International Workshop on Adaptive and Self-Managing Enterprise Applications, pp 83-96
17th Conference on Advanced Information Systems Engineering, Porto, Portugal, 2005

Capitalizing on Awareness of User Tasks for Guiding Self-Adaptation

João Pedro Sousa, Vahe Poladian, David Garlan, Bradley Schmerl

School of Computer Science
Carnegie Mellon University, Pittsburgh, PA
{jpsousa | poladian | garlan | schmerl}@cs.cmu.edu

Abstract. Computers support more and more tasks in the personal and professional activities of users. Such user tasks increasingly span large periods of time and many locations across the enterprise space and beyond. Recently there has been a growing interest in developing applications that can cope with the specific environmental conditions at each location, and adapt to dynamic changes in system resources. However, in a given situation there may be many possible configuration solutions, and an awareness of the user's intent for each task is a critical element in knowing which one to pick. In this paper, we discuss the limitations of building such awareness into applications, and propose to factor the awareness of user tasks into a common software layer. That however, brings up the problem of coordinating the system-wide adaptation performed by such a layer with fine-grain adaptation performed by resource-aware applications. We summarize the main features of an architectural framework that incorporates such a layer, and distill some of the lessons learned in implementing the framework.

Introduction

What was once the concern of specialized systems with high availability requirements (e.g., space systems, telecommunications), is now recognized as being relevant to almost all of today's complex systems, and particularly to those where environmental resources can change radically (e.g., mobile computing) or where user needs can change dynamically, as dictated by their tasks (e.g., different tasks in the context of an office).

In addition to their role in the applications that directly support the business, self-adaptive systems play an increasingly important role in supporting the daily professional and personal activities of employees. In today's companies, employees use computing support to simultaneously handle many *tasks*, such as preparing presentations, writing reports, or answering email, constantly shifting their attention between one and another. Also, computer users are increasingly mobile: a user may start working on a presentation while in his or her office, continue at the office of a collaborator, and pick the task up later at home.

Unfortunately, current computer systems offer little support for user mobility and multiple tasks, and users carry the burden of configuring the computing environment 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.

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.

For the past five years we have developed a task management infrastructure that automatically configures computing environments for supporting the kinds of user tasks commonly found in offices. This infrastructure is an adaptive system in the sense that it continuously monitors the environment for both failures and opportunities for improvement, automatically performing reconfigurations as deemed appropriate.

Two key questions that we sought to answer in our research are: how should we model the needs and preferences of users for their tasks? And, where should we locate the knowledge about user tasks: inside applications, or somewhere else?

In our experience there are significant advantages in factoring knowledge about user tasks into a common software layer. However, fine-grain adaptation to resource changes is best handled within adaptive applications. This brings up the problem of coordinating the system-wide adaptation performed at the new software layer with fine-grain adaptation performed by resource-aware applications.

In this paper we describe an architectural framework for adaptive task management infrastructures. This architectural framework was honed by our experience in building one such infrastructure. The research described herein is part of Project Aura, a wider research initiative on pervasive computing [9].

In the remainder of this paper, Section 0 elaborates on the requirements addressed by our research. Section 0 discusses the pros and cons of locating awareness of user tasks within applications, while Section 0 summarizes the software layers in the proposed architectural framework, and the corresponding roles and interactions. For the sake of space, the formal specifications of the architecture and of the models of user tasks are left out of this paper (see [23] for details). Section 0 discusses the coordination of the adaptations enacted at three levels: changes in user tasks, changes in the supply of services in the environment, and changes in resources (such as battery and bandwidth). Section 0 compares our work to related research, and Section 0 summarizes the main points of this paper and lessons learned from our work.

Requirements

In rich computing environments with ever-changing resources, we believe that any system that targets supporting the daily professional or personal activities of users must account for three important aspects:

– *User tasks*. Users simultaneously handle many *tasks*, such as preparing presentations, writing reports, or answering email, constantly shifting their attention between one and another.

Typically, these tasks do not just involve one computing application, but a set of applications and information resources. Today, the user must think in terms of individual applications and then manually start and tailor those applications for the environment that they are currently in. To support self-adaptation, it is crucial that the computing system represent users' tasks, and can map those tasks to the applications and resources currently available in the environment, and to be able to adapt those tasks when the environment changes.

The notion of task also provides a locus of information about the properties that are important to the user for achieving the task, which play a crucial role when adapting the task to particular environments. For instance, for taking notes during a long meeting, a user may prefer to run his PDA in reduced performance mode so that the battery lasts for the whole meeting. However, if the user needs to email a large file before boarding a plane, he may prefer draining the battery by using the full performance mode to risk sending only part of the file.

– *User Mobility*. People are mobile in their daily activities: for example, a user may start preparing a business presentation while in his or her office, continue at the office of a collaborator, and pick the task up later at home. 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.

  Ideally, users 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 in their surroundings.

– *Environment change*. Every time users resume a task interrupted somewhere else, or sometime ago, the computing *environment* (meaning the set of devices, applications, and resources that are accessible to a user) at a particular location may be different from the last time that task was interrupted.

  Furthermore, the computing environment may change dynamically for ongoing tasks, as a consequence of variations of performance and availability of services over networks, or as a consequence of user mobility. For instance, a user may join a teleconference while walking down the hall, where only his wireless PDA is available, and then enter a smart room, where he may take advantage of a large display and wired connectivity.

We took on the three aspects above as requirements for an infrastructure for supporting task management. This infrastructure promotes user tasks to first class entities in computing environments, 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 his office and *resume* it in a meeting room, with whatever computing capabilities are locally available.

Furthermore, the infrastructure continuously monitors the environment for both failures and opportunities for improvement, automatically performing reconfigurations as deemed appropriate.

Table 1. Summary of the terminology used in this paper

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*.
service	Either (a) a service type, such as printing, or (b) the occurrence of a service proper, such as printing 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. a printer.
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 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.

This infrastructure is an adaptive system in the following sense: adaptive systems hold a model of the universe of discourse, 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 computing environment. The purpose of the infrastructure is to maximize the utility of the environment with respect to the needs of users for each task, by acting on (configuring) the environment.

For the sake of being precise, **Table 1** summarizes the terminology used throughout this paper.

Locating the Awareness of User Tasks

Currently, many applications incorporate some level of awareness of user tasks. Typically this is done by having each application learn and store some user-level state, such as preferences, the last few files worked on, window size, and active options.

Resource-adaptive applications take awareness of user tasks in another direction, by applying user-specific policies for guiding their adaptation to dynamically changing resources. For instance, an adaptive speech recognizer might make tradeoffs between the accuracy of the recognition and the latency constraints expressed by the user, based on the available CPU cycles.

Incorporating awareness of user tasks directly into each application has the benefit that the knowledge about the user's task can be fine tuned to the features of each application. However, it also has some serious limitations:

- *Software engineering costs*. Currently, task-awareness features are added to applications with little concern for generality, and often by intertwining those features with application code. This stovepiped approach makes it very hard to reuse solutions across different applications.

- *Awareness of user tasks*. In everyday computing, the *same* application may be used to support different user tasks in turn. For instance, a text editor may be used to support writing a conference paper at one time, but writing a monthly report in another, each with its own files, options and window settings. Currently, applications store user-level state, at best, on a per-user basis (older applications store one user-level state, which all users share). Unfortunately, the user-level state that should be recovered can be different for each user *task*.

  Lack of knowledge about the user's task also affects an application's ability to adapt to varying resources. For instance, would the user of a language translator prefer accurate translations or snappy response times? Should an application running on a mobile device use power-save modes to preserve battery charge, or should it use resources liberally in order to complete the user's task before he runs off to board his plane? Today, existing approaches to resource adaptation place the heuristics to determine the adaptation policies within the adaptive application or within the operating system. Such approaches overlook the fact that an *appropriate* adaptation policy should be determined by the nature of the user's task – and that is very hard to infer at the application level.

- *Application vs. task optimization*. Supporting one user task often involves invoking several applications. For instance to write a conference paper, the user may need to edit the document, browse the web for related work, and skim a promotional video released by a competitor research group. If left to their own policies, the web browser and the video player may compete for bandwidth in a way that does not deliver the best Quality of Service (QoS) to the user. Depending on the user's intention, it may be preferable to speedup web browsing, while playing a lower quality video… or the other way around. In general, how resources should be allocated among applications follows from the user's priorities for his task, rather than from generic "fairness" policies adopted by operating systems and networking infrastructures, or from the local optimization policies adopted by applications.

- *Awareness of user mobility*. Suppose the user wants to resume writing his conference paper using his home computer, after he worked on that task earlier at the office. Most applications today offer little or no support for synchronizing the user-level state with applications on other devices. Those who do, interchange the information in a proprietary format, restricted to other instances of the same application. Unfortunately, homogeneity of platforms and applications was not attained in the more uniform world of desktop computing, let alone in the emerging reality of pervasive computing.

An alternative to incorporating awareness of user tasks into each application is to factor it out into a common software layer. In such a layer, user tasks are made explicit by modeling the user needs and preferences for each task. Once such information is represented, it can be used to guide the overall configuration of the computing environment.

Architectural Framework

Our architectural framework factors awareness of user tasks out of the applications and into a common software layer. We built a task management infrastructure according to this framework. The infrastructure exploits models of user tasks to perform automatic configuration and reconfiguration of environments according to the requirements of each user task.

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.

In our 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.

Table 2 summarizes the roles of these software layers and 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.

The Task Management layer (called Prism) plays the main role in adapting to changes in user tasks and preferences. Prism holds knowledge about user tasks and preferences which is used to coordinate the configuration of the environment upon

Table 2. Software layers of the proposed architectural framework

layer	*mission*	*subproblems*
Task Management (Prism)	***what*** does the user need	• monitor the user's task, context and intent • map the user's task to needs for services in the environment • 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 environment-specific capabilities • ongoing optimization of the utility of the environment relative to the user's task
Environment	support the user tasks	• monitor relevant resources • fine grain management of QoS/resource tradeoffs

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.[1] 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 (more on this below).

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. 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 new suppliers became accessible or more attractive in terms of forecast QoS. Whenever an alternative configuration becomes more attractive, the ME layer 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. Of course, if the supplier in question failed, that cost is unavoidable, and the ME layer proceeds to activate the best alternative supplier.

Finding the Best Match

The best match between user needs and preferences, and environment capabilities is determined using a utility-theoretic framework. Prism generates the alternatives for *what* a user may want, while the ME evaluates *how well* the environment can support each alternative. For each alternative configuration within each possible task, Prism generates a *budget* request to the ME. That request contains the model of the configuration and of the user preferences. The quantitative evaluation of each alternative is supported by the notion of *utility*.

Specifically, Prism captures user preferences relative to alternative service configurations, relative to alternative suppliers for each service, and relative to multiple dimensions of quality of service. Such preferences, and the supporting utility-theoretic framework, are used 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. For example, suppose

[1] Complementary research in context-awareness may be integrated with Prism (e.g., [6]), enabling Prism to react to events such as suspending a user's tasks when he leaves the office.

that a user starts working on a paper at home, using MS Word as a supplier for text editing, 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, Fred may prefer using Emacs to Vim (or vice-versa).

Formally, the supplier preferences for a service *s* are represented as a discrete mapping, h_{Supp}: $P_s \rightarrow U$, between the set of known suppliers for the service, P_s, and the utility space $U \triangleq [0,1]$. In the example, the user might signal that he clearly prefers Emacs over Vim by setting h_{Supp}(Emacs)=1 and h_{Supp}(Vim)=0.3. He might also signal that he would be 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 than to use Vim (the opposite might be represented by flipping these values).

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 the user's preferences for the current task. When watching a sports event, the user 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 example, while browsing an e-commerce site over a poor connection, the user may want to skip loading pictures in favor of faster response, but he may be willing to wait for the pictures to load once he reaches the page with the desired product.

Formally, the QoS preferences for a service *s* are given by a set of functions $h_{QoS\ d}$: $Dom(d) \rightarrow U$, that map the domain of each QoS dimension *d* to the utility space. For example, *Dom*(*response time*) is the set of positive real numbers, scaled in seconds, and $h_{QoS\ response\ time}$ indicates how happy the user is with each value of response time.

To define the overall utility of the environment for configuration *c*, 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 $s \in S_c$. 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 is given by:

Definition 1
$$U(c|p,q) \triangleq \prod_{s \in S_c} h_{Supp}^{w_s}(p(s)) \cdot \prod_{d \in D_c} h_{QoS\ d}^{w_d}(q(d))$$

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. The weights w_s and $w_d \in [0,1]$ reflect how much the user cares the choice of supplier for *s*, and about the quality along dimension *d*, respectively. (By assigning a low value to a *w*, the overall utility is desensitized to the corresponding choice/variations.)

To maximize the utility above, the ME 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 *c* and a forecast of the available resources in the environment, the ME determines the supplier assignment, $\hat{p}$, and the forecast levels of QoS, $\hat{q}$, that maximize:

Formula 2
$$\operatorname*{arg\,max}_{\substack{p:S_c \to P_c \\ q:D_c \to Q_c}} \quad U(c|p,q)$$

The algorithms involved in solving Formula 2 are discussed in [21], and other research addresses forecasting available resources (e.g., [17]).

Adaptation at Three Levels

The previous section discussed system-wide configuration and reconfiguration in our framework. Namely, at Task Management level, changes in user tasks cause Prism to either adjust the service composition of currently active tasks, or to activate or deactivate all the services involved in some task. For that, Prism interacts with the ME to evaluate how well alternative service configurations can be supported in the environment, and once a decision is reached, Prism requests the ME to carry out a specific reconfiguration in the environment. Reconfiguration at this level is triggered by human actions, or intentions, 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. The ME periodically carries out an evaluation of the alternative ways to support the configuration of services requested by Prism. If a better alternative is found for a currently active supplier, and depending on what was specified by Prism for the service, the ME may proactively swap the supplier, or it may coordinate with Prism on whether and when to swap it. This kind of evaluation takes place at a time-scale of a few seconds.

In addition to the two kinds of system-wide adaptation discussed above, another kind occurs at the Environment level. Here, resource-adaptive applications are able to change their internal behavior to make the most out 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 [1].

To support integrating such adaptive applications into our framework we need to answer questions like: should the identification and configuration of remote components be managed by the ME, 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 ME to find and configure distributed components. 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 ME's role with respect to resource allocation, such applications should expose a model of their QoS behavior to the ME (see [23] for details). Based on that model, the ME views, activates, and manages the corresponding supplier as a unit: all internal behavior is treated as a black box by the ME.

Furthermore, resource allocation and adaptation policies need to 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. Consequently, adaptive applications should be amenable to have their resource usage and adaptation policies determined externally and passed dynamically

Specifically, the Aura framework addresses this problem as follows. First, the ME calculates the optimal resource allocation among the suppliers, as part of the maximization in Formula 2. Second, Prism elicits the QoS preferences that drive the preferred QoS tradeoffs for the task (the set of functions h_{QoS} in Definition 1). These are passed to the suppliers upon activation of a service and whenever there are changes: for instance, if the user changes preferences in the middle of a task.

Related Work

Currently, adaptive systems fall into two broad categories: fault-tolerant systems, and resource-aware systems. First, fault-tolerant systems react to component failure, compensating for errors using a variety of techniques such as redundancy and graceful degradation [5,11]. Such systems have been prevalent in safety-critical systems or systems for which the cost of off-line repair is prohibitive (e.g., telecom, space systems, power control systems, etc.) Here the primary goal is to prevent or delay large-scale system failure.

Second, 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.) [7,14,17,19]. Many of these systems emerged with the advent of mobile computing over wireless networks, where resource variability becomes a critical concern. While most of this research focuses on one component at a time, our work leverages on this research but tackles the problem of multi-component integration, configuration, and reconfiguration. Although somewhat related, this kind of automatic configuration is distinct from the automatic configuration being investigated in other research [16]. 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.

Resource scheduling [13], resource allocation [15,18], and admission control have been extensively addressed in research. From analytical point of view, closest to our work are Q-RAM [15], a resource reservation and admission control system maximizing the utility of a multimedia server based on preferences of simultaneously connected clients; Knapsack algorithms [20]; and winner determination in combinatorial auctions. In our work, we handle the additional problems of selecting applications among alternatives, and accounting for cost of change. Dynamic resolution of resource allocation policy conflicts involving multiple mobile users is addressed in [2] using sealed bid auctions. While our work shares utility-theoretic concepts with [2], the problem solved in our work is different. In that work, the objective is to select among a handful of policies so as to maximize an objective function of multiple users. In our work, the objective is to choose among possibly thousands of configurations so as to maximize the objective function of one user. As such, our work has no game-theoretic aspects, but faces a harder computational problem. Furthermore, our work takes into account tasks that users wish to perform.

At a coarser grain, research in distributed systems addresses global adaptation: for example, a system might reconfigure a set of clients and servers to achieve optimal load balancing. Typically, such systems use global system models, such as architectural models, to achieve these results [4,8,10]. To achieve fault-tolerance and coarse-grain adaptation (e.g. hot component swapping,) our work builds on this, as well as on service location and discovery protocols [12,22].

Research in middleware for accessing the World Wide Web has proposed models of QoS for web services. Currently, such models of QoS adopt a generic (i.e., service-independent) view of QoS based on parameters such as price, time, availability and reliability [3,24]. In contrast, the models of QoS we adopt are service-specific. For instance, a video streaming service might be qualified by the image quality and frame rate of the stream, whereas a language translator might be qualified by the latency and accuracy of translation.

Conclusions

We presented an architectural framework for adaptive task management infrastructures. Task management infrastructures promote user tasks to first class entities in computing environments, and thus enable users to operate directly on their tasks, namely for suspending and resuming tasks. This framework was honed by our five-year experience in building such an infrastructure.

Specifically, the proposed framework (and infrastructure that implements it) targets the requirements associated with user mobility and dynamic change in computing environments. The infrastructure continuously monitors user actions, or intentions, and triggers the reconfigurations required by suspending and resuming tasks, or by adjusting the service composition of currently active tasks. Furthermore, the infrastructure continuously monitors the environment for both failures and opportunities for improvement, and automatically performs reconfigurations as deemed appropriate.

Our work sheds light on the coordination of adaptations enacted at different levels. We distinguished three different levels at which adaptation is enacted: first, at task management level, the infrastructure reacts to user mobility and to changes in user needs. Second, at environment management level, the infrastructure reacts to changes of supplier availability, and to consistent trends in quality of service. And third, at the level of applications, components react to fine-grain variations of resources.

We argued that, for systems that support daily professional and personal activities of users, adaptation needs to be driven by knowledge about user needs and preferences. Our architectural framework introduces a layer dedicated to capture such knowledge, and specifies the protocols to disseminate it and to coordinate its effective use.

We argued that designs that rely on ad hoc mechanisms inside applications to capture knowledge about user tasks make it very hard to match local adaptation policies to the user preferences for each task; 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 to wherever it is needed.

Note on Legacy Applications

For our research, we implemented service suppliers by wrapping existing applications to conform to the infrastructure's APIs. We have implemented suppliers that wrap 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.[2]

The wrapper code (residing in the ME layer) effectively 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. The richer the native APIs offered by the wrapped applications, the better job we can do in recovering the user-perceived state of services. Fortunately, providing such APIs is a growing tendency in the industry.

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

In our experience, doing a usable first-cut integration of one application into our infrastructure takes an experienced graduate student an average of two week, time on task. This includes studying the application's APIs, mapping the application-specific state into a more generic set of concepts in the user-perceived state of the service, and implementing the translator between the generic APIs in our infrastructure and the application-specific APIs. Typically, about ten user-level state parameters are recovered in this first cut. For example, for a text editor, things like currently open files, window position, size and scroll; cursor positions, editing overstrike, etc. For a web browser, the navigation history, current page, window settings (as before), etc.

Controlling the policies of resource-aware applications proved to be more challenging. 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 be an involved task.[3] Second, commercial software, which either doesn't expose mechanisms to control the adaptation policies in the offered APIs, or for which we often can not observe a reliable correlation between the controls transmitted to the application and its actual behavior – consistently greedy. But here also there is reason for being optimist: recent versions of media streaming applications offer a rich API to control the resource demand and QoS tradeoffs of the application. Our experiments with, for instance, RealOne Player indicate a good correlation between the controls and the actual behavior with respect to resource adaptation and QoS tradeoffs.

References

1. Balan, R., Satyanarayanan, M., Park, S., Okoshi, T.: Tactics-Based Remote Execution for Mobile Computing. *Procs of the 1st Intl Conf on Mobile Systems, Applications, and Services (MobiSys'03)*, pp 273-286, San Francisco, May 2003.
2. Capra, L., Emmerich W., Mascolo, C.: A Micro-Economic Approach to Conflict Resolution in Mobile Computing. *Proc Foundations of Software Engineering* (ACM SIGSOFT/FSE), 2002.
3. Cardoso, J. *Quality of Service and Semantic Composition of Workflows*. PhD thesis, University of Georgia, 2002.
4. Cheng, S.W. et al.: Software Architecture-based Adaptation for Pervasive Systems. *Proc of the Intl Conf on Architecture of Computing Systems: Trends in Network and Pervasive Computing*, April 2002. Springer LNCS Vol. 2299, 2002.
5. Cristian, F.: Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM*, 34(2):56-78, 1991.
6. Dey, A.: Understanding and Using Context. *Personal and Ubiquitous Computing Journal*, 5(1), pp 4-7, 2001.
7. Flinn, J., de Lara, E. et al.: Reducing the Energy Usage of Office Applications. *Proc. IFIP/ACM Intl Conf on Distributed Systems Platforms (Middleware)*, 2001.
8. Garlan, D., Cheng, S.W., Schmerl, B.: Increasing System Dependability through Architecture-Based Self-repair. *Architecting Dependable Systems*, R. Lemos, C. Gacek, A. Romanovsky (Eds), Springer-Verlag, 2003.

[3] We did extensive work in integrating and controlling the adaptive behavior of one adaptive application originating at a research project [1]. However, that work involved a close collaboration with one researcher working on the adaptation mechanisms of the application.

9. Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste P.: Project Aura: Towards Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 21(2), April-June, 2002.
10. Georgiadis, I., Magee, J., Kramer, J. Self-Organising Software Architectures for Distributed Systems. *Proc. ACM SIGSOFT Wksp on Self-Healing Sys. (WOSS'02)*, November 2002.
11. Hiltunen, M., Schlichting, R.: Adaptive Distributed and Fault-Tolerant Systems, *International Journal of Computer Systems Science and Engineering*, 11(5), pp 125-133, 1996.
12. Jini. www.jini.org.
13. 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.
14. de Lara, E., Wallach, D. S., Zwaenepoel, W.: Puppeteer: Component-based Adaptation for Mobile Computing. *Proc 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, 2001.
15. Lee, C., et al.: A Scalable Solution to the Multi-Resource QoS Problem. *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 1999.
16. Kon, F., et al. Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. *Proc USENIX Conference on OO Technologies and Systems (COOTS)*, 2001.
17. Narayanan, D., Flinn, J., Satyanarayanan, M.: Using History to Improve Mobile Application Adaptation. *Proc 3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, 2000.
18. Neugebauer, R., McAuley, D.: Congestion Prices as Feedback Signals: An Approach to QoS Management. *Proc ACM SIGOPS European Workshop*, 2000.
19. Noble, B., et al. Agile Application-Aware Adaptation for Mobility. Proc of the 16th ACM Symp on Operating Systems Principles (SOSP'97). *Operating Systems Review 31(5)*, ACM Press, pp 276-287, October 1997.
20. Pisinger, D.: An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, pp 114, 1999.
21. Poladian, V., et al.: 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.
22. Service Location Protocol. http://www.ietf.org/html.charters/svrloc-charter.html.
23. Sousa, J.P., Garlan, D.: The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. *Carnegie Mellon Technical Report CMU-CS-03-183*, August 2003.
24. Zeng, L., Benatallah, B., et al.: QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5), pp 311-327, 2004.