned above.

[13] Naturally, task migration is constrained by the suppliers available under each platform. At present, only Suppliers for Emacs and Xanim were developed for Linux.

Evaluation

This subsection focuses on evaluating the performance of the infrastructure. See Chapter 6 for the thesis validation and Chapter 7 for a discussion of the design and engineering decisions.

For evaluating the performance, we focused on metrics that determine the user perception of whether the infrastructure is usable on a daily basis. Broadly, there are three groups of such metrics: first, once a user authenticates, how long does it take to have an operational Dashboard; second, how long does it take to search tasks using Lamp; and third, how long does it take to suspend and resume tasks.

With respect to the first and second groups of metrics above, we expect the values to depend on the number of tasks defined by the user. Specifically, we expect that the higher the number of tasks, the longer to find out which tasks should be listed in the Dashboard, the longer it will take to search for a task, and the larger the memory footprint. We measured the memory footprint, since that may critically influence performance in small devices.

To measure the performance variation with respect to the number of tasks, we populated a large number of task definitions (see Chapter 3, Task Identity) using data extracted from random text documents. Since we want to support the definition of hundreds of new tasks per year of usage, we created about 10,000 task definitions. We then repeatedly divided the task directory size in half to obtain the variation of the performance with respect to the number tasks.

The experiments below were carried out on a IBM ThinkPad 30 laptop running Windows XP Professional, with 512 MB of RAM, 1.6 GHz CPU, and WaveLAN 802.11b card. Prism and the EM each run on a Hot Spot JRE from Sun Microsystems, version 1.4.0_03.

For the first group of experiments, we measured (a) the latency *d* of reading the user's directory of tasks, (b) the latency *s* of searching for the pending tasks (the ones that should be listed in the Dashboard), and (c) the latency *f* of finding the feasibility of one task. The overall latency between authentication and the availability of an operational Dashboard is *d+s+n.f*, where n is the number of tasks listed on the Dashboard.

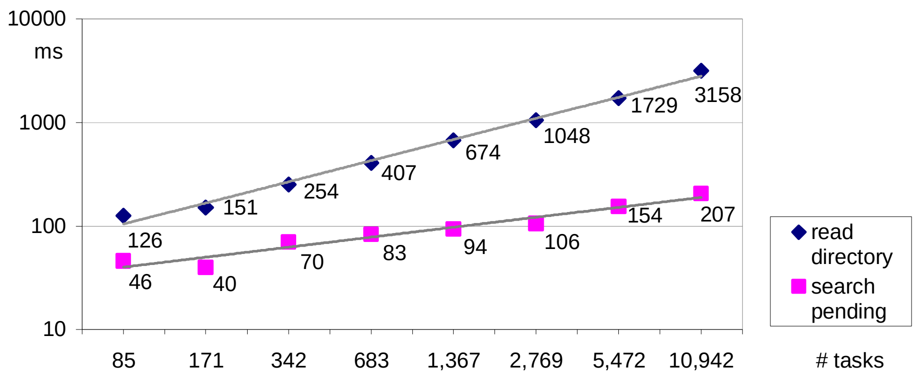


Figure 4.5 Latency of dashboard availability after user authentication.

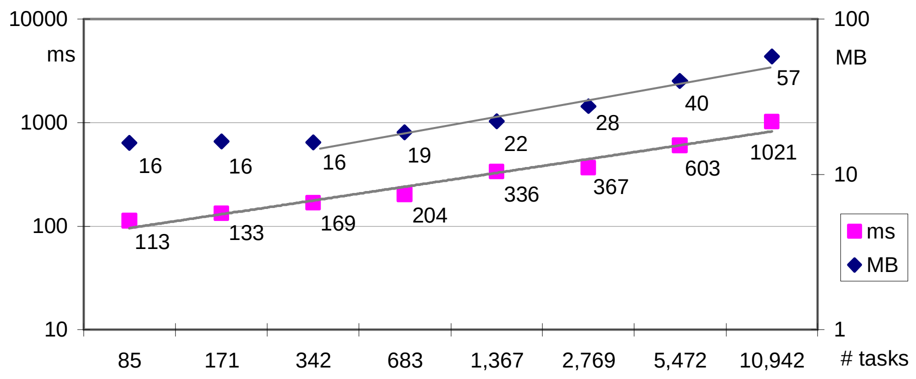


Figure 4.6 Latency of task searching and memory footprint of Prism.

The diamond-shaped points in Figure 4.5 correspond to the latency *d* of reading the user's directory of tasks, currently implemented over the file system. The square-shaped points correspond to the latency *s* of searching the pending tasks (the ones that should be listed in the Dashboard) after the directory was read. As expected, the latency grows linearly with the number of tasks, being under 1 second for well over 2000 task definitions.

The latency *f* of finding a task's feasibility is on average 200 ms (standard deviation 50 ms). Recall that this involves the constrained maximization of the utility function for each of the alternative configurations (Formula 4.2). These numbers were obtained from tasks ranging from 4 to 24 alternative combinations of suppliers for the required services. The performance variation is due to the different numbers of services in the task, and QoS profile of the suppliers.

For the second group of experiments, we measured the latency of searching tasks using Lamp and the memory footprint of the infrastructure. As expected, see Figure 4.6, both grow linearly with the number of tasks, after a significant number of tasks have been created. The current implementation keeps the task directory in memory, after it has been read after authentication. Of course, the penalty in memory footprint is compensated by the swift search times: less than 1 second for a search such as the one illustrated in Chapter 5, even against 10,000 task definitions.

Should the memory footprint become an issue, for instance when deploying the infrastructure on a handheld computer, the task directory can be read for every search. The memory footprint of Prism would drop to 16 MB, and the latency of each search would be increased by the latency of reading the directory (Figure 4.6).

The memory footprint of the EM ranges linearly from 7 MB to 15 MB when it holds the descriptions of 20 up to 400 services in the environment. By comparison, a "hello world" Java application under the used Java Runtime Environment (JRE) has a memory footprint of 4.5 MB, and a Java/Swing application that shows a "hello world" dialog box has a memory footprint of 12 MB.

For the third group of experiments, we measured the latency of suspending and resuming tasks.[14] For resuming a task, the infrastructure takes an average of 700 ms (standard deviation 200 ms) to activate all the required services. For suspending a task, the infrastructure takes an average of 170 ms (standard deviation 20 ms) to obtain a snapshot of the user-level state and deactivate the involved services.

In conclusion, the latencies for obtaining an operational Dashboard after authentication, and for searching tasks, are well within the usual values for obtaining an operational desktop after authentication, and for searching files, respectively. Furthermore, the latency introduced by the infrastructure for suspending and resuming tasks is mostly insignificant when coupled with starting up applications. What a user clearly perceives is that applications quickly recover the user-perceived state where a task was previously interrupted, and that all services associated with that task start up as a unit.

[14] The experiments were conducted with tasks containing between 1 and 5 services, but it should be noted that this latency is overhead introduced by the infrastructure in the service activation/deactivation protocols. The activations themselves may then proceed in parallel in a modern operating system. Consequently, the resume/suspend latency introduced by the infrastructure is mostly constant, rather than growing proportionally to the number of services in a task.

Chapter 5

Describing and Operating on Tasks

This chapter focuses on the part of the infrastructure that enables users to describe and operate on their tasks. Task management promotes user tasks to first-class entities in the system and thus enables users to operate directly on their tasks.

Such operations treat as a unit all the services and materials involved in a task.[15] For instance, a user may suspend a task at home and resume it at the office. To support scalable task management, the infrastructure needs to address the three key properties: treating tasks as coordinated set of services, scalability in space, and in time.

First, the mechanisms available to users for describing their tasks should be simple to use, yet powerful enough to capture the task models defined in Chapter 3. Second, describing tasks should have a low entry cost and provide incremental benefit for incremental effort. And third, the mechanisms for describing tasks should be clear in the assumptions made, and make it easy for users to make adjustments and modifications.

The following sections discuss the user interfaces we incorporated in the infrastructure for users to operate on tasks, and to describe tasks, in turn.

[15] Refer to Figure 3.1 for terminology.

Operating on Tasks

The fundamental tenet of task management is that users should be able to refer to, and manipulate as a unit, the collection of services and materials involved in a task. As such, users can perform operations on a task, such as suspending it, resuming it, or closing it once they believe they are done with the task. Applying such operations to tasks changes their state, and pushes tasks through a life cycle.

Mobile users may like to take full advantage of the computing systems accessible to them, much like they take advantage of the furniture in each space. However, for carrying out their tasks in different locations, users may have to deal with environments with very different capabilities.

Therefore, it is important for users to know, at a glance, of the feasibility of carrying out each of their tasks in the current environment. To prevent overwhelming users with information that is likely to be irrelevant, the view over a user's tasks should be limited to the tasks that the user may want to work on at the present time and location.

Scalability of task management in time means that users may want to recover tasks closed long ago, to reopen them, or to create a new instance of a recurring task. Of course, users should always be able to find their tasks, and operate on them as necessary, whether or not those tasks appear on the task view mentioned above.

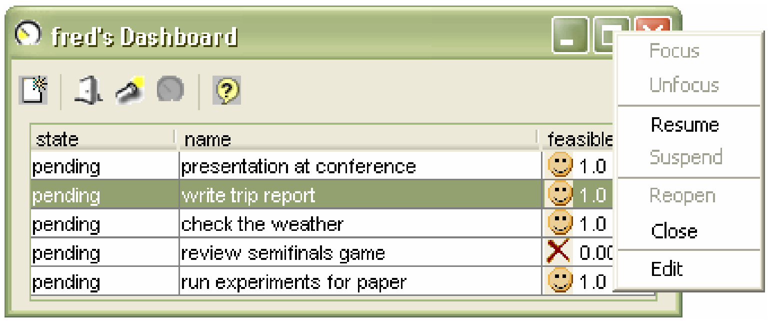
The three subsections below address each of these topics.

Tasks at a Glance

Mobile users that wish to take full advantage of the computing environments at each location may interact with many different devices ranging from mobile phones, to personal computers, to smart rooms. Given the differences in those environments, some tasks may be better supported than others; and which tasks are well supported may change from one environment to the next.

Additionally, users might want to constrain the context in which some tasks should be carried out. For instance, a work-related task involving confidential data may be carried out at the company's premises, but not at the café; or watching a movie may only be carried out after business hours.

To address environment variability, the infrastructure should provide users with a clear indication of the feasibility of each of their tasks in the current environment. The question is: how to show that information to users? And even before that, how to show the tasks that a user may want to work on? Many current desktop managers present tasks as the set of services itself, normally in a well-defined area of the screen. In that case, the user identifies tasks either by visually recognizing the windows corresponding to the set of services, or by remembering their location on the screen. However, the amount of screen real estate required by this technique grows rapidly with the number of tasks – as does the amount of other resources, if the services are kept active. Therefore this technique is limited to rich environments such as smart rooms.



state	name	feasible
pending	presentation at conference	☺ 1.0
pending	write trip report	☺ 1.0
pending	check the weather	☺ 1.0
pending	review semifinals game	✗ 0.0
pending	run experiments for paper	☺ 1.0

Figure 5.1 Fred's list of pending tasks on the dashboard.

Taking advantage of the persistent identity defined for tasks (see Chapter 3 and also Task Identity, below) the infrastructure lists the relevant tasks on a Dashboard. Figure 5.1 shows an example Dashboard for Fred. Tasks are listed by name, and the feasibility of the task in the current environment is listed alongside. Of course, this technique works best when users assign tasks names that are meaningful to them, at least for the duration of the task, i.e. while it is listed in the Dashboard. As discussed in Chapter 3, task names don't have to be unique, or even be present: users can always find (identify) their tasks using information other than the name (see also Finding Tasks, below).

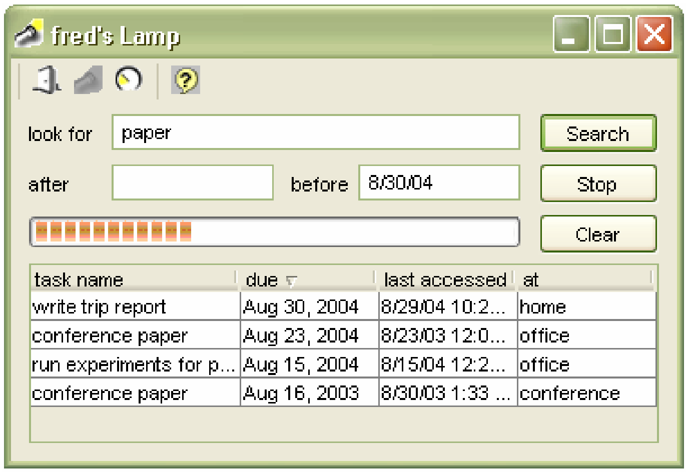
Task feasibility, as defined in Definition 4.4, is a real number between zero and one. To make it easier to recognize feasibility visually, the current version of the infrastructure codifies it in four intervals. Figure 5.1 shows these intervals ranging from a happy face (☺), for values close to one, to a neutral face, to a frown, down to a red cross (✗), for values close to zero. In the figure, the task *review semifinals game* has very low feasibility, presumably because the current environment lacks the services or the resources to play the movie adequately.

To avoid listing information that is likely to be irrelevant, by default the Dashboard lists only tasks that are pending and enabled. A task is *pending* until the user decides that the task is completed and closes it (see Task Life-cycle, below). A task is *enabled* if it might be carried out at the present time and location, according to user-defined context constraints (see Task Identity, below). In Figure 5.1, all the listed tasks are pending and enabled for the current context, but reviewing the game is not feasible with the current resources.

Users can always access all their tasks using a query mechanism (see below), and drop any task of interest on the Dashboard. Once a task is dropped on the Dashboard, its feasibility is evaluated and listed.

Finding Tasks

Scaling task management in time means that users may want to refer to their tasks after they are gone from the list of currently pending tasks – that is, after they are gone from the Dashboard introduced above. Additionally, users may want to refer to tasks that are not listed because they were not marked as enabled for the current context. In the latter case, a user may want to adjust



look for: paper
after:
before: 8/30/04

task name	due	last accessed	at
write trip report	Aug 30, 2004	8/29/04 10:2...	home
conference paper	Aug 23, 2004	8/23/03 12:0...	office
run experiments for p...	Aug 15, 2004	8/15/04 12:2...	office
conference paper	Aug 16, 2003	8/30/03 1:33 ...	conference

Figure 5.2 Search for papers accessed before 8/30/04.

the context constraints, or he may want to access a task despite the constraints. Whatever the case, users may need to find any of their tasks and access them.

Taking advantage of the information associated with tasks, the infrastructure includes a task browser, called Lamp (see the discussion in Chapter 7, Finding Past Tasks). Figure 1.3 shows an example search for Fred. Lamp presents a metaphor similar to web search engines, allowing users to enter searched keywords in the *look for* field.

Searching consists of computing the similarity between the index of searched keywords and the index of terms for each task. Lamp builds an index of terms for each candidate task, which contains the terms present anywhere in the task's model, as defined in Chapter 3, Task Identity. This includes the task name, notes on the purpose or goals, due date, when and where the task was accessed, etc. For computing the similarity, each searched keyword that is present in the index scores one point.

However, unlike keywords, dates are not amenable to exact matching. For instance, Fred may remember that he wrote a paper in the summer of 2004, but may have no idea of the exact dates.

Date criteria are expressed in the fields *before* and *after*, which allow the specification of a timeframe of interest. If no timeframe criterion is specified, all tasks will be searched. If a *before* date is entered, only the tasks that were created, accessed or due before the specified date will be considered. For instance, *due* on *Aug 23, 2004* satisfies *before 8/30/04*. If both a *before* and *after* dates are specified only tasks with at least one date in the specified interval will be searched for keywords.

Lamp's search results are sorted by similarity, and include all the tasks that match at least one of the searched keywords – and that satisfy the timeframe criteria. In the current implementation, if no searched keywords and no timeframe criteria are entered, the search returns all the tasks that the user ever defined.

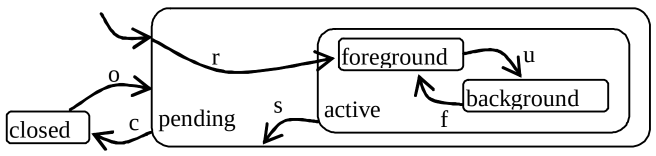


Figure 5.3 State transition diagram for a task.

Task Life-cycle

Task management turns user tasks into first class entities in computer systems. As such, users can perform operations on a task. For instance, when a user decides to *resume* working on a task, the infrastructure configures all services involved in the task, and recovers the user-perceived state of the task as of the last time the task was interrupted. Similarly, once a user decides to *suspend* working on a task, the infrastructure captures a snapshot of the user-perceived state of the task and deactivates all the involved services. Once a user believes he no longer needs to work any further on a task, he may *close* the task. If later on he decides otherwise, he may *reopen* the task.

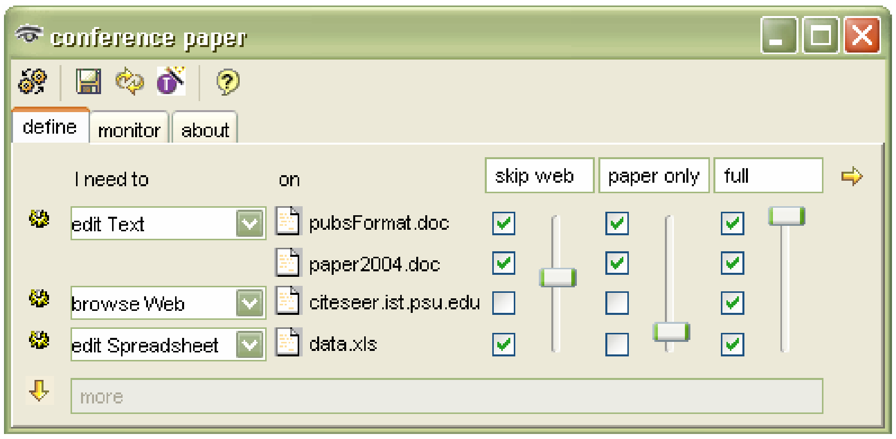
Additionally, we provide two operations as a lightweight mechanism for swapping among active tasks without deactivating the services. Users may switch active tasks between the foreground and the background of their attention, by selecting the *focus* and *unfocus* operations, respectively. For instance, upon an unfocus operation, applications with a GUI may react by minimizing their windows; data streaming servers may react by not streaming data (without closing the connection,) etc.

Figure 5.3 shows a state transition diagram for user tasks. States are represented as boxes and transitions as arcs annotated by the first letter of the triggering operation.

Users may operate on tasks by selecting the options in the popup menu associated with each entry in the Dashboard (see Figure 5.1). The operation entries on the Dashboard's popup menu are enabled or disabled depending on each task's state. Tasks change state as a result of the operations.[16]

When a task is first created it becomes *pending*. Pending tasks become *active* after being resumed, and go back to pending upon suspension. Pending or active tasks can be closed. Closed tasks do not show on the dashboard, by default, but can be browsed using Lamp, dropped back into the Dashboard, and operated on, as necessary. Note that the state *active* is a sub-state of *pending*: an active task is still pending. Note also that *foreground* and *background* are sub-states of *active*: when a task is resumed, it goes to the foreground.

[16] As mentioned earlier in the Tasks at a Glance subsection, users can associate context constraints with a task, constraining its appearance on the Dashboard to certain locations, timeframes, or other context properties. However, *enabled* is not a task state since it doesn't depend on the operations on tasks, but rather a selection of tasks determined by the user's context and the constraints associated to each task.



I need to	on	skip web	paper only	full
edit Text	pubsFormat.doc	✓	✓	✓
	paper2004.doc	✓	✓	✓
browse Web	citeseer.ist.psu.edu	☐	☐	✓
edit Spreadsheet	data.xls	✓	☐	✓

Figure 5.4 Fred's task definition for writing a conference paper.

Describing Tasks

As discussed previously, the mechanisms for describing user tasks should be simple to use, yet powerful enough to capture the required task models. Additionally, the mechanisms should have a low entry cost and deliver incremental benefits for incremental effort. Finally, the mechanisms for describing tasks should also be explicit about the assumptions they make, and they should make it easy for users to correct incorrect assumptions.

These goals guided the design of the mechanisms for describing user tasks. The following subsections present the mechanisms that allow users to describe each part of the task models introduced in Chapter 3.

Coordinated Use of Services

The fundamental tenet of task management is that users should be able to refer to, and manipulate as a unit, the collection of services and materials involved in a task. Therefore, the baseline model of a task needs to include the services and materials involved in that task.

To make the mechanisms for describing tasks accessible to the average computer user, their model of interaction is grounded in the familiar metaphor of drag-and-drop. Users associate materials with a task by dropping them into the task definition window. When that happens, a default service is chosen, based on the type of material, but users may always override the default service selection. Users may also include services in a task independently of materials.

To illustrate this, we present a simple scenario of a user, Fred, who is about to write a paper. Fred considers opening the relevant files and applications on a need-to basis, using standard OS mechanisms. However, since this task will persist for the next few weeks, Fred decides to create a task definition (see Figure 5.4). Initially, Fred includes only editing the paper, and he does that by pressing the down arrow at the bottom-left of the (empty) task definition window and selecting *edit text*. The text editor activated by the infrastructure brings up a (default) blank document and Fred starts working. As Fred browses the web, he decides to associate a relevant web page with the task, so that it is brought up automatically every time the task is resumed. To do that, Fred drags the page shortcut out of the browser and into the *more* field at the bottom of the task window (the default service *browse web* appears automatically). Later, Fred decides to start entering the performance data on a spreadsheet. Again, Fred simply drags the file produced by the data-gathering tool, from the file system explorer into the *more* field and selects *edit spreadsheet* for it.

The services listed in the task definition window support popup menus for deletion and for bringing up the preferences associated to that service (see below). Materials also support popup menus for disassociating them with the task.

Recall from Chapter 3, Configuration Preferences that the infrastructure can do a much better job in configuring diverse environments if it knows what the user prefers in different circumstances.[17] The right-hand side of Figure 5.4 defines alternative operation-mode configurations and their order of precedence. The (default) *full* configuration includes all the activities defined for the task. In addition to that, Fred defined the *skip web* degraded-mode configuration for when the circumstances are such that either a browser or connection are not available, or that the quality of service is so poor (for instance, due to low bandwidth) that Fred would rather focus on the other activities. Fred also defined the *paper only* configuration for last resort circumstances, for instance when having only a handheld with extremely limited resources. Fred can define as many or as few operating modes as he feels appropriate.

Preferences

User preferences play a key role in addressing the heterogeneity of computing environments, and the fact that their resources are subject to frequent variation. The model of user preferences seen in Chapter 3 spans three aspects: first, *configuration preferences* focus on the alternative sets of services to support a task; second, *supplier preferences* focus on the choice of particular service suppliers for each service within a task; and third, *QoS preferences* focus on the acceptable levels of quality of service and preferred tradeoffs. The latter are the most complex.

The expressiveness of these models is key to address the variability of environments. However, that same expressiveness may imply a non-trivial user investment for building and fine-tuning such models.

To lower the entry cost of describing user preferences, the infrastructure provides defaults and templates, as appropriate. When describing a new task, users specify their preferences by building on the provided defaults. A convenient mechanism for cloning task models is also available, which enables users to describe new tasks starting from existing ones as templates (see also the discussion in Chapter 7, Task Recurrence).

[17] Chapter 3 also defines models for the interconnection of services, but for scoping reasons, the current implementation of the infrastructure makes no provision for describing service interconnection.

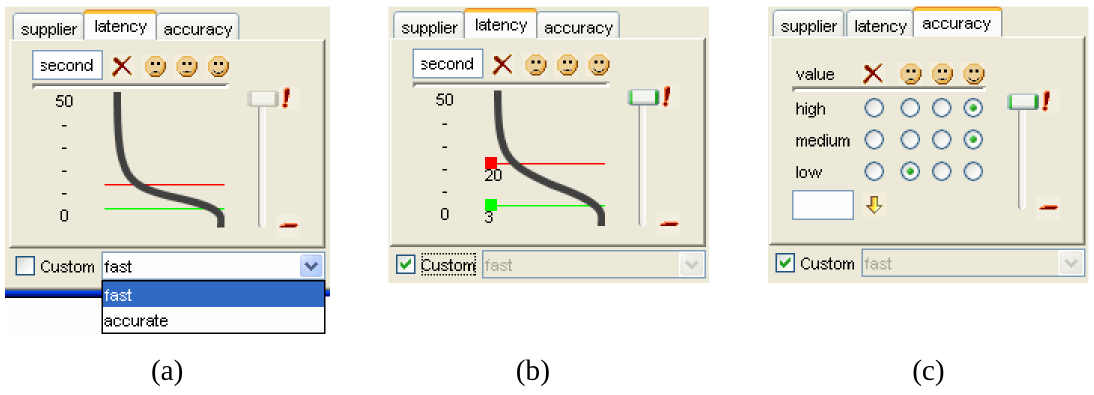


Figure 5.5 QoS preferences for the language *translation* service.

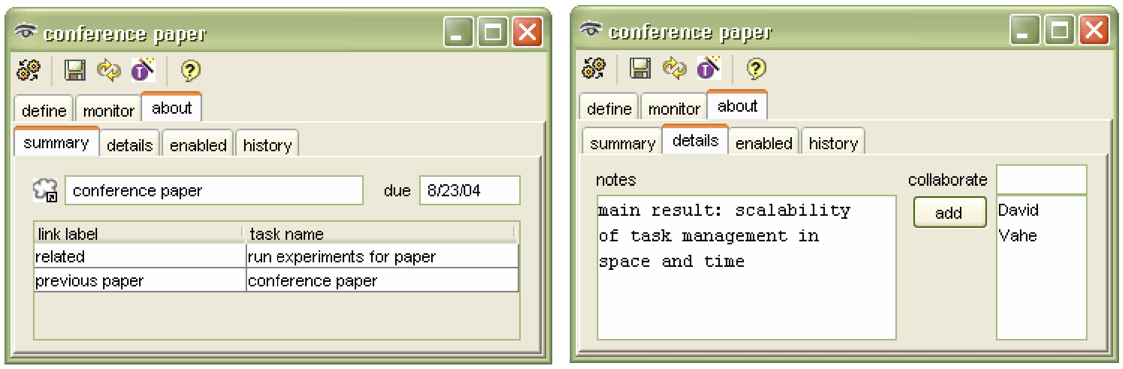
The infrastructure provides a set of templates for the QoS preferences of each service type. Each template encodes a common QoS tradeoff. For instance, take the natural language *translation* service mentioned in Chapter 3. The service has two QoS dimensions: *response time* and *accuracy*. The latency of recognizing each utterance has a numeric domain and is expressed in seconds. The accuracy of translation reflects how much the meaning is preserved: with *high* accuracy, the meaning is mostly preserved; with *medium* accuracy, the meaning is roughly preserved; and with *low* accuracy, the meaning may be distorted.

For the language *translation* service, the infrastructure currently provides two templates: the *default* template, and the *accurate* template. The default template has stricter constraints on response time than the accurate template, but is willing to tolerate a lower accuracy of translation.

Users may switch between the two templates in the selection box shown at the bottom-right in Figure 5.5 (a). The area above the selection box shows the meaning of the selected template. There is a tab corresponding to each QoS dimension for the service: a sigmoid is shown for the numeric dimensions, with the *good* and *bad* thresholds highlighted, and a table is shown for enumerated dimensions. To make it easier to interpret visually, the infrastructure codifies the utility space in the same four intervals used in the Dashboard: from a happy face, for values close to one, down to a red cross, for values close to zero. The slide bar associated with each dimension corresponds to the weights w_d in Definition 3.3, and captures how much the user cares about variations along that dimension.

In addition to seeing the precise meaning of their selection, users may adjust their preferences for each task, as required. After selecting the custom check-box to the left of the template selection box, users may depart from the templates and set their preferences directly. The good and bad thresholds of sigmoid curves may be adjusted by dragging the green (lighter) and red (darker) handles, respectively. The entries in tables may be easily changed by selecting the utility for each value in the domain.

The preference templates associated with each service include not only QoS preferences, but also supplier preferences. The supplier preferences are show as an extra tab alongside the QoS preferences (see Figure 5.5). Of course, supplier preferences are shown as a table.



conference paper due 8/23/04

link label	task name
related	run experiments for paper
previous paper	conference paper

notes

```
main result: scalability
of task management in
space and time
```

collaborate add

David
Vahe

Figure 5.6 Defining the identity of Fred's task for writing a conference paper.

At least one template, the default, is provided for each service type. Every time users add a given service to a task, the default preference template for that service is applied. Users may then switch to another template, if provided, or customize the QoS and supplier preferences for the task.[18] The customization applies for the particular entry of the service in the particular task.

Finally, configuration preferences are described by the slide bars associated with each configuration in Figure 5.4. The bar under each configuration C_t corresponds to a maplet $C_t \mapsto U$ in Definition 3.1, and by default, such maplets map to the value 1.

Task Identity

Entering information that defines a task's identity plays a key role in the ability to find that task later on (see Finding Tasks, above). In the current implementation, users may enter a task name, due date, notes (e.g., on the purpose or goals), and who collaborated on it (see tabs *summary* and *details* in Figure 5.6). None of these items are mandatory or have to be unique, including the task name, and users are free to enter as much or as little information as they feel appropriate.

To lower the cost of entering such information, some is harvested automatically: the created date and the log of places and dates the task was resumed and suspended (tab *history* in Figure 5.6).[19]

In addition to task browsing, another way of finding tasks is by establishing relationships among tasks. Users may then follow a stream of relationships to find related tasks. The generic relationships supported by the models defined in Chapter 3 allow for structured relationships, such as task decomposition. To support such structured relationships, the mechanisms for describing tasks would have to restrict and assign a specific meaning to the possible kinds of relationships.

[18] For scoping reasons, the current implementation of the infrastructure includes no user interfaces for adding new templates or change defaults, although that can be done by updating XML files.

[19] The tab *enabled* in Figure 5.6 would support editing the context constraints for the task – not included in the dissertation for scoping reasons.

The current version of the user interfaces for describing tasks allows relationships to be freely established among tasks, with no restrictions on (or interpretation of) the kind of relationship. Users may therefore build unrestricted graphs, much like inserting hyperlinks on web pages. Double-clicking on a task link brings up the window describing the corresponding task.

Task relationships are created using drag-and-drop. For example, a relationship between a task t_1 and a task t_2 is established by dropping a link to t_2 in the links table for t_1 (see *summary* tab in Figure 5.6). The relationship may be freely labeled by the user (the default label is *related*). Links to tasks can be obtained anywhere the task is listed (Lamp, Dashboard, other links tables, etc.) or in the shortcut icon to the left of the task name in the summary tab.

Chapter 6

Thesis Validation

This dissertation describes a new approach to the scalability of task management in space and in time. The approach is based on high-level models of what users need from the computing environment for each of their tasks. Such models are exploited at run-time by an infrastructure that automatically configures the environment on behalf of users. Specifically, in this dissertation, we argue that:

> *High-level models of user tasks can be used to address the scalability of task management in space, across heterogeneous environments, and in time; while simultaneously (a) enabling users to take full advantage of the capabilities and resources accessible in each environment; and (b) relieving users from routine chores associated with configuring and managing those environments.*

Validating this thesis entails demonstrating the following premises: that the proposed models of user tasks can be used to (i) scale task management in space, and (ii) in time; that an infrastructure that exploits such models (iii) enables users to take full advantage of computing environments, and that using that same infrastructure (iv) poses less overhead to users than configuring the environment themselves.

To validate that our approach supports scaling task management in space and time we built an infrastructure that does it. Validating that users are enabled to take full advantage of the surrounding computing environment is demonstrated by construction. Finally, validating that the infrastructure reduces the overhead for users is demonstrated by comparing the overhead of interacting with the infrastructure against the overhead of interacting with the raw environment.

The four sections below focus on each of the premises in turn.

Scalability in Space

An increasingly important property of user tasks is that they may span multiple locations. For instance, a user may start working on the presentation while in his or her office, continue at the office of a collaborator, and pick the task up later at home. If they so desire, users should be able to resume their tasks with whatever computing systems are available at each location. Also, the increasing pervasiveness of smart spaces is causing a shift in the paradigm of computer use: from single-device, tightly integrated interaction, to multiple-device, loose interaction. Scalability in space implies addressing the heterogeneity and distribution of environments.

This section summarizes the features of the infrastructure that support scalability in space and address the associated challenges, as discussed in Chapter 1.

Chapters 4 and 5 described an infrastructure for task management that automatically configures a computing environment on behalf of users. To know what to configure for, this infrastructure exploits the models of user tasks defined in Chapter 3.

An installation of the infrastructure will be able to configure the environment for a given task *t* provided that (a) a model for *t* is available, (b) the materials involved in *t* are available, and (c) the services required for *t* are available in the environment.

Conversely, users will be able to resume the tasks of their choice at any location where they can (i) find an installation of the infrastructure, and (ii) provide that installation with enough information and authorizations to access (a) and (b) above. For the purposes of this dissertation, we will consider scalability only to the locations where users can satisfy (i), either by finding a device that grants them access to an installation, or by carrying one such device on them. We will also assume that users are able to satisfy (ii), either by securing network access and providing information such as URLs and access passwords, or by carrying a personal storage device with the relevant task models and materials.

From the point of view of the infrastructure, securing the access to task models and materials, conditions (a) and (b) above, is addressed by incorporating results from networking and distributed file access (e.g. [76]) – in addition to the obvious ability for reading a personal storage device presented by a user.

An important research aspect addressed by this dissertation is the matching between task models and the services available in the environment. If task models were expressed in terms of the concrete applications used in a particular environment, the tasks could not be resumed in an environment where the same applications are not available.

By making task models independent of specific applications, and expressing them in terms of high-level services, we can reactivate tasks across heterogeneous environments. For instance, for the task of preparing a report, we capture the fact that the user needs to *edit a text document*, not that MS Word is involved in the task. Note that the ability of handling heterogeneity does not curtail taking advantage of a preferred set of applications, wherever they are available. For that, task models register the supplier preferences for the services involved on each task.

By including an environment-independent representation of the user-perceived state in task models, and by using the programming interfaces (APIs) in existing, native applications, we capture and reconstruct the user-perceived state of tasks across heterogeneous environments. The XML-based representation of the user-perceived state enables suppliers with different degrees of sophistication to extract the aspects of the state that they recognize, while preserving the representation of the aspects they don't know how to handle. The supplier code that wraps native applications acts as a specialized translator for the features of each application.

By exchanging asynchronous configuration and monitoring messages between the managing components of the infrastructure and the service suppliers, we can configure distributed environments and react appropriately to failure in components or communication. The interaction protocols among the infrastructure's components discussed in Chapter 4, Heterogeneity and Distribution, enable the configuration and reconfiguration of distributed environments.

Scalability in Time

Another increasingly important property of user tasks is their duration and recurrence. Users may work on some tasks for days or even months; and tasks may need to be referred back to after users thought they were done. For instance, a user may need to find "the report on the trip to Pittsburgh that I wrote last year." Also, users may periodically carry out distinct instances of the same kind of task, for instance, preparing monthly reports.

By presenting users with a list of tasks they may want to work on, in which a task is included until dismissed as completed, we enable users to quickly access their tasks, regardless of where or how long ago those tasks were last worked on. The Dashboard presented in Chapter 5, Tasks at a Glance, presents a list of the pending tasks that are also enabled for the current environment, according to user-defined context constraints. Users may act on the listed tasks not only for suspending and resuming them, but also for closing or re-opening them, as necessary.

By giving tasks a persistent semantic identity we enable users to find their tasks, regardless of where or how long ago those tasks were defined or completed. The Lamp presented in Chapter 5, Finding Tasks enables users to browse their tasks using a metaphor similar to web browsing. The information that supports browsing is circumstantial information about tasks: a name, due date, the purpose or goals for the task, who collaborated on it, etc. Users may enter as much or as little such information as they wish, using the mechanisms presented in Chapter 5, Task Identity.

By offering a mechanism for replicating task models, we enable users to easily create new instances of recurring tasks. The mechanisms for defining tasks presented in Figure 5.4 include the capability to clone task models, which then may be modified from the original, as needed.

Taking Full Advantage

Users take full advantage of an environment when its configuration best matches the users' needs. Users express what they need from the environment for each of their tasks: which services they need, the preferred suppliers for those services, and the preferred levels of Quality of Service (QoS). Environments can be configured in a variety of ways: there may be several alternative suppliers for a given service, and resources may be allocated differently among the service suppliers to best meet the users' preferred levels of QoS.

Users may contribute to establishing a best match by choosing to work on tasks that are well supported by the current environment. For instance, a user may be willing to take notes on a promotional video; but if the video cannot be played with adequate fidelity, maybe because of insufficient bandwidth, the user may be willing to work on his weekly report instead.

Once an optimal match is established for a given task, its optimality should be maintained even in the face of dynamic changes in user needs and in the environment. User needs may change during a task: new services may be added or removed, and users may adjust the preferred QoS tradeoffs to suit their evolving intentions. In networked environments, remote components constantly change their response times and even availability, and resources may fluctuate widely. For users to take full advantage of the environment, it is not enough to match the users' needs when a task is resumed. The environment's configuration should be reevaluated, and adjusted as appropriate, while the user is working on the task.

By solving Formula 4.2, the infrastructure determines the optimal supplier assignment and resource allocation that best matches the user's needs. The Environment Manager (EM), presented in Chapter 4, keeps track of the suppliers available in the environment, of their resource demand for different computing modalities, and of resource availability. The task models presented in Chapter 3, and captured by Prism, hold the alternative configuration of services that may support a task, and the user preferences relative to suppliers and to QoS tradeoffs. Given one such model, the EM uses Formula 4.2 to determine the optimal supplier assignment and resource allocation, and the corresponding utility as defined by Definition 4.1.

By consulting the Dashboard (Chapter 5, Tasks at a Glance), users can make an informed decision about which tasks they wish to resume working on. Prism interacts with the EM to determine the achievable utility for each alternative configuration of services in each task. Then Prism uses Formula 4.3 to determine the feasibility of each task and presents that to users via the Dashboard.

By periodically reevaluating Formula 4.2, the infrastructure detects opportunities for improvement, and may reconfigure the environment, as appropriate. Following the protocols discussed in Chapter 4, Adaptation at Three Levels, the infrastructure monitors changes both in the user's task and in the environment. Prism detects and reacts to changes in user tasks and preferences; and the EM monitors and reacts to changes in supplier availability. In addition to opportunistic reactions to such changes, the EM monitors trends in the QoS being offered to the user and compares the current configuration against possible alternatives in the environment by reevaluating Formula 4.2. In case a better alternative is detected, the EM may reallocate resources among suppliers, or it may cooperate with Prism to determine whether and when to reassign suppliers.

By passing the preferred QoS tradeoffs to service suppliers, the infrastructure guides resource-adaptive applications in enforcing fine-grain adaptation policies. Following the protocols discussed in Chapter 4, Adaptation at Three Levels, Prism guides the policies of resource-adaptive applications according to the QoS tradeoffs preferred at each moment. Research complementary to this dissertation verified that resource-aware applications can dynamically change their adaptation policies in response to Prism's guidance [10].

Reducing the Overhead

When a user resumes a task interrupted somewhere else, or sometime in the past, the environment needs to be configured for supporting that task: suitable suppliers need to be found and activated, the relevant materials need to be accessed, and the user-perceived state of the task needs to be recovered.

Whatever actions a user needs to take to trigger, enable, or oversee the configuration of the environment constitute a task as well. From a strictly technical perspective in human-computer interaction, users carry out tasks while interacting with a computer system: users interact with devices such as keyboard and mouse, and with abstractions such as windows, menus and buttons (e.g. [19]). From this perspective, the universe of user tasks appears rather uniform.

However, from a user's perspective not all tasks have the same nature. When asking a typical user what he or she is doing, one may get answers like "writing a report" or "preparing a presentation." These are the tasks that we focused on elsewhere in this dissertation, and for the purposes of this section let us call them the users' main tasks, *m-tasks*, or simply tasks. Nonetheless, to a lesser or greater extent, users must also participate in the configuration of the environment for each of their m-tasks. Let us call these the configuration tasks, or *c-tasks*.

Users do not choose to perform c-tasks: c-tasks are imposed on users by the nature of computer systems. The more frequent and the more complex the c-tasks, the more distracting they become. Looking into c-tasks is especially relevant in mobile and ubiquitous computing systems, where users are forced to cope with drastic variations of resources and of availability of services [77]. Nevertheless, as we will demonstrate, reducing the complexity of c-tasks also benefits the users of conventional computer systems, such as personal computers on a wired network.

Task management systems change the c-tasks users need to perform. When interacting with a raw environment for resuming a task, users have to deal with finding and starting suitable hardware and software components, they have to deal with accessing the information resources, and with reconstructing things such as layout of windows, cursor positions and application settings. When interacting with a task management system, users need to find the task they want to work on and indicate they want to resume it. The task management system automatically configures the environment on the users' behalf. For that, however, users must have previously defined the task with the task management system, and that constitutes an extra c-task relative to the situation of the raw environment.

In this section we argue that the infrastructure described in Chapters 4 and 5 reduces the overhead associated with the c-tasks involved in scalable task management. For that, we demonstrate that the infrastructure either reduces, or does not increase the overhead associated with each of the three fundamental properties of scalable task management seen in Chapter 1: first, we analyze the effect on the overhead while suspending and resuming m-tasks as coordinated sets of services. Second, we analyze the effect on the overhead while scaling task management in space, across heterogeneous environments subject to dynamic change. And third, we analyze the effect on the overhead while scaling task management in time.

For those analyses, we compare the overhead of using the infrastructure against users configuring the environment themselves, because that represents the state of the art for scaling task management across heterogeneous environments.

The following three subsections analyze the impact on the c-tasks associated with each of the fundamental properties, in turn. The forth subsection discusses the overall impact.

Coordinated Use of Services

Task management systems provide users with a notion of task that involves the coordinated use of a set of services in an environment, and that can be suspended and resumed as a whole. However, to reap the benefits of easily suspending and resuming their tasks, users must pay the cost of defining their tasks.

In this subsection we first analyze the overhead of defining a new task; then we analyze the benefits of using the infrastructure for suspending and resuming tasks. Finally, we present a cost-benefit analysis over time, as tasks go through successive cycles of being suspended and resumed.

We use a count of *operations* as a baseline proxy for the overhead associated with these c-tasks. Such operations represent interactions between users and the environment or the infrastructure. The model of operations is kept at a coarse level, such as opening an application, or resuming a task, since the analysis needs to hold across heterogeneous environments. In particular, it would not make sense to go down to the level of the particular commands, menu selections, or mouse clicks used in a particular operating system, or application. That said, care is taken so that operations correspond to single, comparable interactions. Furthermore, care is taken so that operations put side by side have comparable latencies. For instance, as demonstrated in Chapter 4, Evaluation, resuming a task using the infrastructure has a comparable latency to opening the same applications individually.

Figure 6.1 shows an example of the overhead associated with using the infrastructure for defining a task. For concreteness, we consider an example task of preparing a presentation, where a user, Fred, edits the slides, refers to a couple of papers on the topic, and browses the web for new developments. The column on the left shows the steps involved in starting to work on the presentation without the presence of the infrastructure. The second column shows the

without	ops	with	ops	overhead
		– new task	1	1
– open app (slide editor)	1	– new service (slides)	1[†]	
		– resume task	1	1
– look for relevant papers	*n*	– look for relevant papers	*n*	
– open app (text viewer) on the papers	*m*	– add the papers to the task	*m*[‡]	
		– save task	1	1
– look for web pages	*k*	– look for web pages	*k*	
		– add pages to the task	*n. pages*	*n. pages*
		– save task	1	1

(†) We'll consider that the infrastructure introduces no overhead: for opening an application, users issue a command, click on some icon, etc.; for adding the slide editing service to the new task, users click one button and make a list selection in Figure 5.4.

(‡) Same as above; for adding the papers to the new task, users drag the papers from some file view in the operating system and drop them into the *more* area in Figure 5.4.

Figure 6.1 Decomposition of starting to work on/defining an example m-task of preparing a presentation, and overhead incurred by (extra operations while) using the infrastructure.

steps involved in starting to work on the presentation *and* defining that as an m-task with the infrastructure. The column on the right shows the overhead introduced by using the infrastructure in terms of operations, as defined above.

Note that some steps are common, such as looking for relevant papers; some are added, such as creating a new task definition with the infrastructure; and some may be changed, such as opening an application directly versus including a service/material in the task. For the latter case, the rationale for the comparison is included in the figure. Note also that saving the task definition after including each service prompts the infrastructure to configure the environment accordingly, therefore finding and activating suppliers for the latest additions to the task.

This example illustrates the rule that, for defining a task with the infrastructure, users perform at most two operations for each service/material they want to include in the task definition (one to add the service/material, and optionally another to save the definition), plus one for creating the task definition itself.[20] Quantitatively, let n denote the number of entries (lines) corresponding to services/materials in a task definition window such as the one illustrated in Figure 5.4, then, the overhead for defining the task is at most $2n+1$ operations.

Figure 6.2 shows an example of the benefit associated with using the infrastructure for suspending Fred's presentation task. As before, the column on the left shows the steps involved in suspending the task without the assistance of the infrastructure; and the second column shows the same with the infrastructure's assistance. The third column shows the benefit of using the infrastructure, in terms of the count of operations saved by using the infrastructure.

[20] The cost of naming and entering information about the task is addressed in Figure 6.2.

without	ops	with	ops	benefit
– name new files	*n*	– enter info about task	$n^{\dagger}$	
– close and save files/links	*n. files/pages*‡	– suspend	1	*n. files/pages* - 1

(†) We'll consider that the infrastructure introduces no benefit (or overhead). For naming newly created files, users need to think of names and storing locations for everyone of them, and they need to feed those to the applications. For entering information about the task, users may enter as much or as little as they want in Figure 5.6. If a user enters just a name for the task, the overhead would be 1 instead of n, thus bumping up the benefit to n-1.

(‡) We'll consider that the user could perform saving and closing with a single operation. This is a conservative estimate, since in many applications users have to perform separate operations for saving and closing files: first saving, then closing; or closing and then confirm saving.

Figure 6.2 Decomposition of suspending the m-task defined in Figure 6.1,
and benefit from (operations saved while) using the infrastructure.

This example illustrates the claim that by using the infrastructure to suspend a task, users save at least as many operations as services/materials in their task, minus one operation for telling the infrastructure to suspend the task. Quantitatively, let *n* be as before, then, the benefit of using the infrastructure for suspending a task is at least *n*–1 operations.

Figure 6.3 shows an example of the benefit associated with using the infrastructure for resuming Fred's presentation task. The layout of the example is the same as in Figure 6.2.

This example illustrates the claim that by using the infrastructure to resume a task, users save at least two operations for each service/material in the task (one to start a supplier, and another to recover the user-perceived state), minus operation for telling the infrastructure to resume the task. Quantitatively, let *n* be as before, then, the benefit of using the infrastructure for resuming a task is at least 2*n*–1 operations.

Figure 6.4 plots a cost-benefit analysis of using the infrastructure over time, based on the results above for defining, suspending, and resuming tasks. The horizontal axis shows task life-cycle, starting with definition, and being successively suspended and resumed. The vertical axis shows the cumulative count of operations saved by using the infrastructure (negative values correspond

without	ops	with	ops	benefit
– find files and links	$1^{\dagger}$	– find task in Dashboard	1	
– open files/links with apps	*n. files/pages*	– resume	1	*n. files/pages* – 1
– recover state of apps	*n. files/pages*‡			*n. files/pages*

(†) We'll consider that users have fresh on their minds where the files are stored, and that all the relevant files and links are stored in the same location. This is a conservative estimate.

(‡) We'll consider that users can recover the user-perceived state of each file, and the associated app, with a single operation. This would correspond, for instance, to recover the editing position on a text file. In many cases users may need to recover more of the user-perceived state, making this a conservative estimate.

Figure 6.3 Decomposition of resuming the m-task defined in Figure 6.1,
and benefit from (operations saved while) using the infrastructure.

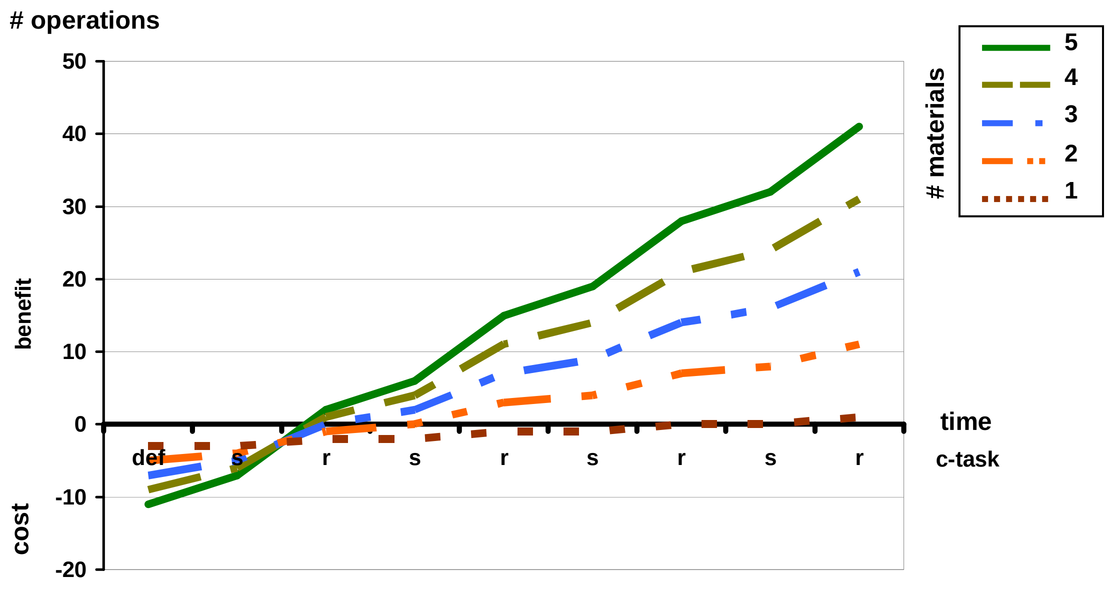


Figure 6.4 Cost-benefit of using the infrastructure over the number of suspend-resume cycles.

to a cost of using the infrastructure). Analytically, these curves are given by:

Definition 6.1 $-(2n+1)+(n-1)s+(2n-1)r \quad \bullet \quad n \geq 1, \quad r,s \geq 0, \quad s-1 \leq r \leq s$

where *n* is the number of entries (lines) corresponding to services/materials in a task definition window such as the one illustrated in Figure 5.4; and *s* and *r* denote the number of suspend and resume operations, respectively. The graph shows curves for the values of *n* from 1 through 5.

The conclusion from this analysis is that the benefits of using the infrastructure are larger, and realized faster, the more services/materials are involved in a task. Although defining a task has larger overhead the more the services/materials involved, that overhead is recovered quicker, especially during resume operations. For instance, a task with 5 services/materials starts off with a definition overhead of 11 operations, but it breaks even on the first resume, and by the fourth resume it saved the user more than 40 operations. In contrast, according to this baseline model, a task with only one service/material only breaks even on the third resume, and saves the user few operations over time. However, defining such a task with the infrastructure would be more rewarding whenever the cost of recovering the user-perceived state is significant (see note ‡ in Figure 6.3).

From Definition 6.1, it is clear that no task will break even on the first suspend, but any task with *n*≥3 will break even on the first resume. Note also that the cost-benefit model expressed in Definition 6.1 is additive. For instance, if a user originally defines a task with *n*=4 and then adds two new services/materials the third time around he works on it, the resulting curve is given by the curve for *n*=4 up until the second resume, and adding the curve for *n*=2 from that point on. This corresponds to the intuition that when users make incremental additions to a task definition, they pay the overhead of making those changes and then the savings increase since the infrastructure can automatically configure more services on the users' behalf.

Scalability in Space

The infrastructure presented in Chapter 4 supports the scalability of task management in space, across heterogeneous environments subject to dynamic change. Users may wish to resume their tasks in different locations, with whatever computing systems are available. When accessing services in networked environments, the availability of the suppliers for those services may change dynamically. Furthermore, the resources necessary to provide those services may fluctuate widely, leading to fluctuations of performance which may affect the user's task.

The previous subsection modeled the overhead of suspending and resuming tasks as a coordinated set of services independently of environment heterogeneity. That is, provided users have the necessary knowledge to configure each environment, the count of operations in the c-tasks will not change due to heterogeneity.

However, environment heterogeneity may cause resuming a task without the infrastructure to become harder for users. The operations on the first column in Figure 6.3 may become harder, since users reaching a new environment need to decide on a possibly different set of applications to activate (different relative to the previous environment). Moreover, users need to recover the user-perceived state using the specific features of those new applications.[21] How much harder these operations become depends on the users' familiarity with each environment. Assuming that the infrastructure offers similar features and interactions across environments, there would be no significant changes on the second column in Figure 6.3.

If a user doesn't have the necessary knowledge to configure the new environment, the infrastructure may prevent that from being a showstopper. When faced with a new environment users may not know which applications to invoke for each service, or how to invoke them (commands, parameters, etc.). Nevertheless, given the convergence of user interface metaphors, users may be able to use those applications, once started automatically by the infrastructure.

The infrastructure also contributes to reduce the overhead due to faults in service supply, and to help users take full advantage of the environment. In distributed environments, the availability of service suppliers may change dynamically: the suppliers supporting a task may fail or become inaccessible, and new, preferred suppliers may join the environment dynamically. By continuously monitoring the environment and performing appropriate reconfigurations, the infrastructure saves users the overhead of doing that themselves.

Finally, the infrastructure contributes to reduce the overhead due to resource fluctuations. Provided resource-adaptive applications are available in the environment, the infrastructure automatically guides those applications in enforcing the fine-grain adaptation policies adequate to each task. Without such guidance, it is hard for applications to determine which policies to apply, and in most cases end up applying one-size-fits-all policies. It is then up to users to

[21] Note that analyzing the impact of heterogeneity is limited to the c-tasks. Although users may find it easier to carry out their m-tasks in some environments, that is not relevant for analyzing the effects of introducing the infrastructure to handle the automatic configuration of environments.

without	ops	with	ops	benefit
– think of file names, directories, or file content		– think of circumstantial facts about the task		†
– interact with search tools	n	– interact with Lamp	*n*‡	
– scan search results	m	– scan search results	*m*	

(†) We'll consider that the notion of task brings no measurable benefit, although one might argue that it is easier to remember domain-related facts about a task, than system-related facts about files and directories. Recent tools index the content of files and allow users to browse the file system in a similar fashion to browsing the web (e.g., [39]) thus bridging some of the gap between domain reminiscences and file system.

(‡) We'll consider that finding one material will be enough to situate all relevant materials. This is a conservative estimate, since if the relevant materials are scattered, the user may need as many searches as materials in the task, as opposed to a single search for the task.

Figure 6.5 Decomposition of finding an m-task completed long ago,
and benefit from using the infrastructure.

bridge the gap, and configure the policies themselves every time a task is resumed, or to endure the effects of policies that are not adequate to their intent.

Research complementary to this dissertation carried out user studies on the usability of the infrastructure for enforcing QoS tradeoffs [10]. In that study, 10 out of 10 participants were receptive to the idea of specifying different QoS tradeoffs in different circumstances, and were able to specify the tradeoffs appropriate to different scenarios using the features presented in Chapter 5, Preferences. Moreover, the authors observed with 95% statistical significance that the participants were able to recognize a correlation between the adaptive behavior of the suppliers and the specified tradeoffs.

Scalability in Time

The infrastructure described in Chapters 4 and 5 supports the scalability of task management in time, enabling users to resume long lasting tasks and to find tasks that were completed long ago. Users may work on some tasks for days or even months; and some tasks may need to be referred back to after users thought they were done.

The previous subsection on the Coordinated Use of Services modeled the overhead of suspending and resuming tasks independently of how long those tasks last. As long as a task is marked as pending, it is listed in the Dashboard, and the benefits of using the infrastructure to resume it are as discussed before.[22]

[22] For this specific case – resuming a pending task in the same environment – one may also compare the overhead of using the infrastructure against using a traditional desktop manager: these would be equivalent, as long as the desktop manager keeps a persistent record of the pending tasks, that is, is able to reconstruct the desktop after a shutdown.

Figure 6.5 shows the decomposition of finding an m-task completed long ago. The column on the left shows the steps for finding the task without the assistance of the infrastructure, that is, of finding the materials involved in the task. The second column shows the steps for finding the task using the infrastructure. Note that the costs of entering the information about tasks are already accounted for during suspend (Figure 6.2).

The infrastructure offers mechanisms for finding tasks that are similar to existing mechanisms for browsing the web, and for finding files in the file system. The most significant difference is that searching for a task (using the infrastructure) corresponds to one search, while searching for the materials involved in a task (without the infrastructure) may require several searches. However, we conservatively claim no benefits since the materials involved in a task may be stored adjacently.

Overall Reduction

By reducing the overhead associated with suspending and resuming a coordinated set of services, task management brings benefits to users of traditional environments. We quantified the costs and benefits associated with providing a notion of task, in terms of high-level interactions between users and the environment. That cost-benefit model is independent of issues related to scalability in space and in time, including issues of heterogeneity and dynamic change. We analyzed the cost-benefit over time, and identified that handling a task as a unit brings more benefit the more services/materials are involved in the task.

Scaling task management in space and time brings additional benefits to users. We discussed how the infrastructure contributes to reducing the overhead when working across heterogeneous environments, and when dealing with situations that may occur in distributed or mobile environments: recovery from faults, proactive identification of better solutions, and guiding fine-grain resource-adaptation policies. We also discussed how scaling task management in time, although necessary for practicality, does not significantly change the cost-benefit balance.

Although this chapter makes a convincing case, ultimately the benefits of scaling task management in space and time need to be evaluated with real users working on real tasks. The success of systems for automatic configuration and self-tuning depends on many real-life factors that cannot be captured adequately by models, benchmarks, or even scripted user studies. At least equally important for that success is the practicality and ease of use for the average user.

Nevertheless, the results of user studies such as [10], and of deploying the infrastructure with a selected group of users, clearly indicate that users react well to the concepts and functionality delivered by the infrastructure.

Chapter 7

Discussion

This chapter discusses important design and engineering decisions that we tackled during our research, as well as aspects where our research interfaces with complementary research. We also discuss some aspects that were partially addressed because of scoping considerations, but limit the selection to aspects that extend the research herein: broader topics that deserve their own separate research are discussed in Chapter 8, Future Work.

The following three sections discuss decisions related to modeling user tasks, to the infrastructure for the automatic configuration of environments, and to the mechanisms for describing and operating on tasks, respectively.

Modeling User Tasks

Since the topic of this dissertation is task management, most of the points for discussion focus on the modeling of user tasks. The two subsections below discuss decisions concerning the level of sophistication at which to model user tasks, and how to support task recurrence. The three following subsections discuss issues related with matching demand for services to the supply in the environment: service naming, substitutability, decomposition, and interconnection. The two subsections at the end focus on guiding the choice of alternative suppliers and their configuration: modeling supplier preferences and Quality of Service (QoS) tradeoffs.

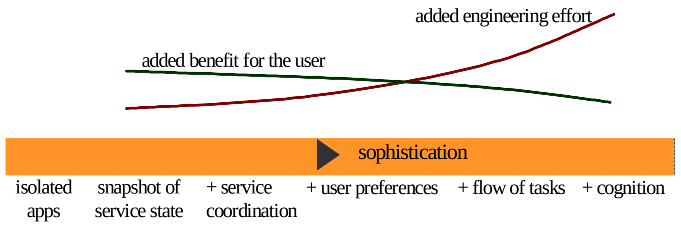


Figure 7.1 Expected added engineering effort and added user benefit as a function of the level of sophistication of task models

Sophistication of Task Models

Possibly the most important decision for the research presented in this dissertation was selecting the level of sophistication for representing user tasks. We based our decision on the working hypothesis informally depicted in Figure 7.1. The horizontal bar shows a range of options for the sophistication of task models: from having no such models (users manually assemble the applications necessary for their tasks and recover the user-perceived state of those applications), to capturing a snapshot of isolated services, all the way up to modeling the cognitive aspects of carrying out the tasks.

Our working hypothesis was that large benefits to the user result from relatively simple representations of tasks, and the more sophisticated those representations, the less significant the added benefit. In other words, we expect the overall benefit for the user to increase monotonically with the sophistication of task models, but that increase will be more significant in the low end of the spectrum.

On the other hand, experience tells us that the effort involved in designing sophisticated task models, and building the mechanisms that exploit those models, increases significantly with the sophistication of the models. This is not to say that research on the upper end of the spectrum is not desirable, however, for practicality, we aimed not to go past the sweet spot of effort vs. benefit.

Specifically, for this research we targeted a level of sophistication that was expected to yield significant benefits for users, with an engineering effort commensurate with the scope of a dissertation. The decision we took, as documented in Chapter 3, was to model the coordinated use of a set of services, including the snapshot of the user-perceived state of those services, and user preferences with respect to the alternative ways the task may be rendered in the environment. Chapter 6 demonstrated that such models, and the infrastructure that exploits them, deliver significant benefits to users.

From the validation work in Chapter 6, it seems plausible that the more significant benefits for the user are achieved by the first levels of sophistication. Although user preferences play a key role in reducing user overhead in the face of strong dynamism in the environment, for frequent situations where the environment is relatively stable, the most benefit is attained by the ability to suspend and resume tasks (recovering their state).

How much benefit the upper levels of sophistication will yield is a topic of future research (see Chapter 8, Future Work).

Task Recurrence

In daily activities, it is common for some tasks to recur. For instance, preparing monthly reports, or preparing weekly class sessions. Although occurrences of recurring tasks share some characteristics, each has its own identity and presumably its own materials (e.g. the monthly report that gets produced).

To save users from defining similar tasks over and over again, it is desirable to support the notion of recurring task. Specifically, users should be allowed to indicate they need to start working on a new occurrence, say, of preparing a monthly report.

The question that arises is how to represent the common characteristics of recurring tasks, simultaneously ensuring the identity and variability specific to each occurrence. One possibility is the explicit definition of task types; another possibility is using an existing occurrence as a template.

Generally speaking, types have the advantage of supporting verification and traceability. For instance, given a task *t*, typing information would allow recognizing whether *t* is an instance of a given type, say, preparing a weekly class session. Should the user change the definition of *t*, typing information supports verifying if *t* still conforms to the type. Furthermore, should the user change definition of the type itself, we can trace all the occurrences of the type and signal the required updates or conflicting definitions.

In the task management domain, however, we believe that the user overhead of defining and maintaining a type system for tasks outweighs the benefits. When a user defines a recurrent task, it is not clear cut which characteristics are common, and should be enforced by typing, and which should be left open to variability. Furthermore, when creating a new instance of a recurrent task, it would be frustrating to users to have to update the type definition every time the requirements for the new task conflict with the original type specification.

The decision we took was to support recurring tasks with the informal notion of templates. When a user decides to start working on a new occurrence of a recurring task, he may copy the definition of an existing occurrence (using the button labeled T in Figure 5.6) and then modify it as desired. Currently, no link is automatically created between the new occurrence and the task used as template, although users may create such links explicitly, using the mechanisms described in Chapter 3, Task Identity.

Service Naming and Substitutability

The naming of services plays a key role in matching service demand with service supply. For instance, if a user task requires a *play video* service, the infrastructure will look for a supplier that advertises a *play video* service. However, one cannot always expect a perfect match of vocabularies in every environment. For instance, suppose that in some environment no supplier announces *play video*, but some suppliers announce *play media*.[23]

The question that arises is where the knowledge about service substitutability should reside: the demand side (models of user tasks), or the supply side (somewhere in the environment)? To answer this question, we need to distinguish two kinds of substitutability: goal substitutability and functional substitutability.

Goal substitutability is determined by what users want to achieve with a task. For instance, for taking notes on a video clip, a user may be willing to either type or to dictate the notes, requiring an *edit text* service or a *speech recognition* service, respectively. These two services then become substitutable in the context of the clip reviewing task; however, the same user may find dictating a research paper cumbersome. Clearly, this kind of substitutability should be represented in the models of user tasks. In our research, we represented goal substitutability as alternative service configurations in task models.

Most work in substitutability coming from research in service ontology focuses on functional substitutability. Underlying that work, there are two possible perspectives for naming: either naming an abstract service, e.g. *printing*, or *sending fax*; or naming an abstract supplier, e.g. *printer*, or *fax machine*. Each perspective offers a tradeoff with respect to naming and substitutability knowledge.

The perspective of naming services is more intensive on naming standards and advertising, but less intensive on substitutability. For instance, suppose that fax machines can also print documents (format issues aside). Both printers and fax machines would agree on a standard for naming services, e.g. *printing*, and advertise them: a fax machine would advertise *printing* and *sending fax*. In this simple example, substitutability is obtained for free in service advertising.

The perspective of naming suppliers is less intensive on naming standards and advertising, but more intensive on substitutability. In the example above, printers would advertise themselves as *printer* and fax machines as *fax machine*. The knowledge that *fax machine* can substitute for *printer* would have to be explicitly represented.

Under either perspective, functional substitutability is clearly environment-specific knowledge and should reside somewhere in the environment: either on the suppliers that announce all services that they are capable of providing, or on a broker for supplier substitutability.

Support for functional substitutability is a topic of active research in the area of service ontology, and work on task management can dovetail on that research.

[23] A general notion of service substitutability involves describing services more extensively than just naming. For instance, it may involve describing the types of materials/parameters associated with providing the service, the supported value ranges, etc. However, the points we make using service naming are valid for more complete service descriptions.

Service Decomposition

Service decomposition is an important aspect of matching service demand with service supply that extends the idea of functional substitutability with one-to-many mappings. For instance, if a user task requires a *speech-to-speech translation* service, and none can be found in the environment, maybe the service can be assembled from parts; e.g. *speech-recognition*, *language translation*, and *speech synthesis*.

The questions that arise are: is there an ideal level of decomposition for naming services? Where should the knowledge about service decomposition reside? And who needs to know about how a composite service is actually being delivered?

Concerning the level of decomposition for naming services, it is not likely that a single definitive answer will ever crystallize. On the demand side, users with different degrees of expertise will ask for things at different levels: an inexperienced user will try to get more abstract services, hiding internal details as much as possible, while an expert user may want to have more control over which, and how the parts are configured. The *same* user may want to have more control over the structure supporting a critical task, but be willing to take an off the shelf solution for a low priority task. On the supply side, it is to be expected that sophisticated environments, such as smart rooms, will have higher-level, well-tuned components, while poorer environments, such as handhelds, will have a collection of generic parts that can be assembled to deliver a similar function but in a less polished way.

Concerning where should the knowledge about service decomposition reside, as argued before with respect to goal vs. functional substitutability, that knowledge resides on either the demand side, whenever users want to be specific about what is being asked; or in the supply side, whenever users wish to rely on technical knowledge resident in the environment. Specifically, Prism knows as much or as little about *what* a user wants as the user tells it: if the user asks for a *speech-to-speech* service, that's what Prism will try to obtain from the environment; if the user asks for three services interconnected in a certain way, that also is what Prism will try to obtain for the user. *How* a requested service can be assembled in a particular environment depends on the capabilities of (i.e. the existing parts in) that environment.

Concerning who needs to know about how a composite service is actually being delivered, that knowledge may have to be exposed all the way up to the user, even if the knowledge to assemble the composite service resides in the environment. For instance, suppose that a requested service is actually being provided by an assembly of parts. The user may have to interact with several applications, meaning several UIs, rather than an integrated one. Prism has to be aware of *what* exactly is being provided in lieu of what was requested, even if for nothing more than explaining it to the user.

Our architectural framework supports these alternatives. The task models described in Chapter 3 support requesting either a (composite) service, or an assembly of services (parts), or requesting both as alternative service configurations for supporting the task.

When building the infrastructure, we experimented with building a supplier that announces a composite service (*speech-to-speech translation*) and that requests the necessary parts from the

EM. In this case, suppliers act as specialized (decentralized) repositories for the knowledge about service decomposition. Whether having a centralized component with such knowledge would be a better solution is a matter for further investigation.

Service Interconnection

In addition to enumerating the services required for each task, an important role of task models is to indicate whether and how those services are interconnected. For instance, to carry out speech-to-speech translation, three services may need to be interconnected: *speech recognition*, *language translation*, and *speech synthesis*.

The question that arises is where the knowledge about service interconnection should reside: in task models, or in the environment? Before answering this question let's examine an example in the physical world. Suppose that a user, Fred, wants to hook up a desktop PC. Fred uses the power cable to connect the power outlet to the PC's power input, the video cable to connect the video output to the monitor's video in, and so on. Fred doesn't need to be familiar with the pin layout of the video cable, or with the specifications of the electrical signals that go in each pin.

There are two levels of knowledge with respect to interconnecting services in the environment. The knowledge that can be asked from non-expert users is high-level knowledge about the types of inputs and outputs, and which get attached to which. For instance, Fred may decide to connect the services as described above, or he may decide not to use speech recognition in some circumstances, and hook the input of the translator to some text input device, instead.

The level of knowledge that is required to verify that interconnection is possible, and to dynamically interconnect services, is the level captured by software architecture description languages (ADLs). ADLs such as Acme, or their XML-based counterparts, such as xArch, may represent the signatures of the interaction ports and details about the interaction protocols [27,34]. This level of knowledge is held by software architects and system's specialists.

The decision we took is to represent user-level knowledge in task models, and to represent software architecture-level knowledge in the environment. Clearly, service suppliers need to be aware of the specifications of their own ports (if nothing else to advertise them to a component in charge of interconnection).

However, whether to centralize knowledge about verifying and establishing interconnections in a component such as the Environment Manager (e.g. [20]), or to decentralize some of it to the service suppliers themselves is a topic for further investigation.

Supplier Preferences

User preferences play a key role when selecting service suppliers among the alternatives in the environment. For services with little or no direct user interaction, such as media playing or language translation, the choice among suppliers is mostly driven by the forecasted quality of service. For services with a heavy component of user interaction, such as text or slide editing, the choice is mostly driven by the available features and by the users' familiarity with the way those features are delivered.

One question that arises is how to model user preferences with respect to the suppliers' features and to the users' familiarity with the way they are delivered. (The subsection on QoS Tradeoffs, below, addresses the quality-related question.)

At one end of the spectrum, user preferences may be modeled by identifying the preferred suppliers. The decision we took, embodied in Definition 3.2, expresses preferences as a discrete mapping between a known set of suppliers and a normalized utility space $U \cong [0,1]$. With this modeling approach, a user, Fred, might note his preference of text editor Emacs over Vim by assigning Emacs a utility close to 1, and a lower value to Vim. To account for unknown suppliers, a wildcard value, *other*, is added to the set of known suppliers. So, for instance, if Fred sets the utility value for Vim higher than the value for *other*, that means Fred would prefer to use Vim than to try an unknown text editor.

At the other end of the spectrum, preferences may be expressed as an abstract model of service features. With this modeling approach, Fred might express that he would prefer a text editor that supports pointing devices and accelerator keys. This approach however, relies on a vocabulary to describe features that is shared between the user (and possibly the task modeling tools) and all the candidate service suppliers across all the environments the user want to work on. Another problem to solve is how to build models of user familiarity with the way features are delivered. For instance, Fred may be familiar with the accelerator keys in Emacs, but may have trouble if the same features are delivered by an unknown editor using a different set of accelerator keys.

A sophisticated solution might permit supplier preferences to be expressed either way, or better yet, combined in a rich format. For instance, Fred might express that he would be happy with either MSWord or with a WYSIWYG[24] editor that supports pointing devices and spell checking. Finding a good balance between the expressiveness and usability of such modeling forms is topic for further investigation.

QoS Tradeoffs

Another question related to the mechanisms for selecting among alternative suppliers is how to model user preferences with respect to the quality of service (see also the subsection on Supplier Preferences, above). As we discussed in previous chapters, quality of service (QoS) is often multi-dimensional. For instance, for watching a video, users may care about frame update rate and about image quality; for automatic translation, users may care about both the accuracy and the latency of translation.

At one end of the spectrum, user preferences may be modeled by indicating which QoS dimension a user cares the most. In the example, a user, Fred, might indicate that response time is preferred over accuracy of translation; and the infrastructure would then configure the environment so that response time is optimized. Although tradeoffs are explicitly captured – in the example above Fred is trading off accuracy for response time – they are so at a very coarse grain. Finer grain questions cannot be answered in this solution: for instance, how short of a

[24] Common acronym for *What You See Is What You Get*.

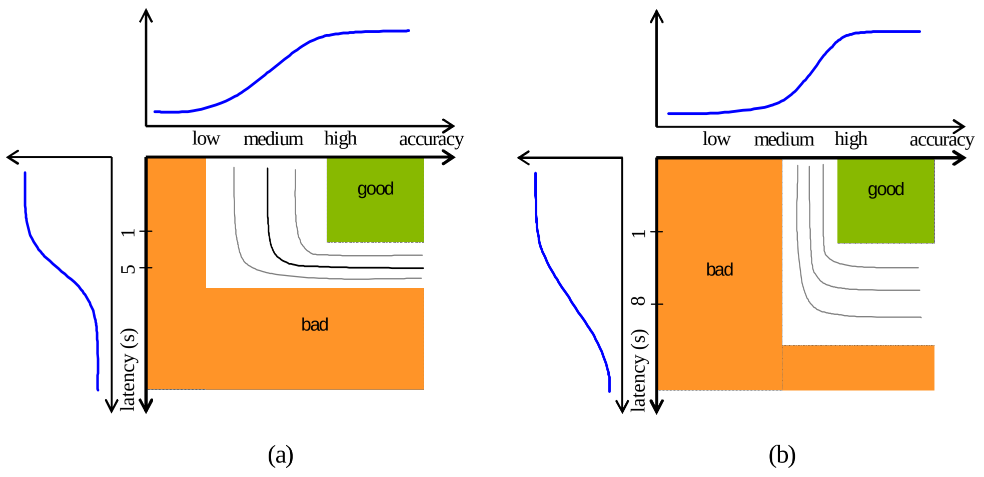


Figure 7.2 Two different tradeoffs obtained by multiplying independent utility functions

response time will satiate the user? And even if accuracy is less important, what if it degrades so much that the translations become unusable?

At the other end of the spectrum, preferences may be expressed as an arbitrary function between the multivariate quality space and the normalized utility space $U \triangleq [0,1]$. For instance, Fred might indicate that he would be happy with medium translation accuracy, as long as latency remains under 1 second, and that he will be happy to wait 5 seconds for highly accurate translations. Although fully expressive, designing mechanisms to elicit this form of preferences from non-expert users is a hard problem, and even more so if more than two QoS dimensions are involved. Additionally, the algorithms used by the infrastructure for finding the optimal configuration need to accommodate an arbitrary multi-dimensional utility function.

The decision we took, embodied in Definition 3.3, lies between these two extremes: user preferences are expressed independently for each dimension. For instance, Fred might indicate that he would be happy with response times under 1 second, but accept response times of 5 seconds; and also that he would be happy with highly accurate translations, but would accept medium accuracy. Figure 7.2 (a) shows the net effect of combining these two (independent) utility functions by multiplication. Combining independent utilities by multiplication corresponds to an *and* semantics: overall QoS is good, only if it is good along each and every dimension. In the figure, the shaded areas labeled *good* and *bad* correspond to the portions of the quality space where the utility is consistently high, or low, respectively. The curves between the two areas are *isoutility*, or indifference curves, along which the utility remains constant.

Although mathematically less expressive than an arbitrary multivariate function, in practical terms the expressiveness of this solution seems adequate. The main reason for this is that automatic configuration is guided my maximizing the utility function – and in a maximization process, variations are the important aspect, not the absolute values that are reached. For instance, note that the highlighted isoutility curve in Figure 7.2 (a) expresses similar preferences to the ones in the example arbitrary function mentioned in the previous paragraph.

One drawback of expressing preferences independently for each dimension is that tradeoffs are not represented explicitly. Looking at the combined plot in Figure 7.2 (a) it becomes apparent that the utility function is allowing wider variation along the accuracy axis than along latency. If the available resources decrease, a configuration algorithm based on utility optimization will tradeoff accuracy for latency; or in other words, latency will be preserved at the expense of accuracy.

However, the very notion of tradeoff is not absolute. Upon a fluctuation of resources, both dimensions are likely to change, and since they are not expressed in the same scale, which dimension is more affected is subject to the observer's interpretation.

The meaning of a tradeoff only congeals when compared against another tradeoff. Figure 7.2 illustrates how two clearly distinct tradeoffs can be obtained by changing user preferences along each dimension independently. In Figure 7.2 (b), Fred got stricter on accuracy and no longer thinks medium accuracy translations are acceptable. On the other hand, Fred is willing to tolerate response times up to 8 seconds. A configuration algorithm based on utility optimization now has more wiggling room – than in Figure 7.2 (a) – along the latency axis.

Although tempting to represent user preferences as concrete tradeoffs, such concrete tradeoffs can only be determined given concrete values of resource availability. Specifically, it would be tempting to find a representation that could answer questions like: when the user wants highly accurate translations, how much latency should he expect? However, the answer to that is not a constant, but a function of the available resources.

For user preferences to play their role in guiding the automatic configuration of diverse environments, those preferences need to remain environment-independent. Nevertheless, such preferences can be superimposed on the concrete conditions of each environment to determine the levels of QoS attainable at each moment.

Infrastructure for Task Management

This section focuses on important aspects where our work concerning building the infrastructure interfaces with complementary research. The four subsections below discuss issues related to the impact of user mobility on privacy, to guidelines for implementing service suppliers (both with respect to APIs and to data representation,) and to the interconnection of suppliers.

User Mobility vs. Privacy

Privacy is an important concern for mobile users. If users are to resume their tasks at different locations using local devices and software, how can their privacy be assured?

In many circumstances this may not be an issue. For instance, mobile workers within premises administered by their company may trust the computing environment in a colleague's office as much as they trust the environment in their own office. On the other hand, mobile users may have concerns about the kind of assurances offered by the environment at their favorite café.

The privacy concerns that arise from scaling task management in space can be addressed by the combination of the following three mechanisms. First, individual encryption keys can be assigned to each task; second, users may express constraints on which tasks can be resumed where; and third, secure file systems guarantee that only the applications involved in supporting a task see the contents of the manipulated files.

Assigning individual encryption keys to each task is the base to guarantee that granting access to one task does not compromise the contents of other tasks. Those keys may be used to encrypt both the task model and the materials manipulated by task. For instance, suppose that a user, Fred, wants to work on reviewing a video clip at a café. Upon Fred's authentication, only the key for reviewing the video clip needs to be granted to the local infrastructure so that Fred can resume his task. Even if the (untrusted) infrastructure or other software components at the café manage to read other files pertaining to Fred, they lack the encryption keys to make sense of those files.

Furthermore, if keys are made to expire so that new keys need to be granted for every access to a task, the log of key requests can be used to detect unauthorized access attempts.

Expressing constraints on which tasks can be resumed where facilitates the automatic distribution of keys upon user authentication. For instance, once Fred authenticates at the café, the Dashboard described in Chapter 5, Tasks at a Glance, obtains access only to the tasks that are pending and enabled at the café.

The decision we took is that such constraints provide a default, but do not prevent authenticated users to browse and resume any of their tasks. For instance, Fred may decide that he needs to resume a task he would not normally work on at the café. For that, he provides the local infrastructure with the necessary information to obtain the corresponding access keys.

Secure file systems guarantee that only the applications holding a valid key see the contents of encrypted files. These file systems keep only encrypted data on permanent storage and decrypt/encrypt the data into the memory space of the application holding the access key (e.g. [32]). For instance, if a supplier is granted a key to manipulate a material in the file system, only that supplier sees (in its memory space) the decrypted contents of the file. Even if other software in the environment obtains the location of the material in the file system, it will not be able to access its contents. A similar argument can be made for accessing data streams over the network, for instance, by using https.

Together, these three mechanisms guarantee that users control which tasks are accessed by which computing environments, and that once a task is accessed, only the infrastructure and the suppliers directly involved in supporting the task access the associated information.

Mechanisms such as discussed above are available from other work on authentication, security and privacy, and incorporating them into the task management infrastructure is a matter for further work.

Software Engineering of Service Suppliers

There is one assumption made by legacy application designs that most significantly stands in the way of integrating such applications into scalable task management systems. That assumption is that an application is primarily used by one user, and that a user will primarily use the same application to obtain a given service. In other words, the assumption is that there is a one-to-one mapping between users and applications.

However, application design should assume that such mapping is many-to-many. A mobile user may encounter many applications that could provide the desired service at different locations. Conversely, an application running on a device sitting at a shared space, such as a meeting room, or a coffee shop may provide services to many users over the course of a single day.

There is an immediate design consequence of adopting a many-to-many assumption: the persistent state of a service needs to be transferred among the applications providing that service.

When assuming a one-to-one mapping, persistent state such as user preferences and application settings can simply be stored internally and recovered the next time the (same) user starts the application. Furthermore, persistent application data, such as documents being edited, may be stored in proprietary formats.

In contrast, when a mobile user resumes a task at some new location, the application chosen to provide a service in the task needs to recover the persistent state captured when the user last worked on that particular task (potentially using some other application at a different location).

For enabling the transference of the persistent state of services, two design requirements need to be met: first, applications need to expose APIs for exporting and importing the persistent state; and second, the representation of the state needs to be intelligible by other applications providing the same service. The rest of this subsection focuses on the APIs, while the following subsection discusses issues associated with the representation of the service state.

Many modern applications expose APIs for setting and capturing the state of the application (often via standards such as COM or CORBA [22,23]). In our experience, the effort for developing a new supplier by wrapping such an application is about two weeks time-on-task for an experienced student. Specifically, this includes implementing the capture and recovery of basic user-perceived state. For example, the user-perceived state for a web browser would include the current page, navigation history, window position and size, scroll, etc.

Other applications define no clear APIs for that purpose but publish the application's source code. For wrapping such applications, the amount of effort varies widely with the complexity of the code and the prior experience of the person doing the wrapping.

However, the application market is maturing in the sense that increasingly more applications include documented APIs for setting and capturing the application's state.

Furthermore, to make it easier to develop Supplier wrappers for legacy applications, we developed a generic set of classes (both in Java and C++) that support the exchange of messages with Prism and the EM, and a generic representation of the service state to be mapped. Integrating an application as a service supplier in the infrastructure consists of extending those generic classes and interfacing with the application's specific APIs. This is joint work with Vahe Poladian.

Controlling the resource adaptation policies of an application proved to be more challenging (e.g., for Media Player and Xanim). These applications tend to fall into two fields: first, those coming from research or open-source projects, for which controlling the policies, although possible, can involve considerable effort.[25] Second, commercial software, which either doesn't expose APIs to control the adaptation policies, or for which we could not observe a reliable correlation between the controls transmitted to the application and its actual behavior.

However, the application market seems to be maturing also in this respect: in recent experiments with RealOne Player we could observe a good correlation between the control knobs for the resource-adaptation polices and the application's actual behavior.

Representation of Service State

The second design requirement for enabling the transference of the persistent state of services (see previous subsection) is that the representation of the state needs to be intelligible by other applications providing the same service. Here, we can distinguish two parts to the state of services: first, preferences and settings, such as window size and cursors, and second, persistent application data, such as files being edited.

Representing preferences and settings in XML (see Chapter 3, State Snapshot) enables that part of the state to be manipulated by applications with different degrees of sophistication. Specifically, sophisticated suppliers may interpret all or most of the settings, while simple suppliers interpret only the settings they know about and leave others undisturbed. For example, suppose that a user, Fred, is editing a text document with an editor that supports spellchecking and that keeps a representation of the settings associated with spellchecking; for instance, which language to check against. Later, Fred resumes this task on a PDA using a simple text editor that cannot instantiate the state pertaining spellchecking. However, once Fred returns to the more sophisticated environment, that part of the state can be interpreted and recovered.

The compatibility of data formats is arguably the hardest problem associated with making the representation of the service state intelligible across applications. In the example above, suppose that the sophisticated text editor in the first environment supports rich formatting. When Fred resumes his task on the PDA, the available editor doesn't support rich formatting and Fred allows a conversion to a simpler format that the unsophisticated editor can handle. Back at the sophisticated environment, Fred now has the previous version of his document containing the painstakingly added formatting, but none of the updates, and the new version of the document containing the updates, but none of the formatting. Clearly, this is a problem that one-shot format conversion cannot address.

[25] Other research is addressing the problem of reducing the effort involved with extending applications for adaptation [9].

Ideally, persistent data pertaining to a service would be seamlessly accessible by any application providing that service: simple applications would access the basic data (e.g. plain text) while sophisticated applications would additionally access the more sophisticated information (e.g. text formatting).

One possible step in this direction is to keep a log of operations whenever a supplier needs to make a conversion to a simpler format. That log of operation can be replayed (automatically, to the extent possible) back at a sophisticated environment, so that the operations affect all layers of data. However, developing effective strategies for representing and accessing persistent data across applications with different degrees of sophistication is still an open problem, and a matter for further work.

Interconnection of Service Suppliers

How service interconnection is supported plays an important role in the QoS of complex configurations. For instance, suppose that to carry out speech-to-speech translation, three services are interconnected: *speech recognition*, *language translation*, and *speech synthesis*. How the results of the speech recognition are passed to translation, and how the results of translation are passed to speech synthesis is key to the overall latency of the service. Furthermore, suppose that the secrecy of the translated utterances is a concern: the availability of adequate mechanisms in the interconnections is key to offering overall assurances.

Legacy applications may have their own mechanisms for interconnection, typically taking advantage of standard mechanisms available at the operating system or networking levels. As discussed in Chapter 4, Heterogeneity, legacy applications are made available to the infrastructure as service suppliers by wrapping them with Supplier code.

The wrapper code around legacy applications may have access to controlling the application's mechanisms for interconnection via the APIs, or may resort to interconnect applications via standard operating system mechanisms: e.g. piping the input/output streams of the applications, or taking the data and sending it across a TCP connection.

A question that arises is: could the infrastructure change the QoS properties of the connectors in a configuration of services? For instance, by substituting increased performance connectors for default ones, or by adding secrecy guaranties to existing connectors that provide none.

Research in the area of software architectures addresses modifying and adding properties to existing connectors by wrapping [86]. Incorporating such results and clarifying the roles of centralized reasoning in the Environment Manager vs. specialized reasoning in the Supplier wrappers is a matter for further work.

Describing and Operating on Tasks

This section focuses on important decisions we took concerning the mechanisms available for users to describe and operate on their tasks. The two subsections below discuss issues related to the impact of the principle of offering incremental benefit for incremental effort, and to the approach we chose for enabling users to find past tasks.

Incremental Benefit for Incremental Effort

A fundamental guiding principle that we adopted is that of offering incremental benefit for incremental effort (sometimes called *gentle slope systems* [61]). With the interfaces we designed for describing tasks, users are able to reap significant benefits from task management, even with little effort put into describing their tasks. Furthermore, the more effort users are willing to put in describing their tasks, the better job the infrastructure for task management can do.

Specifically, users can benefit from suspending and resuming tasks just by associating the relevant files and services with the task definition. When a user provides such a minimal task description, the user preferences for that task take on default values. However, distinct user preferences may be associated with each service within each task.

The infrastructure currently includes predefined preference templates for each service type. Such templates specify both supplier preferences and QoS preferences. At least one template, the default, is provided, and depending on the service type, more templates may be available. For instance, for web browsing, an additional *fast* template is currently defined, which specifies strict constraints for the response time but is tolerant of not loading pictures.

Whenever the preferences specified by the default template fail to capture the user's intent for the service within the particular task, the user may quickly associate another set of preferences with the service by selecting another template, as illustrated in Figure 5.5 (a). If the user is not satisfied by the preferences in the available templates, he may fine-tune the preference specification for the service, as illustrated in Figure 5.5 (b,c).

The current implementation reads the template definitions from configuration files, which may be edited for personalizing and adding new templates. Extending the task definition interfaces to make personalizing templates easier for non-expert users is a matter for further work.

Finding Past Tasks

The ability to find past tasks plays an important role in the usability of an infrastructure for task management. Although sophisticated mechanisms are currently available for finding files in the file system, such as the Google Desktop Search [39], such mechanisms are not a substitute for finding the task definitions themselves. Tasks may need to be referred back to, and restarted, long after the user thought they were done. For instance, a user may need to find a presentation given last year, so that the associated spreadsheet can be updated with the latest data, and the slides updated with new developments for an upcoming meeting.

There are two main alternatives for finding past tasks. The first is to provide a time slider that allows users to backtrack time (much like a time slider in video playing software); the second is to provide mechanisms for browsing tasks based on information related to those tasks.

Equipping desktop managers with a time slider enable users to revert the aspect of their desktop to the specified time. This approach is convenient for small time scales, but it becomes awkward for time scales of months or years, especially if users don’t have a precise idea of when they worked on the tasks they are looking for. Worse, while this concept makes sense if all the tasks take place on the same desktop, it might be problematic to reconstitute the state of past tasks that took place across different locations, and maybe in computing environments with different capabilities than the current one.

The decision we took, embodied in the Lamp component described in Chapter 5, was to support finding tasks by browsing. Users may associate circumstantial information to tasks, thus establishing a persistent task identity that constitutes the basis for browsing.

In addition to the pragmatic aspects above, we believe that allowing users to browse their tasks highlights the notion of *task* as a first class entity, as opposed to a time cursor, in which users manipulate *time* as a first class entity to reach their tasks.

Chapter 8

Conclusion and Future Work

This dissertation investigated the use of high-level models of user needs and preferences for scaling task management in space and in time. Past research has employed user models for achieving other goals; for instance, guiding users along complex tasks, helping designers during the analysis and development of systems that support specific user tasks, and automatically deriving user interfaces adapted to the characteristics of diverse devices (see Chapter 2). Broadly, that research had mixed success, mostly because the benefit for end users was not always clear: often the goal of employing such user models was to constrain the behavior of end users, or to reduce the cost of system development.

My research focused specifically on reducing user overhead with task management across heterogeneous computing environments and across long periods of time. The importance of improving the experience for end users has increased dramatically with the developments in ubiquitous computing over the past decade. Users are increasingly surrounded by devices capable of computing in many forms: portable personal computers and organizers, cell phones, media and game consoles, processors embedded in cars, etc. Thanks to connectivity, which is quickly becoming universally available, many of those devices are already able to browse information on the web, and will increasingly be used to store and access personal information.

I believe that the next decade will bring a significant shift in the way software is designed. Today, software is application-centric. An application assumes that one user will be using it, and that the user will always use that same application to carry out his tasks. Consequently, each application places itself in the center of the universe, and users must approach each and reconcile the differences to carry out their tasks.

In the future, software will be user-centric. Users will use many applications running on many devices indistinctively to carry out their tasks. And applications sitting on shared spaces, such as meeting rooms and coffee shops may serve the needs of many users over the course of a single day. Therefore, models of what users need and prefer for each of their tasks will play a key role in coordinating the configuration of devices and software for supporting such tasks.[26]

This dissertation takes a step in this direction. Specifically, it introduced an approach for scaling task management in space and in time that is based on high-level models of what users need from the computing environment for each of their tasks. Such models are exploited at run-time by an infrastructure that automatically configures the environment on behalf of users.

This approach reconciles the competing requirements of reducing the overhead incurred by users when configuring computing environments, while simultaneously enabling users to take full advantage of the environments around them at different locations.

The scalability of task management was demonstrated by defining models of user tasks, and by building an infrastructure that supports scalable task management. Chapter 3 specified how to build models of user tasks, while Chapters 4 and 5 described an architectural framework for scalable task management and an infrastructure that fits that framework. The infrastructure supports task management in the sense that it supports suspending and resuming user tasks as a coordinated set of services in the environment. Scalability in space is supported by allowing users to suspend tasks at one location and to resume them at another location at will, provided an installation of the infrastructure is available. Scalability in time is supported by allowing users to browse their tasks regardless of how long ago those tasks were defined, or completed. Chapter 6 argued how the models and infrastructure support scalable task management.

Chapter 6 also demonstrated how the infrastructure reconciles two competing requirements: reducing user overhead while simultaneously enabling users to take full advantage of the environment. Taking full advantage was demonstrated by construction: the utility-theoretic framework presented in Chapter 4 enables the infrastructure to find the best match between the user's needs and the available components and resources in the environment. This utility-theoretic framework works based on the precise modeling of user needs and preferences described in Chapter 3. Chapter 4 also described the internal workings of the infrastructure for monitoring and adapting to dynamic changes, thus keeping the optimality of the match.

Reducing the overhead for users was demonstrated in Chapter 6 by comparing the overhead of interacting with the infrastructure against the overhead of interacting with the raw environment. We compared against users configuring the environment themselves because that represents the state of the art for scaling task management in space across heterogeneous environments. Specifically, we isolated relevant configuration tasks – defining, suspending and resuming main tasks – and introduced a quantitative model for comparing user overhead.

[26] Furthermore, while the models investigated in this dissertation focused on tasks concerning information processing, with services such as text editing, in the future, user models will be exploited to actuate on the physical context of the user; for instance, to set the preferred temperature of an office, or to automatically adjust the settings of a chair in a meeting room.

Starting with an analysis independent of scalability, we have shown how the benefits of using the infrastructure accrue with the number of suspend-resume cycles, and how those benefits are more significant the more services and materials are involved in a task. Then we discussed how scaling task management in space, across heterogeneous environments, increases the benefits of using the infrastructure and how scalability in time does not significantly increase the benefits already gained with automating task management and with scaling it in space.

Overall, this dissertation evaluated our research from three perspectives. First, Chapter 6 evaluated our approach for task management by validating the research thesis. Second, Chapter 7 evaluated the architectural framework that supports our approach by discussing the benefits and limitations of the framework and its current implementation. And third, Chapter 4, Evaluation, evaluated the performance of the infrastructure that implements the framework.

Contributions

This dissertation offers contributions at three levels: an approach to scaling task management in space and time, an architectural framework that supports the approach, and an implementation of the framework. The following subsections elaborate on the contributions of each of these.

Contributions of the Approach

The main contribution of my research is demonstrating that high-level models of user tasks can be exploited for scaling task management beyond traditional office environments. Below, we elaborate on the specific contributions of the approach, which hold regardless of the specific architectural framework defined in this dissertation, or the infrastructure we provide:

- **Scaling in space**. We have shown how to build task models that reflect user needs and preferences in a way that is independent of the specifics of each environment. Specifically, tasks are described as a coordinated set of abstract services, such as *editing text*, rather than a set of applications, such as MSWord. Additionally, these models capture abstract representations of the user-perceived state of the services and of the user preferences relative to quality of service tradeoffs. We have also shown how these abstract models can be dynamically mapped into the specific capabilities of each environment. These ideas enable scaling task management in space, across heterogeneous environments, in the sense that users may suspend a task in one environment, and resume it in another environment running a different set of applications and devices.
- **Scaling in time**. We have shown how to endow task models with circumstantial facts about the tasks they represent so that algorithms akin to web browsing can be applied to that information. This idea enables scaling task management in time, in the sense that it allows users to find and recover tasks defined or completed long ago.
- **Reconciling two competing requirements**. We have shown that an automated task management system reduces the overhead associated with configuring the computing environment for each task. We have shown how a task management system can find the best

match between user needs and the capabilities of the environment. Putting these two ideas in practice by the same infrastructure reconciles the requirements of reducing the configuration overhead, while simultaneously enabling users to take full advantage of the environments around them at different locations.

– **Offering control over which tasks to resume**. We have shown how having explicit task models enables task management systems to discriminate individual tasks. We have shown how offering users the equivalent of a to-do list, or task dashboard, enables them to control which tasks to resume on different circumstances. These ideas enable environments to save resources, by activating only the services required by the tasks that users intend to work on. We have also discussed how privacy mechanisms can associate different access keys to different tasks. On top of the ones above, this idea enables users to control which files are accessed from which environments, allowing for sensitive materials to be manipulated only by trusted environments.

Contributions of the Architectural Framework

The architectural framework that supports our approach clarifies the responsibilities and interaction protocols between the components of an infrastructure for task management. Here we list the contributions of the framework, which hold regardless of the specific infrastructure we provide for implementing the framework:

– **Task Management layer**. We have shown how to obtain knowledge about user needs and preferences for each task, and how to represent it in a new software layer for task management. We have described the protocols to disseminate this knowledge thus coordinating resource allocation and adaptation policies in service suppliers. Designs that rely on ad hoc mechanisms inside applications to capture knowledge about user tasks make it very hard to have a consistent view across applications, and to transfer that knowledge to a different set of applications when a task is resumed in an another environment. In contrast, our design promotes a consistent system-wide awareness of user needs and preferences, and makes it easy to disseminate that knowledge wherever it is needed.

– **Environment Manager**. We have shown how to obtain an environment-wide view of the capabilities of a computing environment, and how to update that view in the face of dynamic changes. We have also described how the EM gathers information about the quality of service being delivered by active service suppliers. The EM plays a key role in finding the best match between abstract representations of user needs and the concrete capabilities of each environment. The EM also plays a key role in detecting and adapting to coarse-grain changes in the environment, such as changes in the availability of suppliers, and to trends in resource availability.

– **Utility-theoretic framework**. We have shown how to quantify user preferences and how to exploit that in a utility-theoretic framework for finding the best match between user needs and preferences, and environment capabilities. Specifically, we quantified preferences relative to alternative service configurations, relative to alternative suppliers for each service, and relative to multiple dimensions of quality of service. Separate research provided the algorithms that exploit the utility-theoretic framework to derive the optimal assignment of suppliers to requested services, the optimal resource allocation among those suppliers, and the optimal fine-grain resource-adaptation policies within those same suppliers.

- **Coordinating dynamic adaptation at three levels**. We have investigated the nature of changes related to task management and separated them into three levels: changes in user tasks, changes in the availability of service suppliers and coarse-grained trends in quality of service, and fine-grain changes in resource availability. We have defined an architectural framework for monitoring and reacting to these kinds of changes, and clarified the responsibilities and interactions between the layers responsible for each. We have shown how to incorporate the utility-theoretic framework mentioned above for coordinating the overall behavior of this architectural framework.

Contributions of the Infrastructure

The *infrastructure* that implements the architectural framework provides a working foundation for research in task management and complementary areas. Here we list the contributions of this infrastructure:

- **Demonstrates the feasibility of the approach and features of the framework**. We have shown that the approach is feasible by providing an infrastructure for task management that supports the anticipated capabilities. This infrastructure also demonstrates how the features of the architectural framework deliver practical benefits: the infrastructure not only delivers the features, it does so with a performance that makes it usable on a daily basis.
- **Demonstrates usability by non-experts**. By carrying out user studies with a general academic population, we have shown that people other than the author can understand and manipulate the concepts in the approach. Specifically, by deploying the infrastructure with about 10 voluntary students and researchers who used it freely, we could confirm that users can define, suspend and resume tasks with the infrastructure. Furthermore, scripted user studies performed with a general academic population confirm that users can understand and manipulate quality of service tradeoffs in environments with fluctuating resources.
- **Provides a foundation for research**. By providing a working infrastructure with a clear and well documented design, we make it possible to reuse it, or its components, for future and complementary research. The infrastructure imposes a low buy-in cost to future extensions since it makes no assumptions on their internal structure or programming language. The only constraint to such extensions is that they be able to exchange XML messages with the existing components over TCP. Within this constraint, the infrastructure supports the affordable integration of legacy applications by wrapping them with code that translates between their specific programming interfaces (APIs) and the infrastructure's communication protocols.

Future Work

Although desktop management has been around for two decades, the changes brought about by the increasing pervasiveness of computing make task management an exciting research area.

The increasing pervasiveness of smart spaces is causing a shift in the paradigm of computer use: from single-device, tightly integrated interaction, to multiple-device, loose interaction. The physical environment around users is increasingly endowed with sensors and actuators, and computer-supported tasks are blending with daily activities and artifacts, such as cell phones, cars, TV sets and refrigerators.

These changes also stimulate research in complementary areas, such as the management of adaptation to changes in the computing environment (see for instance work by Vahe Poladian and Shang-Wen Cheng [20,70]), the development of resource-aware applications (see for instance work by Rajesh Balan [8,9]), as well as in other areas such as context-aware systems and natural human-computer interfaces (e.g., language, vision, and gesture).

Focusing on the future development of the work herein, I envision extending the architectural framework towards supporting more complex user tasks, and towards clarifying the software engineering requirements for applications as service suppliers in this framework.

Specifically, I envision research on extending the role of the infrastructure for mediating the delegation of tasks, on scoping the computing environment, on task-aware proactive suppliers, on offering guidance on complex tasks, and on learning complex task models. The following subsections elaborate on each of these topics, in turn.

Delegation

The problem that we addressed in this dissertation can be characterized as automatically creating the conditions for a task to be carried out, on demand. By creating the conditions we mean activating a coordinated set of services in a computing environment, for which suppliers need to be identified and configured. The entity creating those conditions we called "infrastructure for task management," or task management system.

In this dissertation, just like in traditional desktop management systems, humans appear exclusively as users of the system. Users define tasks with the task management system, request tasks to be suspended and resumed, and the role of users in the tasks is not characterized further.

Taking a richer view of daily activity, humans frequently *delegate* parts of tasks to other humans. For instance, suppose that a user, Fred, needs to prepare a presentation. For that, Fred may edit the slides, and refer to a couple of papers on the topic. Now suppose that some of the papers Fred is interested in are available only as archived hardcopies. Creating the conditions for Fred's task requires physically getting the hardcopies from the archive: maybe Fred can get them, maybe Fred's assistant, or maybe the office robot. In our framework, this corresponds to considering humans as candidate suppliers for the service of getting the paper hardcopies from the archive. Which supplier gets assigned to the service may depend on properties such as proximity, availability, and willingness, if known.

But if humans are to participate in tasks as service suppliers, the infrastructure must have the mechanisms to guide and monitor that participation. Although from a high-level perspective in task management, guiding humans is akin to configuring an automated supplier, it is sure to pose a new set of problems. A task management system can activate and configure a piece of software by using its API; but how can it “activate” and “configure” a human supplier? Maybe Fred’s assistant is amenable to getting the hardcopies if asked to do so via an appropriate notification, but how can the task management system monitor whether Fred’s assistant is working on that, or has forgotten about it?[27]

The challenge becomes developing a single framework that successfully combines the ability to guide humans as participants in tasks, with the ability to automatically configure computing environments for those same tasks. Guiding human participation in tasks has been addressed in other areas, such as business process modeling and generic task modeling systems (e.g., [36,81,100]). However, in that research, the responsibility of acting on the environment to create the conditions for those tasks was also left to humans.

And if humans can both play the role of user of a task management system and participate in a task as a service supplier, could an automated “agent” transcend the role of service supplier and become a user too? In other words, can an automated “agent” request the task management system to define, suspend and resume a task? And if the task requires human intervention, can we think in terms of the infrastructure *delegating* a task to a human?

Dedicated systems, often embedded, do that today: for instance, door bells, fire alarms, etc. A part of the system is charged with the task of continuously monitoring a particular situation, and upon detection triggers a new task, normally involving human participation. In the door bell example, upon detection of a visitor, some notification is issued to a human, who then proceeds to open the door. Although trivial, this example illustrates the triggering of a task by an automated agent. The triggered task may involve a combination of automated suppliers, e.g. for the bell notification service, and humans, e.g. for getting the door.

Extending task management to facilitate delegation of tasks, whether the originator of tasks is a human or an automated “agent,” is an important and hard problem. Task delegation is prevalent in human organizations, and having automated support to facilitate and control task delegation can play an important role in reducing the overhead for users and improving overall productivity and accountability.

Scoping the Environment

When addressing the problem of automatically creating the conditions for a task to be carried out, an important matter is how far to search for service suppliers; or in other words, what the limits of the computing environment are.

In this dissertation we assumed that an environment’s borders are set administratively, much like the borders of local network domains are set today: when an Environment Manager searches for candidate suppliers for a given service, it restricts the search to its administratively set domain. Internet searches stand in stark contrast: any search is a global search, although web search engines impose their own search heuristics. DNS (Domain Naming Service – the internet

[27] The completion of tasks and flow between related tasks is a separate idea, and is discussed in the subsection on Guidance on Tasks.

service that translates domain names into IP addresses) takes a hierarchical approach: when a local server cannot perform a translation, it asks the upper level, and so forth.

However, it is not clear that such a hierarchical approach is well suited for supplier searching in the context of task management. In fact, there is a tension between organizing searches according to supply-side priorities (organizational or geographic boundaries), and organizing searches according to demand-side priorities (what is important to each user).

Specifically, mobile users may prefer to concentrate on suppliers that are familiar, or accessible to them, and for that, searches may need to work seamlessly across administratively set boundaries. For example, a user working at a coffee shop and looking for a printing service may want to consider a printer at the office, where the hardcopies will be needed in the morning, even though there are no organizational links between the two locations.

However, although the demand-side priorities may help directing the search to suppliers that users will consider attractive, one cannot expect users to have full knowledge of the possibilities available to them. Therefore, searches also need to consider supply-side knowledge about suppliers that may be attractive to the user, whether or not he knows about them.

Scoping the search for suppliers is an increasingly important problem for mobile users, since the availability of commodity computing is becoming more prevalent. To address this problem, we need to develop a framework that successfully integrates both demand-side (user) priorities and supply-side (environment) information.

Proactive Suppliers

Another matter that comes up when addressing the problem of automatically creating the conditions for a task to be carried out is how readily available are the services that users need. This matter gains more relevance when the services go beyond manipulating software components to actuate on physical properties in the environment surrounding users.

In the architectural framework presented in this dissertation, service suppliers are essentially passive. Suppliers wait to be activated by the Environment Manager, and after that they accept information from the Task Manager about the user-perceived state to be set and preferences with respect to quality of service. This approach is adequate when the services can be set up quickly, as was the case for the software applications we worked with in the office environment.

However, how fast a service can be set up may become an issue, for instance, for services where the user-perceived state includes physical properties, such as the temperature of a cup of coffee. In the approach that we took so far, an Environment Manager may factor in the warm up time of alternative suppliers when finding the best match for the user's needs (see Chapter 3, Supplier Preferences). Nevertheless, there is no assurance that in the end the user will be getting the best service, compared to an alternative where the warm up delay could be eliminated.

An alternative approach would be for suppliers to proactively obtain information about prospective services and to prepare for those. Suppliers may ask the Task Manager what are likely requests in the near future, or they can use internal mechanisms to anticipate future requests: for instance, history-based detection of patterns of usage. Suppliers may then prepare in advance for the anticipated demand and publicize the availability of the customized service.

However, preparing in advance may carry a risk for suppliers: resources may be wasted if the demand doesn't come forth, or suppliers may miss the target if the demand that comes forth is significantly different than anticipated. Resources in computing environments are increasingly abundant and wasting may not be an issue, especially for resources that get wasted anyway if not used, as is the case for CPU cycles. As far as energy, warming up a cup of coffee may be inconsequential if the user ends up not drinking it, but warming up a whole house in anticipation of people coming home at a certain time may become an issue if they come in later, or not at all.

Furthermore, proactive suppliers may pose unwanted overhead on users. Even well-meant efforts of proactive suppliers may hinder more than help users, if the users' goals are misunderstood and the suppliers' actions have perceivable effects in the environment.

Balancing the advantages and risks of proactive suppliers is a hard problem that is likely to require sophisticated mechanisms both to anticipate user needs and to guard against taking action when there is a risk of being more harmful than helpful.

Guidance on Tasks

In addition to automatically creating the conditions for a task to be carried out, another important problem to be addressed by task management systems is offering guidance on which tasks to carry out. Such guidance takes two basic forms: either passive guidance, where users take the initiative to consult the task management system, or active guidance, where the task management system takes the initiative to resume or suspend tasks without the direct intervention of humans.

In this dissertation we adopted a simple model of tasks, where there is very restricted information about the chaining of tasks, about the association of tasks with particular physical contexts, or about the timing of tasks. Although task models include links for related tasks, the mechanisms on top of that targeted task browsing and not offering guidance on which tasks should be worked on. The Dashboard presented in Chapter 5, Tasks at a Glance, advises on the feasibility of tasks in the current environment, and we discussed how it could restrict the view to tasks *enabled* at a particular location, but again there is no base for inferring which tasks should be *favored* in a particular context. Finally, the information about a task kept timestamps for the creation and due date, but again the mechanisms on top of that targeted task browsing and not offering guidance based on timing constraints.

Tasks frequently have interdependencies such as causality (if task A is carried out, task B should also be carried out) and ordering. A significant body of work addresses complex models of tasks, and a frequent way of modeling complex tasks is task decomposition: carrying out task C really corresponds to carrying out tasks A *and* B – as far as the problem of creating the conditions to carry out a task is concerned, only the tasks that stand as leafs in a decomposition need to be addressed. Commonly, this work takes a prescriptive view of tasks, which is adequate to guide automated "agents" in carrying out complex tasks [82,88], or, although prescriptive, the models are meant to be interpreted only by humans [36,100].

However, the way humans carry out tasks is often haphazard, riddled with interleaving, and heavily influenced by physical and social context. The human brain is trained in extracting the important rules from a model and adapting task execution to changing circumstances, or simply to its whims. Even a technician doing something as structured as a biology experiment in a lab may interleave several experiments, doing manual operations for one while waiting for some equipment to produce results for another; he may change the usual procedure if he thinks of an alternative way of achieving the desired goals [6]; or he may drop what he is doing at a loss of the results, if a higher priority request comes in.

The challenge becomes to successfully combine human goals and behavior with the automated actions of a task management system. It is risky to offer unsolicited guidance: on one hand, the task of responding to a fire situation should be started the moment a fire is detected; on the other hand, upon detection that the house cat ran out of food, the task of refilling the cat feeder can wait until someone is nearby the cat food, or until enough time has gone by for that to become a problem. Even in restricted areas in the office domain, offering unsolicited guidance is notoriously hard, as efforts like Microsoft's Clippy demonstrated [44].

Rather than having a one-size-fits-all approach to guidance on tasks, users should be able to decide on which tasks[28] guidance should be offered, and which to be let freeform. Furthermore, users should be able to determine levels of guidance, such as: provide active guidance (that is, let the task management system take the initiative to start or stop tasks automatically), issue an unsolicited reminder, issue a reminder only if the user takes no action after some time, issue a reminder only if asked for clarification, etc.

Learning Task Models

Bearing in mind the goal of reducing user overhead in the context of task management, an important aspect is the overhead associated with defining task models. This aspect gains more importance the more complex the task models become; for instance, when task models aim at capturing enough knowledge to offer guidance during complex tasks. And since task definitions evolve as work progresses, work on this aspect must focus both on the initial definition and on incremental updates.

In this dissertation, the task management system relied on users to explicitly define their tasks and preferences; and we discussed a quantitative model of the overhead associated with defining the set of services and materials in a task in Chapter 6, Reducing the Overhead.

Previous work targeted learning models of tasks by observation (e.g., [4,80,98]), but it is very hard to learn task models by observing the natural behavior of users (see discussion in the

[28] Taking the perspective of task decomposition, offering guidance on a complex task corresponds to guiding a user through the decomposition steps. Taking the perspective of flow at the level of the leafs (the ones that matter for the automatic configuration of the environment) guidance is with respect to the *transitions* between tasks: when one gets done, what should the user work on next.

subsection Guidance on Tasks, above). A more promising approach relies on a training phase, where users walk the system through tasks, in a similar way they would for a novice person.

There seem to be advantages in making training explicit. By making it easier for the system to learn task models, and to separate what belongs to each task, the system can do a better job at automatically creating the conditions for those tasks later on. By making it clear for users when the system is being trained (a) it builds a foundation of shared assumptions, (b) it calibrates the expectations of users, and (c) it makes it possible to distinguish exceptional situations that are unlikely to be repeated from deliberate incremental updates to the task definition.

But if training is to be made explicit, the next question is what training metaphor to use. The overhead of training is likely to be less significant the closer the interactions for training the system are to the natural interactions for carrying out the tasks. For instance, in the desktop paradigm covered in this dissertation, it is natural to drag and drop files into a task definition window, since that's the kind of interaction familiar to users in the office domain. However, for tasks carried out using the instruments in a biology lab, or using artifacts in a home, it may be awkward to present an interface for defining tasks that follows the same interaction metaphors as the one for the office domain.

The research challenge is to develop a framework for learning task models that accommodates appropriate metaphors for a range of domains, that makes it easy for users to train the system, and that delivers incremental benefits for incremental effort.

Epilogue

The research underlying this dissertation used extensively basic principles and techniques from software architecture and formal specification. We used a variant of context-free grammars to specify the syntax of task models, and utility-theoretic models to specify its semantics (Chapter 3). We used layered and component-connector models to lay out the software architecture of the infrastructure for task management. We used event-sequence diagrams and FSP to specify the interactions between the components of the infrastructure, and Zed to specify the state changes in those components as a result of the interactions. We used math again to specify how to find the best match between user needs and environment capabilities (Chapter 4, [85]). Finally, we used a quantitative model for the cost-benefit analysis of using the infrastructure (Chapter 6).

Although that is not reflected in the thesis statement or its validation, the contribution of those basic principles and techniques for this research is reflected in the soundness of the research results and in the usefulness of the contributions.

A specific principle being investigated in software architecture, and extensively used in this dissertation, is an utility-theoretic approach to designing the programming interfaces (APIs) of software components. Traditionally, components would deterministically provide a set of outputs given a set of inputs; any deviation to the expected outputs would signify that an error occurred. With the popularization of applications streaming media over networks, it became obvious that the quality of output varies with factors uncorrelated with the inputs, namely with the available resources. Database queries in mobile environments became another arena for experimenting with this principle: depending on the available resources, which may include how long the user is willing to wait, the reply will be more or less complete, or accurate. For instance, if asking for a maximum value, the reply may be only approximate if not all records could be scanned.

In this dissertation, the interactions with service suppliers take root in utility-theoretic principles: service suppliers announce the achievable levels of quality of service (possibly multi-dimensional) as a function of available resources; and choosing one supplier over another is based on the forecasted quality of service for the current conditions.

This dissertation contributes with a data point for the applicability and usefulness of both the basic principles and the specific utility-theoretic approach to designing software components. The long-term validation of the research results herein will confirm the validity of this data point.

References

1. G. Abowd, A. Bobick, I. Essa, E. Mynatt, W. Rogers: The Aware Home: Developing Technologies for Successful Aging. *Proc. of AAAI Workshop on Automation as a Care Giver*, Alberta, Canada, July 2002.
2. G. Abowd, E. Mynatt: Charting Past, Present and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction*, Vol. 7:1, pp 29-58, March 2000.
3. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, J. Lilley: The design and implementation of an intentional naming system. *Proc. of the 17th Symposium on Operating System Principles*. Kiawah-Island Resort, North Carolina, December 1999.
4. D. Albrecht, I. Zukerman, A. Nicholson, A. Bud: Towards a Bayesian model for keyhole plan recognition in large domains. *Proc. 6th Int. Conference on User Modeling (UM '97)*, pp 365-376. Wien, Jameson, Paris and Tasso (Eds.) Springer, New York, 1997.
5. K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, A. Wollrath: *The Jini Specification*. Addison-Wesley, 1999. See also http://www.sun.com/software/jini/
6. L. Arnstein, S. Sigurdsson, R. Franza: Ubiquitous Computing in the Biology Laboratory, *Journal of Lab Automation (JALA)*. Vol. 6:1, March 2001.
7. S. Baker, T. Kanade: Hallucinating faces. *Proc. of the Fourth International Conference on Automatic Face- and Gesture-Recognition*, Grenoble, France, March 2000.
8. R. Balan, M. Satyanarayanan, S. Park, T. Okoshi: Tactics-Based Remote Execution for Mobile Computing. *Proc. of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys'03)*, pp 273-286, San Francisco, May 2003.
9. R. Balan, J.P. Sousa, M. Satyanarayanan: Meeting the Software Engineering Challenges of Adaptive Mobile Applications. *Carnegie Mellon University Technical Report CMU-CS-03-11*, 2003.
10. R. Balan, J.P. Sousa, V. Poladian, M. Satyanarayanan: Guiding Adaptation through User-Specified Tradeoffs. Submitted for publication.
11. L. Bannon, A. Cypher, S. Greenspan, M. Monty: Evaluation and analysis of user's activity organization. *Proc. of CHI'83*, pp 54-57, ACM, New York, 1983.
12. A. Berson. *Client/Server Architecture*. McGraw Hill, 1996.
13. K. Bharat L. Cardelli. Migratory applications. *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, Pittsburgh, Pa., November 1995.
14. T. Bourke. *Server Load Balancing*. O' Reilly, 2001.
15. B. Brumitt, et al.: EasyLiving: Technologies for Intelligent Environments. *Proc. 2nd Int'l Symposium on Handheld and Ubiquitous Computing (HUC2000)*. LNCS 1927 pp 12-29. Gellersen, Thomas (Eds.), Springer-Verlag, September 2000

16. P. Brusilovsky. Adaptive Hypermedia, User Modeling and User Adapted Interaction, Ten Year Anniversary Issue, Kobsa (Ed.), Vol. 11:1/2 , pp. 87-110, Kluwer Academic Publishers, 2001.

17. J. Burrell, G. Gay, K. Kubo, N. Farina. Context-Aware Computing: a Test Case. UbiComp 2002: Ubiquitous Computing, Proceedings of the 4th International Conference, Borriello and Holmquist (Eds.), LNCS 2498, pp 1-15, Göteborg, Sweden, September 2002.

18. S. Card, A. Henderson Jr.: A multiple, virtual workspace interface to support user task switching. *Proc. of CHI+GI'87*, pp 53-59, ACM. New York, 1987.

19. S. Card, T. Moran, A. Newell: *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.

20. S. Cheng, A. Huang, D. Garlan, B. Schmerl, P. Steenkiste: Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure, *IEEE Computer* Vol. 37:10, October 2004.

21. H. Christensen, J. Bardram. Supporting Human Activities – Exploring Activity-Centered Computing. *Proc. of the 4th International Conference on Ubiquitous Computing (UbiComp 2002)*, Borriello and Holmquist (Eds.), LNCS 2498, pp 107-116, Göteborg, Sweden, September 2002.

22. COM: Component Object Model Technologies, Microsoft. http://www.microsoft.com/com.

23. CORBA: Common Object Request Broker Architecture, Object Management Group. http://www.corba.org/

24. F. Cristian: Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, Vol. 34:2, pp 56-78, ACM, 1991.

25. M. Czerwinski, E. Horvitz, S. Wilhite: A diary study of task switching and interruptions, *Proc. of the 2004 Conference on Human Factors in Computing Systems*, p.175-182, April 24-29, 2004, Vienna, Austria

26. The DAML Services Coalition (multiple authors): DAML-S: Web Service Description for the Semantic Web, *Proc. Int'l Semantic Web Conference (ISWC)*, 2002. See also http://www.daml.org/.

27. E. Dashofy, D. Garlan, A. Koek, B. Schmerl: xArch: an XML Standard for Representing Software Architectures. http://www.isr.uci.edu/architecture/xarch/

28. Deskman Desktop Manager http://www.microsoft.com/windowsxp/downloads/powertoys/xppowertoys.mspx.

29. D. Doubleday, M. Barbacci. Durra: A Task Description Language User's Manual. *Software Engineering Institute Technical Report CMU/SEI-92-TR-36*, http://www.sei.cmu.edu/publications/documents/doc.list/1992.htm, December 1992.

30. J. Flinn, E. de Lara, et al.: Reducing the Energy Usage of Office Applications. *Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

31. J. Flinn, D. Narayanan, M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. Procceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany, May 2001.

32. Flinn, J., Sinnamohideen, S., Tolia, N., Satyanarayanan, M. Data Staging on Untrusted Surrogates. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp 15-28, San Francisco, CA, 2003.

33. K. Gajos. Rascal: a Resource Manager for Multi Agent Systems In Smart Spaces. Proceedings of The Second International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS 2001), Kraków, Poland, 2001.

34. D. Garlan, R. Monroe, D. Wile. Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems, Leavens and Sitaraman (Eds.), Cambridge University Press, pp 47-68, 2000.

35. D. Garlan, D. Siewiorek, A. Smailagic, P. Steenkiste: Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, April-June 2002.

36. D. Georgakopoulos, M. Hornick, A. Sheth: An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure, Journal of Distributed and Parallel Databases, Vol. 3:2, 1995.

37. I. Georgiadis, J. Magee, J. Kramer. Self-Organising Software Architectures for Distributed Systems. *Proc. ACM SIGSOFT Wksp on Self-Healing Sys. (WOSS'02)*. Nov. 2002.

38. The GNU Emacs text editor (multiple authors). http://www.gnu.org/software/emacs/emacs.html.

39. Google Desktop Search. http://desktop.google.com/

40. R. Grimm, T. Anderson, B. Bershad, D. Wetherall. A system architecture for pervasive computing. Proceedings of the 9th ACM SIGOPS European Workshop, pp 177-182, Kolding, Denmark, September 2000.

41. A. Harter, A. Hoper, P. Steggles, A. Ward, P. Webster. The Anatomy of a Context-Aware Application. Proceedings of the Fifth ACM/IEEE International Conference on Mobile Computing and Networking, pp 59-68, Seattle, Washington, August 1999.
42. J. Hightower, G. Borriello. Location Systems for Ubiquitous Computing. *Computer* Vol. 34:8, pp 57-66, 2001.
43. M. Hiltunen, R. Schlichting: Adaptive Distributed and Fault-Tolerant Systems, *International Journal of Computer Systems Science and Engineering*, 11(5):125-133, September, 1996.
44. E. Horvitz, J. Breese, D. Heckerman, D. Hovel, K. Rommelse. The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, pp 256-265, Madison, Wisconsin, 1998.
45. International Standards Organization (ISO) Extended Backus-Naur Form, *ISO/IEC 14977:1996(E)*, www.iso.org.
46. S. Intille: Designing a home of the future. *IEEE Pervasive Computing*, vol. April-June, pp. 76-82, 2002.
47. R. Jacob: A Specification Language for Direct Manipulation Interfaces. *ACM Transactions on Graphics*, Vol. 5:4, pp 283-317, ACM, 1986.
48. Jones, M., Rosu, D., Rosu, M.: CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Proc ACM Symp Operating Systems Principles (SOSP)*, 1997.
49. G. Judd, P. Steenkiste. Providing Contextual Information to Pervasive Computing Applications. *Proc. IEEE International Conference on Pervasive Computing (PERCOM)*, Dallas, 2003.
50. The K Desktop Environment. http://www.kde.org/
51. H. Kautz, L. Arnstein, G. Borriello, O. Etzioni, D. Fox: An Overview of the Assisted Cognition Project, AAAI-2002 Workshop on Automation as Caregiver: The Role of Intelligent Technology in Elder Care, http://www.cs.washington.edu/homes/kautz/papers/.
52. A. Kobsa: Generic User Modeling Systems. *User Modeling and User Adapted Interaction*, Ten Year Anniversary Issue, Kobsa (Ed.), 11 (1/2), pp. 49-63, Kluwer Academic Publishers, 2001.
53. F. Kon, et al.: Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. *Proc. USENIX Conference on OO Technologies and Systems (COOTS)*, 2001.
54. M. Kozuch, M. Satyanarayanan: Internet Suspend/Resume. Presented at the Fourth IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY. Available as *Intel Research Report IRP-TR-02-01*, Jun. 1, 2002.
55. E. de Lara, D. Wallach, W. Zwaenepoel: Puppeteer: Component-based Adaptation for Mobile Computing. *Proc. 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.
56. C. Lee et al.: A Scalable Solution to the Multi-Resource QoS Problem. *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 1999.
57. B. MacIntyre, E. Mynatt, S. Voida, K.Hansen, J. Tullio, G. Corso. Support For Multitasking and Background Awareness Using Interactive Peripheral Displays. *Proc. ACM User Interface Software and Technology (UIST'01)*, Orlando, Florida, November 2001.
58. The MacUpdate Desktop Manager. http://www.macupdate.com/info.php/id/12682
59. J. Magee, J. Kramer. *Concurrency, State Models & Java Programs*. J. Wiley & Sons, 1999.
60. B. Myers, S. Hudson, R. Paush: Past, Present and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction (TOCHI)* Vol. 7:1, pp 3-28, ACM Press, March 2000.
61. B. Myers, D. Smith, B. Horn: Report on the 'End-User Programming' Working Group. *Languages for Developing User Interfaces*. Jones and Barlet (Eds.), Boston, MA, 1992.
62. D. Narayanan, J. Flinn, M. Satyanarayanan: Using History to Improve Mobile Application Adaptation. *Proc. 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2000.
63. Neugebauer, R., McAuley, D.: Congestion Prices as Feedback Signals: An Approach to QoS Management. *Proc ACM SIGOPS European Workshop*, 2000.
64. B. Noble, et al.: Agile Application-Aware Adaptation for Mobility. Proc of the 16th ACM Symp on Operating Systems Principles (SOSP'97), October 1997. *Operating Systems Review 31(5)*, ACM Press, 276-287.
65. M. Paolucci, O. Shehory, K. Sycara, D. Kalp, A. Pannu. A Planning Component for RETSINA Agents. Lecture Notes in Artificial Intelligence, Intelligent Agents VI. M. Wooldridge and Y. Lesperance (Eds.) 1999.
66. PC Anywhere. http://www.symantec.com/pcanywhere/Consumer/index.html
67. T. Phan, K. Xu, R. Guy, R. Bagrodia. Handoff of application sessions across time and space. *Proceedings of the IEEE International Conference on Communications* (ICC 2001), June 2001.

68. R. Picard, Affective Computing. MIT Press, Cambridge, Massachusetts. 1997.

69. D. Pisinger: An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114, (1999).

70. V. Poladian, J.P. Sousa, D. Garlan, M. Shaw: Dynamic Configuration of Resource-Aware Services. *Proceedings of the 26th International Conference on Software Engineering - ICSE 2004*, IEEE Computer Society, pp. 604-613, Edinburgh, UK, May 2004.

71. S. Ponnekanti, B. Lee, A. Fox, P. Hanranhan. ICrafter: A Service Framework for Ubiquitous Computing Environments. UbiComp 2001: Ubiquitous Computing, Proceedings of the 3rd International Conference, Abowd, Brumitt and Shafer (Eds.), LNCS 2201, pp 56-75, Atlanta, Georgia, September 2001.

72. The Radar Project at Carnegie Mellon University. http://www.radar.cs.cmu.edu/external.asp

73. R. Rajkumar, C. Lee, J. Lehoczky, D. Siewiorek. Practical Solutions for QoS-Based Resource Allocations. Proceedings of the 19 th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998.

74. T. Richardson, F. Bennett, G. Mapp, A. Hopper. Teleporting in an X Window System Environment. *IEEE Personal Communications Magazine*, 1(3), pp 6-12, 1994.

75. M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, K. Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. IEEE Pervasive Computing, pp 74-83, Oct-Dec 2002.

76. M. Satyanarayanan. Mobile Information Access. IEEE Personal Communications, Vol. 3, No. 1, February 1996.

77. M. Satyanarayanan. Pervasive Computing: Vision and Challenges. IEEE Personal Communications, pp 10-17, Aug 2001.

78. A. Schmidt et al. Context Acquisition based on Load Sensing. UbiComp 2002: Ubiquitous Computing, Proceedings of the 4th International Conference, Borriello and Holmquist (Eds.), LNCS 2498, pp 333-350, Göteborg, Sweden, September 2002.

79. Service Location Protocol. http://www.ietf.org/html.charters/svrloc-charter.html and also http://www.openslp.org

80. S. Shearin, H. Lieberman. Intelligent Profiling by Example. Proc. International Conference on Intelligent User Interfaces (IUI 2001). Sante Fe, New Mexico, January 2001.

81. D. Siewiorek et al. Adtranz: A Mobile Computing System for Maintenance and Collaboration. *Proceedings of the 2nd IEEE International Symposium on Wearable Computers*, pp 25, IEEE Computer Society, 1998.

82. R. Simmons, D. Apfelbaum. A Task Description Language for Robot Control. Proceedings Conference on Intelligent Robotics and Systems, Vancouver Canada, October 1998.

83. The Smart Medical Home at the University of Rochester. http://www.futurehealth.rochester.edu/smart_home

84. Smith, G., Baudisch, P. et al. GroupBar: The TaskBar evolved. Proceedings of OZCHI'03, Brisbane, Australia, 2003.

85. J.P. Sousa, D. Garlan: The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. *Carnegie Mellon Univ. Technical Report CMU-CS-03-183*, 2003.

86. Spitznagel, B., Garlan, D. A Compositional Formalization of Connector Wrappers. *Proceedings of the 2003 International Conference on Software Engineering (ICSE'03)*, Portland, Oregon, USA, May 3 - 10, 2003.

87. J. Spivey: *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, Prentice-Hall, 1992.

88. K. Sycara, M. Paolucci, M. van Velsen, J. Giampapa. The RETSINA MAS Infrastructure. Joint issue of *Autonomous Agents* and *MAS*, Vol. 7, Nos. 1 and 2, July, 2003.

89. P. Tandler. Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices. UbiComp 2001: Ubiquitous Computing, Proceedings of the 3rd International Conference, Abowd, Brumitt and Shafer (Eds.), LNCS 2201, pp 96-115, Atlanta, Georgia, September 2001.

90. L. Terveen, L. Murray. Helping users program their personal agents. *Proc. of the 1996 Conference on Human Factors in Computing Systems, CHI'96*. Vancouver, Canada, April 1996.

91. J. Trevor, D. Hilbert, B. Schilit: Issues in Personalizing Shared Ubiquitous Devices. *Proc. of the 4th International Conference Ubiquitous Computing (UbiComp 2002)*, Borriello and Holmquist (Eds.), LNCS 2498, pp 56-72, Göteborg, Sweden, September 2002.

92. The Universal Description, Discovery and Integration protocol (UDDI). http://www.uddi.org/

93. The Vim text editor. http://www.vim.org/.

94. Z. Wang, D. Garlan. Task Driven Computing. *Carnegie Mellon University Technical Report CMU-CS-00-154*, http://reports-archive.adm.cs.cmu.edu/cs2000.html, May 2000.

95. R. Want, A. Hopper, V. Falcão, J. Gibbons. The Active-Badge Location System. *ACM Transactions on Information Systems*, Vol. 10:1, pp 91-102, ACM, 1992.

96. A. Wasserman, D. Shewmake: Rapid Prototyping of Interactive Information Systems. *ACM Software Engineering Notes*, Vol. 7:5, pp 171-180, ACM, 1982.

97. The Web Services Description Language (WSDL). http://www.w3.org/TR/wsdl

98. D. Weld, Recent Advances in AI Planning. AI Magazine, 20(2), pp 93–123, 1999.

99. J. Whittaker, M. Thomason. Markov Analysis of Software Specifications. ACM Transactions on Software Engineering and Methodology. 2(1) pp 93-106, 1993.

100. A. Wise, “Little-JIL 1.0 Language Report,” Department of Computer Science, University of Massachusetts at Amherst, Technical Report 98-24, April 1998.

Appendix A

This appendix complements Chapter 4 by specifying the protocols of interaction between Prism and the EM, between Prism and Suppliers, and between the EM and Suppliers – and how those interactions affect the state kept by the EM. This specification plays an important role in making sure that they are deadlock free, and also in verifying liveness conditions, such as the ability to recover from faults. Additionally, these models proved a precious tool during the low-level design and implementation of the infrastructure.

Prism – EM Protocol

The interaction between Prism and the EM is structured around the notion of task session. A session is created for each task that Prism needs to evaluate the feasibility in the current environment. To evaluate the feasibility of a task, Prism issues *budget* requests for each alterative configuration of services. If and when the user decides to resume a task, Prism issues a *setup* request.

Prism initiates a session for task *t* whenever the user starts working on task *t*, and closes the session whenever the user interrupts or finishes working on that task. Figure 4.2 shows an event sequence diagram that illustrates a typical task session.[29] Prism starts a task session by sending the EM a `newTask` message. The EM reply, `createdTask`, includes an id for the task session that will be attached to the exchanged messages throughout the session. Note that many sessions between Prism and the EM may be active concurrently, for one or more users. The session is terminated by a `disband` message issued by Prism, to which the EM replies with a `taskGone` message.

[29] The interactions between the EM and Suppliers are elided for simplicity: see the section on the Protocol EM – Suppliers.

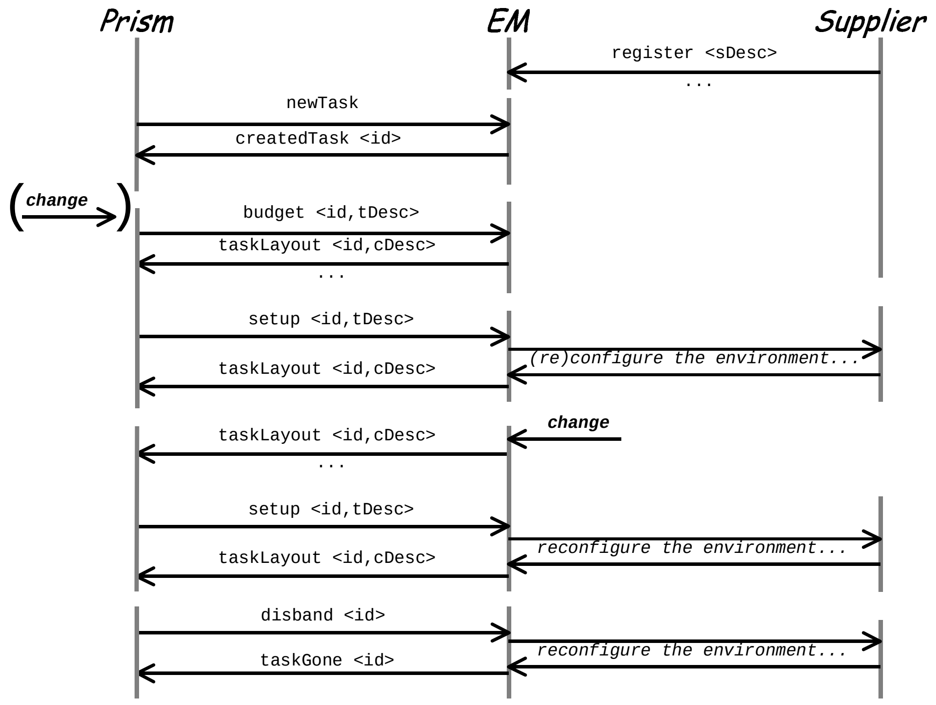


Figure A.1 Event sequence diagram for the communication between Prism and the EM.

Once a session is started, Prism obtains estimates for the utility that the environment can offer for the user's task by sending `budget` messages to the EM. Since there may be alternative configurations of services to support a given task, Prism will send one `budget` message for each of those candidate configurations. For example, for taking notes, either *text editing* or *speech recognition* services may be used. To each `budget` request, the EM replies with a `taskLayout`. For each service requested within a `budget`, the `taskLayout` indicates the Supplier that best matches the request among those currently available in the environment, as well as the Quality of Service (QoS) achievable with the current resources. The `taskLayout` also indicates the overall utility for the configuration. A budgeting exchange is also initiated by Prism whenever there are changes in the user's context or intent that justify a reevaluation of alternative configurations for the same task.

After the user requests a task to be resumed, Prism issues the corresponding `setup` message to the EM. Once the services are set up in the environment (see the section on the Protocol EM – Suppliers, below), the EM replies with a `taskLayout` containing the up-to-date description of the configuration supporting those services. Note that since there is a time lag between a `budget` and the `setup`, there may be some differences in what is achievable in the environment. Ideally, the contents of the two corresponding `taskLayout` messages will be the same, but Prism must be prepared to double check that, and either accept the differences, or if they are too significant, look for other alternatives and request a reconfiguration.

Of course, the capabilities of the environment may change, for better or for worse, during a task session. For example, a Supplier involved in the activated configuration may fail or become disconnected; a new Supplier that is a better match for a requested service may become

```
TaskSession = ( createdTask    -> MoreTaskReq ),
MoreTaskReq = ( {budget,setup} -> AnswerReq
              | taskLayout     -> MoreTaskReq     // EM's initiative
              | disband        -> DisbandSess ),
AnswerReq   = ( taskLayout     -> MoreTaskReq
              | noEMreply      -> RestartEM ),
DisbandSess = ( taskGone       -> TaskSession
              | noEMreply      -> RestartEM ),
RestartEM   = ( resetEM        -> MoreTaskReq ).

CreateSess  = ( newTask -> t[id:TId].createdTask -> CreateTask ).

||PrismEM   = ( CreateSess
             || forall [id:TId] t[id]:TaskSession )
             /  { noEMreply / {t[TId]}.noEMreply,
                  resetEM   / {t[TId]}.resetEM }.
```

Figure A.2 FSP specification for the connector Prism-EM.

available; or the resource conditions may change so much, that a different choice of Suppliers with distinct resource demands may be preferable. It is up to the EM to monitor the environment and promptly detect such situations. The EM will then reexamine the best match for the requested services, and it will either carry out the reconfiguration autonomously, or issue a `taskLayout` message to Prism containing a reconfiguration suggestion. Upon receiving a `taskLayout`, Prism may wish to study other alternatives to support the task, which it can do by issuing `budget` messages, but it will eventually settle on a reconfiguration and send the corresponding `setup` message.

Figure A.2 shows the FSP specification for the protocol of interaction between Prism and the EM. The permissible sequence of messages exchanged during a task session is specified by the `TaskSession` process.[30] After a `createdTask`, the protocol accepts either a `budget` or `setup`, leading to the `AnswerReq` process, a `taskLayout` initiated by the EM, or a `disband`, leading to the `DisbandSess` process. In the case Prism initiates a `budget` or `setup`, the EM is expected to reply with a `taskLayout` message. In the case Prism initiates a `disband` request, the EM is expected to reply with a `taskGone` confirmation. In both these cases, if the EM fails to reply, the protocol will engage in the `noEMreply` event, leading to the `RestartEM` process. Of course, the `noEMreply` event does not correspond to a real message, but rather to a timeout within Prism leading to a state change in the protocol of interaction. Similarly, the `resetEM` event does not correspond to a message, but to Prism activating a mechanism for rebooting the EM. Notice that after a `taskGone` message, the protocol goes back to the initial state awaiting the start of a new session with the `createdTask` message.[31] The `CreateSess` process states that after a `newTask`

---

[30] Note that in FSP, the originator of each message (or *event*, in process-algebra terms) is unspecified, so a trace permitted by this specification should be understood as a possible sequence of messages observed in the channel between Prism and the EM, with the direction of communication abstracted away.

[31] In FSP, processes are not dynamically created and terminated, but rather transition back and forth from an active state to an inactive state, where all they can accept is the event corresponding to the "creation."

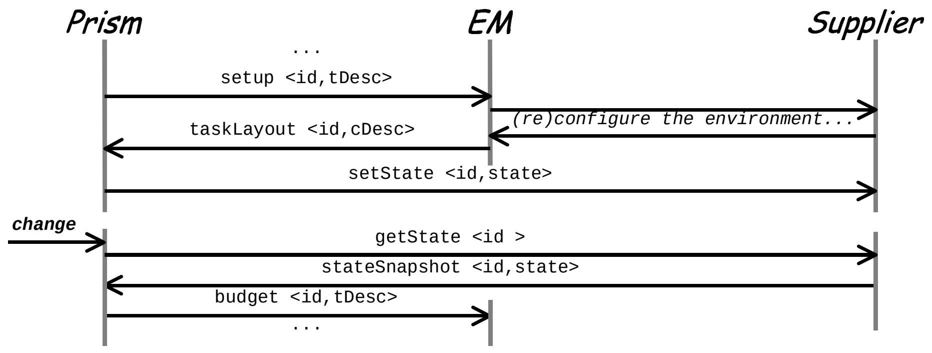


Figure A.3 Event sequence diagram for the communication between Prism and the Suppliers.

message is initiated by Prism, the EM is expected to reply with a `createdTask` message indicating the id for the session.

The Prism-EM protocol is given by the parallel composition of the process for creating new sessions, `CreateSess`, and of some arbitrary number of processes of type `TaskSession`. In the FSP specification, this arbitrary number of processes of the same type is achieved by prefixing the process (and consequently the events within that process) by a label (`t`) and a number (`id`, in the arbitrary range `TId`). The process `CreateSess` and each specific `t[id].TaskSession` process interact by sharing the event `t[id].createdTask` – this models a new task session being created and named. Furthermore, all `t[id].TaskSession` processes share the `noEMreply` and `resetEM` events making sure all task sessions agree on when the EM needs to be restarted. This synchronization in the process model is achieved by relabeling all `t[id].noEMreply` events to a single event `noEMreply`, and the same for `resetEM`.

Prism – Suppliers Protocol

Figure A.3 shows an event sequence diagram that illustrates a typical interaction between Prism and a Supplier. After Prism receives confirmation from the EM that Suppliers have been activated to support the user's task, Prism reconstitutes the user-level state of the task by sending a `setState` message to each of the Suppliers. Examples of user-level state are which files the user is working on, as well as user-interaction parameters such as cursors, window size, etc. Prism recaptures the updated state from the Suppliers by sending a `getState` message to each, and receiving back a `stateSnapshot`. Recapturing the state of the Suppliers is done whenever there are changes in the user's context or intent that hint that the task is about to be suspended. It may also be done periodically, to ensure recovery of an almost up-to-date state in the case a Supplier fails.

```
||PrismSupplier = forall [id:TId] t[id]:SetGetState.
SetGetState  = ( setState -> SetGetState
               | getState -> ( stateSnapshot -> SetGetState
                             | noSuppReply   -> SetGetState )).
```

Figure A.4 FSP specification for the connector Prism-Suppliers.

```
||Prism      = ( InvokeTask
               || forall [id:TId] t[id]:Task
               || forall [id:TId] t[id]:UseSupp )
               /  { noEMreply / {t[TId]}.noEMreply,
                    resetEM   / {t[TId]}.resetEM }.

InvokeTask  = ( newTask -> t[id:TId].createdTask -> InvokeTask ).

Task        = ( createdTask    -> CreatedTask ),
CreatedTask = ( {budget,setup} -> GetLayout
              | taskLayout     -> CreatedTask     // EM's initiative
              | disband        -> DisbandTask ),
GetLayout   = ( taskLayout     -> CreatedTask
              | noEMreply      -> RestartEM ),
DisbandTask = ( taskGone       -> Task
              | noEMreply      -> RestartEM ),
RestartEM   = ( resetEM        -> CreatedTask ).

UseSupp     = ( setup -> SetState
              | {taskLayout,noEMreply,disband}  -> UseSupp ),
SetState    = ( taskLayout -> setState -> SetGetState
              | noEMreply  -> UseSupp ),
SetGetState = ( {setState,getState,noEMreply} -> SetGetState
```

Figure A.5 FSP specification for the behavior of Prism.

Figure A.4 shows the FSP specification for the protocol of interaction between Prism and the Suppliers. For each task session, the protocol admits any sequence of `setState` and `getState` messages, with the proviso that a `stateSnapshot` reply is expected after each `getState`. Similarly to the `noEMreply` event in the Prism-EM connector, `noSuppReply` corresponds to a timeout within Prism rather than to an exchanged message. In this case, however, Prism will not take any action to recover/restart the Supplier. Prism relies on the EM to diagnose and propose the replacement of faulty Suppliers. Therefore, Prism will wait for some indication from the EM, or otherwise try to get the state again. The following section addresses combining these two protocols within Prism.

Prism

Figure A.5 shows the FSP specification for the behavior of Prism. That behavior is the parallel composition of the process for creating new sessions, `InvokeTask`, of an arbitrary number of task session interactions with the EM, `Task`, and of the same number of interactions with Suppliers, `UseSupp`. The processes `InvokeTask` and `Task` state Prism's view of the processes `CreateSess` and `TaskSession`, respectively, in the Prism-EM protocols (they are restated here for completeness of the specification of Prism's behavior). Notice also that, as before, the events `noEMreply` and `resetEM` are shared among the processes for all task sessions.

The glue between the two protocols Prism-EM and Prism-Supplier is specified by the `UseSupp` process. For simplicity, the messages exchanged with all the Suppliers supporting one user's task are modeled as single events. For instance, the `setState` event corresponds to sending `setState` messages to all such Suppliers. The central rule governing this protocol is that after

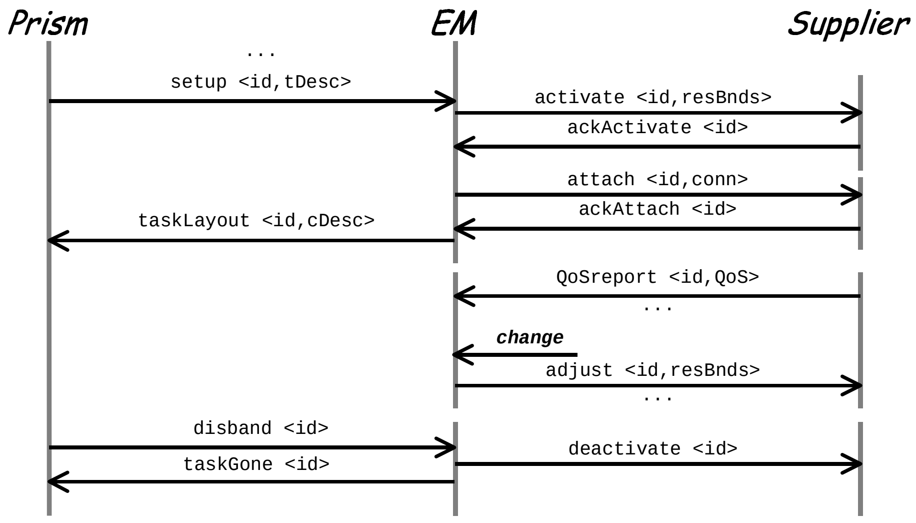


Figure A.6 Event sequence diagram for the communication between the EM and the Suppliers.

receiving a `taskLayout` in response to a `setup`, the state of the Suppliers must be set. Therefore, initially, the `UseSupp` process looks for `setup` events and is permissive to other messages – for instance, `taskLayout` may occur is response to a budget. After the `setup` event, the `SetState` process looks for the corresponding `taskLayout`, after which it issues a `setSate`, or in case the EM fails to respond, it resets to `UseSupp`. After the initial `setState`, the `SetGetState` process allows Prism to issue any number of `setState` and `getState` messages. However, if a `taskLayout` is received at this stage, it will correspond to the EM's initiative to substitute a Supplier. In such a case, Prism must decide whether to keep the supplier or to request a replacement by issuing a `setup` (in `UseSupp`). Notice that both `keepSuplier` and `replaceSupplier` are local events just for the purpose of modeling Prism's decision. Additionally, as a consequence of a change in user intent, Prism may decide to request a change in the configuration by issuing a `setup` and waiting for the corresponding `taskLayout` (in `SetState`). Naturally, disbanding the task session resets the process to its initial state.

Protocol EM – Suppliers

Figure A.6 shows an event sequence diagram that illustrates a typical interaction between the EM and a Supplier. After the EM receives a `setup` request from Prism, it activates the Supplier, indicating the bounds on resource consumption, and it attaches the Supplier's ports as requested in the `setup` message. Both the `activate` and `attach` messages are acknowledged by the Supplier upon successful completion. After the supplier is activated, it issues periodic QoS reports to the EM. If the resources in the environment change significantly, the EM may establish new resource bounds for the Supplier by sending it an `adjust` message with the new bounds. Eventually, the EM will receive a disband message from Prism and proceeds to deactivate the Supplier. Notice that the EM makes sure the Supplier is up and properly attached before returning a `taskLayout` to Prism. Subsequent adjustments to resource bounds and

```
||EMSupplier = forall [id:TId] t[id]:ManageSupp.

ManageSupp   = ( activate        -> ActivateSupp
               | deactivate      -> ManageSupp ),
ActivateSupp = ( ackActivate     -> AttachSupp
               | noSupplierReply -> ManageSupp ),
AttachSupp   = ( attach   -> ( ackAttach       -> MonitorSupp
                             | noSupplierReply -> ManageSupp )
               | noAttach                      -> MonitorSupp ),
MonitorSupp  = ( pQoSreport -> MonitorSupp
               | adjust     -> MonitorSupp
               | activate   -> ActivateSupp      // other Services
               | deactivate -> ManageSupp )
              \ {noAttach}.
```

Figure A.7 FSP specification for the connector EM-Suppliers.

deactivation are not subject to the same constraint, and therefore, acknowledgments are not required.

Figure A.7 shows the FSP specification for the protocol of interaction between the EM and the Suppliers. For each task session, the protocol is given by the process `ManageSupp`. Again, for simplicity, a single event models the interaction with all the Suppliers involved in a task session. For instance, the event `activate` models sending messages to all the suppliers to be activated for the task session. To activate a Supplier, the pair `activate`, followed by `ackActivate`, must be observed. If the Supplier needs to be attached, the pair `attach`, followed by `ackAttach`, will also be observed. Otherwise, attachment is skipped – in FSP this is modeled by the hidden event `noAttach`. Notice that deactivating a Supplier is accomplished by a single message exchange, `deactivate`. Notice also that the timeout event `noSupplierReply`, in case the Supplier fails to acknowledge an `activate` or `attach`, resets the protocol – Section 0 explains how this timeout is handled within the EM. During monitoring, a Supplier issues periodic QoS reports, (represented by `pQoSreport`, since FSP events cannot start with a capital letter) and may receive an arbitrary number of adjustments to its resource bounds, `adjust`. Note that the protocol allows `activate` events during the monitoring phase. There are two reasons for this. The first is that, as a consequence of later `setup` requests, the *same* Supplier may receive additional activations for other services. The second reason is a feature of the simplification explained above, where the communication with all Suppliers for the task session is modeled as a single process: again as a consequence of later setup requests, *other* Suppliers may need to be activated. Naturally, deactivation resets the protocol.

EM

The EM plays a central role in intermediating between the user's needs, for which Prism acts as a proxy, and the applications and devices in the environment. As such, the EM keeps models of both the capabilities of the environment, and of the user's needs, as transmitted by Prism. In addition to the FSP model of the EM's behavior, this section shows a Z model of the state kept by the EM as a result of the interactions with both Prism and the Suppliers. This state model is only as detailed as necessary to clarify the effects of such interactions.

```
||EM = ( CreateTaskModel
      || forall [id:TId] t[id]:UpdateTaskModel
      || forall [id:TId] t[id]:ManageEnv )
      /  { resetEM / {t[TId]}.resetEM }.

CreateTaskModel  = ( newTask -> t[id:TId].createdTask -> CreateTaskModel).

UpdateTaskModel  = ( createdTask       -> CreatedTaskModel ),
CreatedTaskModel = ( {budget,setup}    -> IssueLayout
                   | missQoSReports    -> IssueLayout      // EM
initiative
                   | resetEM           -> CreatedTaskModel
                   | disband           -> RemoveTaskModel ),
IssueLayout      = ( taskLayout        -> CreatedTaskModel
                   | resetEM           -> CreatedTaskModel ),
RemoveTaskModel  = ( taskGone          -> UpdateTaskModel
                   | resetEM           -> CreatedTaskModel ).

ManageEnv    = ( setup      -> ActivateSupp
               | pQoSreport -> deactivate -> ManageEnv
               | resetEM    -> ManageEnv ),
ActivateSupp = ( activate   -> ( ackActivate     -> AttachSupp
                               | noSupplierReply -> IssueLayout )
               | deactivate -> IssueLayout
               | resetEM    -> ManageEnv ),
AttachSupp   = ( attach     -> ( ackAttach       -> IssueLayout
                               | noSupplierReply -> IssueLayout )
               | skipAttach -> IssueLayout
               | resetEM    -> ManageEnv ),
IssueLayout  = ( taskLayout -> MonitorSupp
               | resetEM    -> ManageEnv ),
MonitorSupp  = ( adjust         -> MonitorSupp
               | pQoSreport     -> MonitorSupp
               | missQoSReports -> IssueLayout   // report to Prism
               | resetEM        -> MonitorSupp
               | setup          -> ActivateSupp
               | disband -> deactivate -> {taskGone,resetEM} -> ManageEnv)
              \ {skipAttach}.
```

Figure A.8 FSP specification for behavior of the EM.

Figure A.8 shows the FSP specification for the behavior of the EM. That behavior is the parallel composition of the process for creating task models, `CreateTaskModel`, of an arbitrary number of processes to update as many task models, `UpdateTaskModel`, and of the same number of processes to manage the corresponding configuration of Suppliers in the environment, `ManageEnv`. The processes `CreateTaskModel` and `UpdateTaskModel` state the EM's view of the processes `CreateSess` and `TaskSession`, respectively, in the Prism-EM protocols (they are restated here for completeness of the specification of EM's behavior). Notice that the `noEMreply` event is not seen by the EM since it corresponds to a timeout within Prism. Notice also that the observation of the event `missQoSreports` prompts the EM to issue a `taskLayout` with a reconfiguration suggestion to Prism (more on this below).

The glue between the two protocols Prism-EM and EM-Supplier is specified by the `ManageEnv` process. As before, for simplicity, the messages exchanged with all the Suppliers supporting one task are modeled as single events. For instance, the `activate` event corresponds to sending `activate` messages to all such Suppliers. The implementation of the EM can use standard concurrency mechanisms, such as barriers, to wait for the reception of all the relevant acknowledge messages, and only then, from the protocol specification point of view, consider that it observed the corresponding `ackActivate` event. The activation and attachment of Suppliers is triggered upon receiving a `setup` request. In the case all the acknowledgements are received, that is, upon the successful activation and attachment of the requested Suppliers, the EM issues a `taskLayout` with the complete configuration. In the case some of the acknowledgements timeout, the event `nosupplierReply` is observed, and a `taskLayout` is still issued but this time with a possibly incomplete configuration. In this latter case, the EM may try to find and activate alternative, possibly less optimal Suppliers before returning the `taskLayout`. Notice that a `setup` may request some of the Suppliers in the current configuration to be deactivated (see `ActivateSupp` process). No acknowledgment is needed for deactivation before issuing the updated `taskLayout`.

After the first `setup` for a task session, the EM monitors the configuration according to the `MonitorSupp` process. Active Suppliers in a configuration periodically issue QoS reports to the EM. The EM uses these reports to evaluate the utility of the current set of suppliers against possible alternatives in the environment and may come up with an advantageous reconfiguration. Furthermore, when the EM notices that a particular Supplier fails to issue QoS reports, it will try to replace that (presumably) faulty Supplier: in the FSP model this is represented by the event

[*Id*, *Description*, *UtilityValue*, *Supplier*]

EMTaskModel

serviceDesc: Id ⇸ *Description*
knownServices: ℙ *Id*
suppPrefs: ℙ *Supplier* ⟶ *UtilityValue*

dom *serviceDesc* = *knownServices*

EMEnvModel

knownSuppliers: ℙ *Supplier*
supplierMapping: Id ⇸ *Supplier*
activeServices: ℙ *Id*

dom *supplierMapping* = *activeServices*
ran *supplierMapping* ⊆ *knownSuppliers*

EM

EMTaskModel
EMEnvModel
bestChoice: Description × ℙ *Supplier* ⟶ *Supplier*

activeServices ⊆ *knownServices*

Figure A.9 Z model of the state kept by the EM as a result of the Prism-EM communication.

`missQoSreports`, leading to the `IssueLayout` process. In either case, and according to the autonomy policies for swapping Suppliers, the EM may have to confirm with Prism that the reconfiguration is appropriate/opportune before carrying out. Nonetheless, the EM has full autonomy to `adjust` the resource bounds on the Suppliers.

After each (re)configuration of the environment, made in response to a `setup` request, the EM updates a persistent checkpoint of its models. In case the EM implementation fails, those persistent checkpoints enable restarting the EM without having to reconfigure the environment from scratch. That is, the Suppliers can continue to support the user's task, while Prism, upon detecting the EM's lack of response will restart the EM and reissue any pending `setup` request. The handling of the `resetEM` event in the `ManageEnv` process captures the fact that an environment reconfiguration is transactional in the following sense: if the EM fails anywhere between a `setup` and the corresponding `taskLayout`, no intermediate state is recovered. In such a case, any Suppliers that were activated by the incomplete reconfiguration will be detected and deactivated by the EM: the EM will react to QoS reports from Suppliers it does not recognise as being active by sending them a `deactivate` message.

Figure A.9 shows the Z model of the state kept by the EM as a result of the interactions with Prism and the Suppliers. For each task session, the EM keeps both a model of the task as communicated by Prism, *EMTaskModel*, and of the environment that supports that task, *EMEnvModel*. The task model consists of two pieces: (1) a table of service descriptions indexed by service id, *serviceDesc*; and (2) the user preferences with respect to the choice of Suppliers for each service, *suppPrefs*. The model of the environment consists of two pieces: (1) the *supplierMapping*, which maps the id of each active service to the Supplier providing that service; and (2) the *knowSuppliers* set, which includes all the Suppliers that register with the EM, and is shared among all task sessions. For the sake of simplicity, the schema for the EM represents a single task model and environment model. Notice that the set of active services (the ones being currently provided by a Supplier) is a subset of the known services (the ones with a description transmitted by Prism). This is because Prism may explore a number of alternatives before settling on a set of services to support the user's task. Notice also that the *bestChoice* function corresponds to the algorithms within the EM that, given a service description, select the best fit among a given set of Suppliers. Figure A.10 shows the effect of a `register` message sent by a Supplier: only the set of known Suppliers in the *EMEnvModel* is updated with the new supplier.

register

ΔEMEnvModel
newSupplier?: Supplier

---

knownSuppliers' = knownSuppliers ∪ {newSupplier?}
supplierMapping' = supplierMapping
activeServices' = activeServices

Figure A.10 Z model of the effect of the `register` message on the state kept by the EM.

```
budget
    $\Delta EMTaskModel$
    $\Xi EMEnvModel$
    $EM$
    $addServices?: \mathbb{P}\ Id$
    $replaceServices?: \mathbb{P}\ Id$
    $disbandServices?: \mathbb{P}\ Id$
    $newServDescs?: Id \twoheadrightarrow Description$
    $newSuppPrefs?: \mathbb{P}\ Supplier \twoheadrightarrow UtilityValue$
    $utility!: UtilityValue$
    ----------------
    $disj\ \{addServices?, replaceServices?, disbandServices?\}$
    $addServices? \cup replaceServices? \cup disbandServices? \subseteq knownServices'$
    $knownServices' = knownServices \cup \mathrm{dom}\ newServDescs?$
    $serviceDesc' = serviceDesc \oplus newServDescs?$
    $suppPrefs' = suppPrefs \oplus newSuppPrefs?$
    let $candidateServices == activeServices \setminus disbandServices? \cup addServices?$
      • let $candidateSuppliers == knownSuppliers \setminus \{ s: replaceServices? \bullet supplierMapping\ s \}$
        • let $candidateConfig == \{ s: candidateServices \bullet bestChoice\ ((serviceDesc\ s), candidateSuppliers) \}$
          • $utility! = suppPrefs'\ candidateConfig$
```

Figure A.11 Z model of the effect of the budget message on the state kept by the EM.

Figure A.11 shows the effect of a budget message sent by Prism. The purpose of this type of message is to run a "what if?" scenario against the current conditions in the environment. As such, a budget indicates the ids for the services to be hypothetically activated, deactivated (disbanded), or have the current Supplier replaced. Additionally, a budget piggybacks information for updating the task model: the relevant service descriptions, *newServDescs*, and an update on the user preferences with respect to Supplier choices, *newSuppPrefs*. Note that in the schema, the task model is affected, but the environment model is only observed. Consequently, the EM computes temporary values for the candidate services to be activated; the candidate Suppliers to choose from (all the known Suppliers, except for the ones that the user is unhappy with – the ones to be replaced); and the candidate configuration (the best choice of Suppliers for the candidate services). The utility value for the candidate configuration will be returned by the taskLayout message in reply to the budget.

Figure A.12 shows the effect of a setup message sent by Prism. The purpose of this type of message is to set up or change the configuration of Suppliers currently supporting the user's task. Like a budget, a setup piggybacks information for updating the task model. However, a setup indicates the ids for the services to be effectively added or removed from the configuration. Consequently, the EM updates both the task and environment models (and of course, sends the appropriate messages to the affected Suppliers, as described in the protocol specification). The environment model is updated in the following way: (1) the set of active services is cleared of the disbanded service ids, and appended with the newly activated ones; (2) the supplier mapping is cleared of the mappings for the disbanded or replaced services, and

$$
\begin{array}{l}
\textit{setup} \\
\hline
\Delta EMTaskModel \\
\Delta EMEnvModel \\
EM \\
addServices?: \mathbb{P}\ Id \\
replaceServices?: \mathbb{P}\ Id \\
disbandServices?: \mathbb{P}\ Id \\
newServDescs?: Id \rightarrowtail Description \\
newSuppPrefs?: \mathbb{P}\ Supplier \rightarrowtail UtilityValue \\
\hline
disj\ \{addServices?, replaceServices?, disbandServices?\} \\
addServices? \cup replaceServices? \cup disbandServices? \subseteq knownServices' \\
knownServices' = knownServices \cup \mathrm{dom}\ newServDescs? \\
serviceDesc' = serviceDesc \oplus newServDescs? \\
suppPrefs' = suppPrefs \oplus newSuppPrefs? \\
activeServices' = activeServices \setminus disbandServices? \cup addServices? \\
knownSuppliers' = knownSuppliers \\
\mathbf{let}\ candidateSuppliers == knownSuppliers \setminus \{\ s: replaceServices? \bullet supplierMapping\ s\ \} \\
\quad \bullet\ supplierMapping' = supplierMapping \\
\qquad\qquad\qquad \setminus \{\ d: disbandServices? \cup replaceServices? \bullet (d \mapsto supplierMapping\ d)\ \} \\
\qquad\qquad\qquad \cup \{\ s: addServices? \cup replaceServices? \\
\qquad\qquad\qquad\qquad \bullet\ (s \mapsto bestChoice\ ((serviceDesc\ s), candidateSuppliers))\ \} \\
\hline
\end{array}
$$

Figure A.12 Z model of the effect of the `setup` message on the state kept by the EM.

added with the best choices for the services to be added, or to have their suppliers replaced, among the candidate Suppliers. As before, the *candidateSuppliers* are all the known Suppliers, except for the ones that the user is unhappy with.