Task-Based Adaptation for Ubiquitous Computing

João Pedro Sousa, Vahe Poladian, David Garlan, *Member, IEEE*, Bradley Schmerl, *Member, IEEE*, and Mary Shaw, *Fellow, IEEE*

***Abstract*—An important domain for autonomic systems is the area of ubiquitous computing: users are increasingly surrounded by technology that is heterogeneous, pervasive, and variable. In this paper we describe our work in developing self-adapting computing infrastructure that automates the configuration and reconfiguration of such environments. Focusing on the engineering issues of self-adaptation in the presence of heterogeneous platforms, legacy applications, mobile users, and resource variable environments, we describe a new approach based on the following key ideas: 1) explicit representation of user tasks allows us to determine what service qualities are required of a given configuration; 2) decoupling task and preference specification from the lower level mechanisms that carry out those preferences provides a clean engineering separation of concerns between what is needed and how it is carried out; and 3) efficient algorithms allow us to calculate in real time near-optimal resource allocations and reallocations for a given task.**

***Index Terms*—Multifidelity applications, resource-aware computing, self-adaptation, ubiquitous computing.**

I. Introduction

SELF-ADAPTIVE systems are becoming increasingly important. What was once the concern of specialized systems, with high availability requirements, is now recognized as being relevant to almost all of today's complex systems [10], [20]. Increasingly, computing systems that people depend on cannot be taken off-line for repair—they must adapt to failures in environments that are not entirely under the control of the system implementers and they must adjust their runtime characteristics to accommodate changing loads, resources, and goals.

One particularly important domain for self-adaptation is the area of ubiquitous computing. Today users are surrounded by technology that is heterogeneous, pervasive, and variable. It is heterogeneous because computation can take place using a wide variety of computing platforms, interfaces, networks, and services. It is pervasive through wireless and wired connectivity that pervades most of our working and living environments. It is variable because resources are subject to change: users can move from resource-rich settings (such as workstations and high-bandwidth networks in an office) to resource-poor environments (such as a PDA in a park).

Manuscript received October 15, 2004; revised April 27, 2005. This work was supported in part by the National Science Foundation under Grant ITR-0086003, in part by the Sloan Software Industry Center, Carnegie Mellon University, and in part by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. This paper was recommended by Guest Editors R. Sterritt and T. Bapty.

The authors are with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: jpsousa@cs.cmu.edu; poladian@cs.cmu.edu; david.garlan@cs.cmu.edu; schmerl@cs.cmu.edu; mary.shaw@cs.cmu.edu).

Digital Object Identifier 10.1109/TSMCC.2006.871588

Coping with this situation requires automated mechanisms. In particular, ideally systems should be able to adapt to user mobility, recover from service failures and degradations, and allow continuity across diverse environments. Without automated mechanisms to support this kind of adaptation, users become increasingly overloaded with distractions of managing their system configurations; alternatively, they may simply opt not to use the capabilities of their environments.

This automation raises a number of serious engineering challenges: How can one determine when reconfiguration is appropriate? Assuming reconfiguration is desirable, how does one determine a satisfactory allocation of resources, particularly if there are multiple ways to support a given computing task or limitations on the resource pool? How can users instruct the system about the kinds of adaptation that are desired, without becoming bogged down in low-level system details? How can one add adaptation mechanisms to the everyday computing environments that users are familiar with, e.g., text editors, spreadsheets, video viewers, browsers, etc.

In this paper we describe our experience over the past five years of developing self-adapting ubiquitous computing infrastructure that automates the configuration and reconfiguration of everyday computing environments. Focusing on the engineering issues of providing self-adaptation in the presence of heterogeneous platforms, legacy applications, mobile users, and resource variable environments, Project Aura [9] has developed an approach that we believe addresses each of the questions above. The key ideas behind this work are the following: 1) explicit representation of user tasks allows us to determine what service qualities are required of a given configuration; 2) decoupling task and preference specification from the lower level mechanisms that carry out those preferences provides a clean engineering separation of concerns between what is needed and how it is carried out; and 3) efficient algorithms allow us to calculate in real time near-optimal resource allocations and reallocations for a given task.

This paper is organized as follows. Section II describes our work in the context of related research. Section III outlines the research challenges in making self-adaptive systems task-aware, describes the Aura architecture, and illustrates how the architecture addresses such research challenges. In Section IV, we elaborate on the specifics of supporting such an architecture: how the requirements and user preferences are captured for each task, the formal underpinnings of the internal representation of tasks and preferences, and the algorithm that supports automatic system configuration and self-adaptation. In Section V we evaluate the effectiveness of the approach and consider lessons learned from this work. Section VI presents the conclusions.

II. Related Work

Currently, adaptive systems fall into two broad categories: fault-tolerant systems and fidelity-aware systems. First, fault-tolerant systems react to component failure, compensating for errors by using a variety of techniques such as redundancy and graceful degradation [6], [13]. 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.). The primary goal of these systems is to prevent or delay large-scale system failure.

Second, fidelity-aware systems react to resource variation: components adapt their computing strategies so that they can function optimally with the current set of resources (bandwidth, memory, CPU, power, etc.) [8], [17], [22], [24]. 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 multicomponent integration, configuration, and reconfiguration. Although somewhat related, this kind of automatic configuration is distinct from the automatic configuration being investigated in [21]. In [21], the configuration is taken in the sense of *building and installing* new applications into an environment, whereas in our work it is taken in the sense of *selecting and controlling* applications so that the user can go about his tasks with minimal disruption.

Our work addresses two of the five principles of outonomic computing, which were first introduced in [16]. Specifically, Aura provides self-optimization and self-healing for everyday user tasks in the domain of ubiquitous computing.

Our work leverages microeconomic principles to determine optimal resource allocation. In this respect, our work is similar to [2], which addresses resource allocation in large-scale enterprise deployments. Computing the utility (value) of different resource allocation schemes is expensive. To help mitigate that problem, [2] describes a cooperative mechanism for *incrementally* eliciting utility. In our work, we separate elicitation of inputs into two levels. Specifically, we rely on history-based profiling to obtain application resource requirements for a particular level of quality of service (see [22]) and explicitly acquire user preferences for quality of service using techniques and user interfaces described in this paper.

Resource scheduling [15], resource allocation [18], [23], and admission control have been extensively addressed in research. From an analytical point of view, closest to our work are Q-RAM [18], a resource reservation and admission control system maximizing the utility of a multimedia server based on preferences of simultaneously connected clients; Knapsack algorithms [25]; 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 [3] using sealed bid auctions. While our work shares utility-theoretic concepts with [3], the problem solved in our work is different. In [3], the objective was 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 [5], [11], [12]. 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 [14], [26].

III. Task-Based Self-Adaptation

A. Task-Aware Systems

A central tenet of our work is that systems are used to carry out high-level activities of users: planning a trip, buying a car, communicating with others, etc. In today's systems these activities and goals are implicit. Users must map their tasks to computing systems by invoking specific applications (document editors, e-mail programs, spreadsheets, etc.) on specific files, with knowledge of specific resources. In a ubiquitous computing world with shifting resources and increased heterogeneity, the cognitive load required for users to manage this manually quickly becomes untenable.

In contrast, a task-aware system makes user tasks explicit by encoding user goals and by providing a placeholder to represent the quality attributes of the services used to perform those tasks. So, for example, for a particular task, in the presence of limited bandwidth, the user may be willing to live with a small video screen size, while in another task reducing the frame rate would be preferable. In task-aware systems, users specify their tasks and goals, and it is the job of the *system* to automatically map them into the capabilities available in the ubiquitous environment.

Once such information is represented, a self-managing system can in principle query the task to determine both when the system is behaving within an acceptable envelope for the task and also can choose among alternative system reconfigurations when it is not.

However, a number of important research questions arise, and the way we answer them strongly influences the way we look at and build task-aware systems:

1) How do we represent a task? What encoding schemes can best be used to capture the user's requirements for system quality?
2) How should we characterize the knowledge for mapping a user task to a system's configuration? As a user moves from task to task, different configurations will be appropriate, even for the same set of applications.
3) Should we trigger an adaptation as soon as an opportunity for improvement is detected, or should we factor in how

distracting the change will be to the user against how serious the fault is?

4) Is the binary notion of fault enough, or do we need to come up with a measure of fault "hardness"—a continuum between "all is well," and "the system is down"?
5) What is the length of time that the user is expected to carry out the current task? What are likely other tasks that the user will work on next?

Over the past five years we have been experimenting with various answers to these questions. Centered on a large ubiquitous computing research project, Project Aura [9], we have evolved a system that, in brief, addresses these questions as follows.

1) We represent *a task as a set of services*, together with a set of quality attribute preferences expressed as multidimensional utility functions, possibly conditioned by context conditions.
2) We define *a vocabulary for expressing requirements*, which delimits the space of requirements that the automatic reconfiguration can cover. The set of requirements for a particular task expresses which services are needed from the system as well as the fidelity constraints that make the system adequate or inadequate for the task at hand. The required services are dynamically mapped to the available components and the fidelity constraints are mapped to resource-adaptation policies.
3) We incorporate the notion of *cost of reconfiguration* into the evaluation of alternative reconfigurations. This cost captures user's intolerance for configuration changes by the infrastructure. A high cost of reconfiguration will make the system highly stable, but frequently less optimal; a low cost of configuration will permit the system to change frequently, but may introduce more user distraction from reconfigurations.
4) We invert the notion of fault by adopting an econometric-based notion of *task feasibility*, ranging from 0 (the task is not feasible under the current system conditions) to 1 (system is totally appropriate for the current task). This enables an objective evaluation of configuration alternatives, regardless of the sources of change (both changes to the task and also to the availability of resources and components).

We now describe the system architecture that permits such task-based self-adaptation and elaborate on these decisions.

B. Aura Layers

The starting point for understanding Aura is a layered view of its infrastructure together with an explanation of the roles of each layer with respect to task suspend/resume and dynamic adaptation. Table I summarizes the relevant terminology.

The infrastructure exploits knowledge about a user's tasks to automatically configure and reconfigure the environment on behalf of the user. First, the infrastructure needs to know *what* to configure for; that is, what a user needs from the environment in order to carry out his or her tasks. Second, the infrastructure needs to know *how* to best configure the environment, i.e., it needs mechanisms to optimally match the user's needs to the capabilities and resources in the environment.

TABLE I
TERMINOLOGY

Task	An everyday activity such as preparing a presentation or writing a report. Carrying out a task may require obtaining several *services* from an *environment* as well as accessing several *materials*.
Environment	The set of *suppliers*, *materials*, and *resources* accessible to a user at a particular location.
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.
Supplier	An application or device offering *services,* e.g., a printer.
Material	An information asset such as a file or data stream.
Capabilities	The set of *services* offered by a *supplier* or by a whole *environment*.
Resources	Are 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-level state of a task	User-observable set of properties in the *environment* that characterize the support for the task. Specifically, the set of *services* supporting the task, the user-level settings (preferences, options) associated with each of those services, the *materials* being worked on, user-interaction parameters (window size, cursors…), and the user's preferences with respect to quality of service tradeoffs.

TABLE II
SUMMARY OF THE SOFTWARE LAYERS IN THE INFRASTRUCTURE

Layer	Mission	Roles
Task management	**what** does the user need	monitor the user's task, context and preferences map the user's task to needs for services in the environment complex tasks: decomposition, plans, context dependencies
Environment management	**how** to best configure the environment	monitor environment capabilities and resources map service needs, and user-level state of tasks to available suppliers ongoing optimization of the utility of the environment relative to the user's task
Env.	support the user's task	monitor relevant resources fine grain management of QoS/resource tradeoffs

In our architecture, each of these two subproblems is addressed by a distinct software layer: 1) the *task management* layer determines *what* the user needs from the environment at a specific time and location and 2) the *environment management* layer determines *how* to best configure the environment to support the user's needs.

Table II summarizes the roles of the software layers in the infrastructure. The top layer, *task management* (TM), captures knowledge about user tasks and associated intent. Such knowledge is used to coordinate the configuration of the environment upon changes in the user's task or context. For instance, when the user attempts to carry out a task in a new environment, TM coordinates access to all the information related to the user's task and negotiates task support with environment management (EM). TM also monitors explicit indications from the user and events in the physical context surrounding the user. Upon getting indication that the user intends to suspend the current task or

resume some other task, TM coordinates saving the user-level state of the suspended task and reinstantiates the resumed task, as appropriate. TM may also capture complex representations of user tasks (out of scope of this paper) 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).

The EM layer maintains 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 both heterogeneity and dynamic change in the environments. With respect to heterogeneity, when the user needs a service, such as speech recognition, EM will find and configure a "supplier" for that service among those available in the environment. With respect to dynamic change, the existence of explicit models of the capabilities in the environment enables automatic reasoning when those capabilities change dynamically. The EM adjusts such a mapping automatically in response to changes in the user's needs (adaptation initiated by TM) and changes in the environment's capabilities and resources (adaptation initiated by EM). In both cases adaptation is guided by the maximization of a *utility function* representing the user's preferences (see Section IV-A1).

The *environment layer* comprises the applications and devices that can be configured to support a user's task. Configuration issues aside, these suppliers interact with the user exactly as they would without the presence of the infrastructure. The infrastructure steps in only to automatically configure those suppliers on behalf of the user. The specific capabilities of each supplier are manipulated by EM, which acts as a translator for the environment-independent descriptions of user needs issued by TM.

By factoring models of user preferences and context out of individual applications, the infrastructure enables applications to apply the adaptation policies appropriate for each task. This knowledge is very hard to obtain at the application level, but once it is determined at the user level, by TM, it can easily be communicated to the applications supporting the user's task.

A detailed description of the architecture, including the formal specification of the interactions between the components in the layers defined above, is available in [27].

C. Examples of Self-Adaptation

To clarify how this design works, we illustrate how the infrastructure outlined in Section III-B handles a variety of examples of self-adaptation, ranging from traditional repair in reaction to faults, to reactions to positive changes in the environment, to reactions to changes in the user's task.

To set the stage, suppose that Fred is engaged in a conversation that requires real-time speech-to-speech translation. To do this task, assume the Aura infrastructure has assembled three services: speech recognition, language translation, and speech synthesis. Initially both speech recognition and synthesis are running on Fred's handheld. To save resources on Fred's handheld, and since language translation is computationally intensive, but has very low demand on data flow (the text representation of each utterance), the translation service is configured to run on a remote server.

1) Fault Tolerance: Suppose now that there is loss of connectivity to the remote server, or equivalently, that there is a software crash that renders it unavailable. Live monitoring at the EM level detects that the supplier for language translation is lost. The EM looks for an alternative supplier for that service, e.g., translation software on Fred's handheld, activates it, and automatically reconfigures the service assembly.

2) Resource/Fidelity Awareness: Computational resources in Fred's handheld are allocated by the EM among the services supporting Fred's task. For computing optimal resource allocation, the EM uses each supplier's spec sheet (relating fidelity levels with resource consumption), live monitoring of the available resources, and the user's preferences with respect to fidelity levels. Suppose that during the social part of the conversation, Fred is fine with a less accurate translation, but response times should be snappy. The speech recognizer, as the main driver of the overall response time, gets proportionally more resources and uses faster, if less accurate, recognition algorithms. When the translation service is activated on Fred's handheld in response to the fault mentioned, resources become scarcer for the three services. However, having the knowledge about Fred's preferences passed upon service activation, each supplier can react appropriately by shifting to computation strategies that save response times at the expense of accuracy [1].

3) Soft Fault (Negative Delta): Each supplier issues periodic reports on the quality of service (QoS) actually being provided; in this example, response time and estimated accuracy of recognition/translation.[1] Suppose that at some point during the conversation, Fred brings up his calendar to check his availability for a meeting. The suppliers for the speech-to-speech translation task, already stretched for resources, reduce their QoS below what Fred's preferences state as acceptable. The EM detects this soft fault and replaces the speech recognizer by a lightweight component, which although unable to provide as high a QoS as the full-fledged version performs better under suboptimal resource availability.

4) Soft Fault (Positive Delta): Suppose that at some point, the language translation supplier running on the remote server becomes available again. The EM detects the availability of a new candidate to supply a service required by Fred's task and compares the estimated utility of the candidate solution against the current one. If there is a clear benefit, the EM automatically reconfigures the service assembly. In calculating the benefit, the EM factors in a cost of change, which is also specified in the user's preferences associated with each service. This mechanism introduces hysteresis in the reconfiguration behavior; thus avoiding oscillation between closely competing solutions.

[1] Additionally, the EM uses these periodic QoS reports to monitor the availability of the suppliers, in a heartbeat fashion.

5) Task QoS Requirements Change: Suppose that at some point Fred's conversation enters a technical core for which translation accuracy becomes more important than fast response times. The TM provides the mechanisms, if not to recognize the change automatically based on Fred's social context, at least to allow Fred to quickly indicate his new preferences; for instance, by choosing among a set of preference templates. The new preferences are distributed by the TM to the EM and all the suppliers supporting Fred's task. Given a new set of constraints, the EM evaluates the current solution against other candidates, reconfigures, if necessary, and determines the new optimal resource allocation. The suppliers that remain in the configuration, upon receiving the new preferences, change their computation strategies dynamically, e.g., by changing to algorithms that offer better accuracy at the expense of response time.

6) Task Suspend/Resume: Suppose that after the conversation, Fred wants to resume writing one of his research papers. Again, the TM provides the mechanisms to detect, or for Fred to quickly indicate, his change of task. Once the TM is aware that the conversation is over, it coordinates with the suppliers for capturing the user-level state of the current task, if any, and with the EM to deactivate (and release the resources for) the current suppliers. The TM then analyses the description it saved the last time Fred worked on writing the paper, recognizes which services Fred was using, and requests those from the EM. After the EM identifies the optimal supplier assignment, the TM interacts with those suppliers to automatically recover the user-level state where Fred left off. See [27] for a formal specification of such interactions.

7) Task Service Requirements Change: Suppose that while writing his paper, Fred recognizes that it would be helpful to refer to a presentation he gave recently to his research group. The TM enables Fred to explicitly aggregate viewing the presentation to the ongoing task. As soon as a new service is recognized as part of the task, the TM requests an incremental update to the EM, which computes the optimal supplier and resource assignment for the new task definition and automatically performs the required reconfigurations. Similarly, if Fred decides some service is no longer necessary for his task, he can let the TM know and the corresponding (incremental) deactivations are propagated to the EM and suppliers. By keeping the TM up-to-date with respect to the requirements of his tasks, Fred benefits from both the automatic incremental reconfiguration of the environment and from the ability to suspend/resume exactly the set of services that he considers relevant for each task.

D. Controlling Self-Adaptation

Aura can be viewed as a closed-loop control system, which senses, actuates, and controls the runtime state of the environment, based on input from the user. Each layer reacts to changes in user tasks and in the environment at a different granularity and time scale. The TM acts at a human perceived time scale (minutes), evaluating the adequacy of sets of services to support the user's task. The EM acts at a time scale of seconds, evaluating the adequacy of the mapping between the requested services and specific suppliers. Adaptive applications (fidelity-aware and context-aware) choose appropriate computation tactics at a time scale of milliseconds.

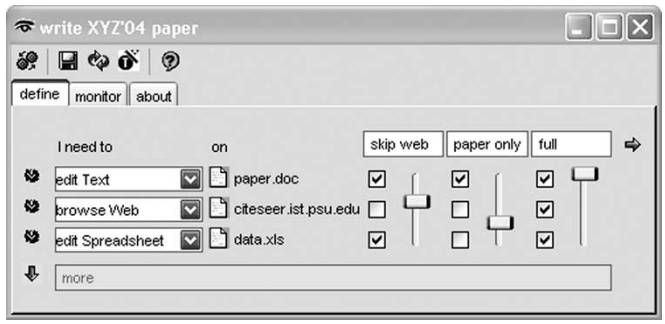


Fig. 1. Fred's task definition for writing XYZ'04 paper.

Specifically, let us see how the infrastructure handles the changes for a number of scenarios described in Section III-C.

1) *Task service or QoS requirements change*: The TM immediately coordinates a change in the environment, by adding, disbanding, or replacing suppliers, or changing their QoS policies appropriately.
2) *Hard fault (failure of a running supplier)*: The EM immediately replaces the failed supplier with an alternative.
3) *Soft fault (negative or positive delta in resources)*: The suppliers immediately adjust their QoS to the available resources. The EM periodically computes a new near-optimal configuration, which may imply swapping suppliers (we have not yet experimented with varying the time scale of reaction of the EM).

IV. Supporting Task-Based Adaptation

A. Defining Task Requirements

The user expresses the requirements for a task by specifying the services needed and the associated preferences. A shared vocabulary of services and service-specific quality dimensions must exist between the user and the system. Developing such a vocabulary is a subject of related research and out of the scope of this paper (see for instance [7]), but we give insights to the essential characteristics of such a vocabulary [27], [28].

For instance, to address user mobility across different machines, we use terms that are generic enough to be meaningful on different platforms. For example, a task may capture the fact that the user needs to *edit text*, as opposed to capturing the fact that he needs to use Microsoft Word.

To make these ideas concrete, let us suppose that Fred is about to start writing a new paper. Fred starts by pressing the down arrow at the bottom of an empty task definition window and selecting *edit text* (Fig. 1). 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 an especially relevant page with the task so that it is brought up automatically every time the task is resumed. To do so, Fred simply drags the page shortcut out of the browser and into the *more* field of the task window (the default *browse web* appears automatically). Later, Fred decides to start analyzing the

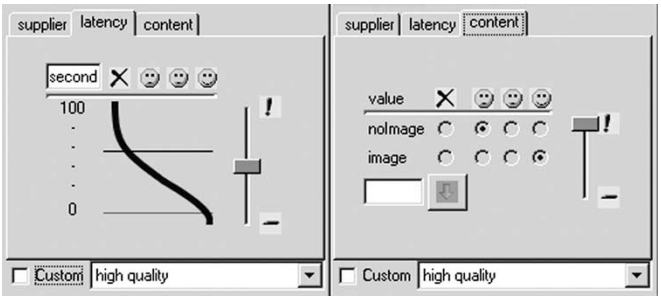


Fig. 2. QoS preferences for the web-browsing service.

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.

Note that the infrastructure imposes no constraints on the user's work. This comes from recognizing that many user activities are spontaneous and short lived, and need not be classified as pertaining to a particular task. However, once the user recognizes an enduring association with a task, the infrastructure makes it easy to update the task definition on the fly.

In addition to specifying the services required by each task, the user may specify preferences with respect to how the environment should be configured. User preferences (and their formal representation, *utility functions*) used in our work have three parts. First, *configuration preferences* capture preferences with respect to the set of services to support a task. Second, *supplier preferences* capture which specific suppliers are preferred to provide the required services. And third, *QoS preferences* capture the acceptable QoS levels and preferred tradeoffs.

The right-hand side of Fig. 1 defines Fred's configuration preferences for the task, that is, alternative operation modes and their order of preference. The (default) *full* configuration includes all the activities defined for the task. In addition, Fred also specifies the *skip web* degraded-mode configuration for when the circumstances are such that either a browser or connection are not available or that the QoS is so poor (for instance, due to low bandwidth) that Fred would rather focus on the other activities. Fred also permits the *paper only* configuration for last resort circumstances, for instance when having only a handheld with extremely limited resources. Note that Fred can define as many or as few operating modes as he feels appropriate and indicate his relative preference for each by sliding the corresponding bar.

Suppose that for typing the notes (*edit text* service), Fred prefers *MSWord* over *Emacs*, and is unwilling to use the *vi* editor at all. This is an example of *supplier preferences*. Note that representing supplier preferences by discriminating the supplier type is a compact representation for the preferences with respect to the availability of desired features, such as spell checking or richness of editing capabilities, as well as to the user's familiarity with the way those features are offered. For the sake of space, the user interface for specifying supplier preferences is not shown, but it is similar to the tabular form shown in Fig. 2.

Suppose now that Fred will be browsing the web over a wireless network link. Suppose that the bandwidth suddenly drops: should the browser preserve the full quality of web pages at the expense of download time or reduce the quality, for instance by skipping images? The answer depends on Fred's *QoS preferences* for the current task. For browsing citations, Fred probably will be fine with dropping images and banners, with benefits in response times. However, for browsing a museum's site on painting or online mapping, Fred may prefer full page quality to be preserved at the expense of download times.

Let us now look at the user interface of defining QoS preferences. Fig. 2 shows an example of QoS preferences for the web-browsing service. The service has two dimensions: *latency* and *content*. Latency refers to the average time a web page takes to load after being requested. Content refers to the richness of the web page content. Latency is numeric and is expressed in seconds. The user manipulates the good and bad thresholds by dragging the green (lighter) and red (darker) handles, respectively.[2] Note that the utility space is represented simply using four intervals: from the lowest where the user prefers the configuration not to be considered, represented by a cross (✕), to the highest corresponding to satiation, represented by a happy face (☺). The slide bar associated to each dimension captures how important, that is how much the user cares, about variations along that dimension.

We do not expect every user to interact with the system at this kind of detail for every task. Rather, the infrastructure provides a set of templates for each service type, corresponding to common situations. For instance, the web-browsing service includes the *high quality* template shown in Fig. 2, as well as the *fast loading* template, where the latency thresholds are stricter and the content threshold is more relaxed. The user can choose which preference template to apply to each service when defining a task (Fig. 1) or, by selecting customized tuning, manipulate preferences directly.

To make preferences easier to both elicit and process, we make two simplifying assumptions. First, preferences are modeled independently of each other. In other words, the utility function for each aspect captures the user's preferences for that aspect independently of others. Second, preferences fall into two categories: those characterized by enumeration and those characterized by numeric values. Supplier preferences are characterized by enumeration (e.g., *MSWord, Emacs*, or *other*) and so are QoS dimensions such as audio fidelity (e.g., *high, medium*, and *low*). For these, the utility function takes the form of a discrete mapping to the *utility space* (see below).

For preferences characterized by numeric values, we distinguish two intervals: one where the user considers the quantity to be good enough for his task and the other where the user considers the quantity to be insufficient. *Sigmoid* functions, which look like smooth step functions, characterize such intervals and provide a smooth interpolation between the limits of those intervals (see Fig. 2). Sigmoids are easily encoded by just

[2]The upper limit of the scale adjusts automatically between the values 10, 50, 100, 500, and 1000, further changes being enabled by a change in unit.

```
<utility combine="product">
  <QoSdimension name="latency" type="float">
    <function type="sigmoid" weight=".5">
      <thresholds good="3" bad="60" unit="second"/>
    </function>
  </QoSdimension>
  <QoSdimension name="vocabulary" type="enum">
    <function type="table" weight="0.7">
      <entry x="noImage" f_x=".2"/>
      <entry x="image" f_x="1"/>
    </function>
  </QoSdimension>
</utility>
```

Fig. 3. Internal representation of the QoS preferences in Fig. 2.

two points: the values corresponding to the knees of the curve that define the limits *good* of the good-enough interval and *bad* of the inadequate interval. The case of "more-is-better" qualities (e.g., *accuracy*) are as easily captured as "less-is-better" qualities (e.g., *latency*) by flipping the order of the *good* and *bad* values. In case studies evaluated so far, we have found this level of expressiveness to be sufficient.

Fig. 3 shows the internal representation of the preferences captured in Fig. 2. Note that the infrastructure creates user interfaces like the one in Fig. 2 dynamically, based on the internal representation, which in turn is updated by manipulating the representations in the interface.

B. Formal Underpinnings

This section describes how user preferences, as defined in the previous section, guide the automatic configuration and reconfiguration of the environment. Our approach is based on finding the best match between the user's needs and preferences for a specific task, as well as the environment's capabilities. This framework is used both to find the optimal initial configuration and to address the ongoing optimization of the support for the user's tasks.

In practice, finding such a match corresponds to a constrained maximization problem. The function to be maximized is a *utility function* that denotes the user preferences and the *constraints* are the environment's capabilities and available resources. The result of the maximization is an abstract measure of the feasibility of carrying out the task, given the current conditions in the environment.

Utility Space: Utility functions map the *capability space* (see next paragraph) onto the utility space. The latter is represented by the real number interval [0, 1]. The user will be happier the higher the values in the utility space. The value 0 corresponds to the environment 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 feasibility of the specific task.

Capability Space: The capability space C_s corresponding to service s is the Cartesian product of the individual quality dimensions d of the service:

$$C_s \hat{=} \otimes_{d \in QoS\dim(s)} \mathrm{dom}(d).$$

For example, possible quality dimensions for the *play video* service are *frame update rate*, the *frame size*, and *audio quality*. Thus, the capability space of video playing is three dimensional. Cartesian product is used to combine the capability space of two services. For distinct services s and t, their combined capability space is formally expressed as

$$C_{s \cup t} \hat{=} C_s \otimes C_t.$$

For example, a *web-browsing* service has two quality dimensions, *latency* and *page richness*, and *video playing* has three dimensions of quality. Thus the joint capability space of *video playing* and *web browsing* has five quality dimensions.

Typically, an application supports only a subset of the capability space corresponding to its various fidelities of output. In practice, approximating this subset using a discrete enumeration of points provides a reasonable solution, even if the corresponding capability space is conceptually continuous. For example, while it makes sense to discuss a video stream encoding of decimal frames per second, typically video streams are encoded at integer rates. Despite discrete approximation, our approach does allow the handling of a rich capability space. For example, the capability space of a specific video player application can have 90 points, which is made possible by combining five frame rates, six frame sizes, and three audio qualities. Such a capability space can be made possible by encoding the same video in multiple frame rates, frame size, and audio quality, and possibly leveraging application-specific features such as video smoothing.

An application profile specifies a discrete enumeration of the capability points supported by an application and corresponding resource demand for each point. Note that specific *mechanisms* for obtaining and expressing application profiles exist. As demonstrated in [22], resource demand prediction based on *historical* data from experimental profiling is both feasible and accurate. Further, *metadata* and *reflection* can be used to express application profiles [4].

Application profiles describe the relationship between the capability points supported by applications and the corresponding resource requirements. Formally, the quality resource mapping of supplier p is a partial function from the capability space of service s to the resource space: $QoS\mathrm{prof}_p : C_s \mapsto R$. The range of the function is the subset of the capability space that is supported by the supplier.

Resource Space: The resource space R is the Cartesian product of the individual resource dimensions r of the entire environment E

$$R \hat{=} \otimes_{r \in RES\dim(E)} \mathrm{dom}(r).$$

Examples of resource dimensions are CPU cycles, network bandwidth, memory, and battery. The actual number of resource dimensions is dependent on the environment.

Utility Functions: There is one utility function for each alternative configuration for a given task. The feasibility of the task corresponds to the best utility among the alternatives, weighted by the user's preference for each alternative. The utility function for each configuration has two components, reflecting QoS and supplier preferences, respectively.

QoS preferences specify the utility function associated with each QoS dimension. The names of the QoS dimensions are

part of the vocabulary shared between the user and the system. The utility of service s as a function of the QoS is given by

$$U_{QoS}(S) \hat{=} \prod_{d \in QoS\dim(s)} F_d^{c_d}$$

where for each QoS dimension d of service s, $F_d : \mathrm{dom}(d) \rightarrow (0,1]$ is a function that takes a value in the domain of d and the weight $c_d \in [0,1]$ reflects how much the user cares about QoS dimension d. As an example, *video playing* has a QoS dimension of *frame update rate*. The function $f_{\text{framerate}}$ gives utility for various frame rates and $c_{\text{framerate}}$ specifies the weight of frame rate.

To evaluate the assignment of specific suppliers, we employ a supplier-preference function, which is a discreet function that assigns a score to a supplier, based on its type. We also account for the *cost of switching* from one supplier to another at runtime.

Precisely, the utility of the supplier assignment for a set a of requested services is

$$U_{\mathrm{Supp}}(a) \hat{=} \prod_{s \in a} h_s^{x_s} \cdot F_s^{C_s}$$

where for each service s in the set a, $F_s : \mathrm{Supp}(S) \rightarrow (0,1]$ is a function that appraises the choice for the supplier for s and the weight $c_s \in [0,1]$ reflects how much the user cares about the supplier assignment for that service. $h_s^{x_s}$ expresses a change penalty as follows: h_s indicates the user's tolerance for a change in supplier assignment. A value close to 1 means that the user is fine with a change; the closer the value is to zero, the less happy the user will be. The exponent x_s indicates whether the change penalty should be considered ($x_s = 1$, if the supplier for s is being exchanged by virtue of dynamic change in the environment) or not ($x_s = 0$, if the supplier is being newly added or replaced at the user's request).

The overall utility is the product of the QoS preference and supplier preference. The overall utility over a set a of suppliers is

$$U_{\mathrm{overall}}(a) = \prod_{s \in a} h_s^{x_s} \cdot F_s^{c_s} \left(\prod_{d \in QoS\dim(s)} F_d^{c_d} \right).$$

1) Optimization Problem: The optimization problem is to find a supplier assignment a, and for each supplier p in this assignment, a capability point such that the utility is maximized:

$$\arg\max_{\substack{p_s \in \mathrm{Supp}(s) \\ f_d \in \mathrm{dom}(d)}} \prod_{s \in a} h_s^{x_s} \cdot F_s^{c_s}(p_s) \cdot \left(\prod_{d \in QoS\dim(s)} F_d^{c_d}(f_{p,d}) \right).$$

The maximization is over a set of constraints, which we express below. The capability constraint stating that the chosen point $f_{p,d}$ is in the capability space for supplier p as follows:

$$\forall_{p \in \mathrm{Supp}(s)} f_p = \otimes_{d \in QoS\dim(s)} f_{p,d} \in C_p.$$

And to ensure that the resource constraints are met

$$\sum_{p \in \mathrm{Supp}(s)} QoS\mathrm{prof}_p(f_p) \leq |\boldsymbol{R}|$$

where the summation is in the vector space $\boldsymbol{R}$ of resources and the inequality is observed in each dimension of that space. In nonmathematical terms, this constraint expresses the fact that the aggregate resource demand by all the suppliers cannot exceed the resource supply.

C. Algorithm and Analysis

In this section we solve the optimization problem. The optimization algorithm must be efficient to be usable at runtime. Two metrics we are interested in are the latency of computing an answer to a given instance of the problem and the computational overhead of the algorithm.

1) Algorithm: The algorithm works in three phases: 1) query; 2) generate; and 3) explore. In the first phase, it queries for relevant suppliers for each service in the task. In the second phase, it combines suppliers into configurations and ranks them only according to the supplier preference. In the third phase, it explores the quality space of the configurations. The pseudocode for the algorithm is shown in Fig. 4.

The double product term of the utility formula in B.1 allows for a clever exploration strategy. The outer product is the supplier preference score. It can be computed at the time the supplier assignment is known (in phase 2) and can be used as an upper bound for overall utility during the explore phase. Since overall utility is the product of supplier preference and QoS preference, and the latter is bounded by one, then maximum overall utility is bounded by supplier preference. The break in the loop in BestConfig takes advantage of this fact.

Consider a simple example. Assume that two services are requested. For each service, there are two possible suppliers: a_1 and a_2 for the first service and b_1 and b_2 for the second, yielding four possible configurations as shown in Table III. The search space can be divided into four quadrants, each representing the capability space of a specific configuration. We are searching for a point with the highest utility.

As noted, the maximum utility that can be achieved within each quadrant is bounded by the supplier preference portion of utility. These observations help provide a stop condition for the search: once a point is found that has an overall utility of Λ, there is no need to explore configurations with supplier preference portion of utility of less than Λ.

In Table III, the shades of each quadrant reflect the hypothetical values of the supplier preference portion of utility for each configuration: the darker the shade, the higher the value. Assume that these values are 0.8, 0.6, 0.4, and 0.2. Each of these values is an upper bound for maximum *overall utility* possible from the respective quadrant. We explore inside the quadrants, starting from the darkest. If the maximum utility for the quadrant a_1, b_1 is higher than 0.6, then we have found the best point in the entire space and can stop the search. And if not, we continue the search in quadrant a_2, b_1, and so on.

Exploring the quality space of a configuration is a variant of a 0-1 Knapsack problem, called multiple dimensional, multiple choice 0-1 Knapsack. Multiple dimensions refer to the multiple constraints that are present in the problem. Multiple choice refers to choosing one among a set of similar items. In our

```
HashMap SuppPrefs;     // supplier preferences
HashMap QoSPrefs;      // qos preferences
HashMap SuppReg;       // registered suppliers

Config BestConfig(Set reqstdSvcs){
  // 1. QUERY
  Map  suppListsBySvc;
  for each svc in reqstdSvcs do{
    List  suppList = null;
    Pref  suppPref = SuppPrefs.get(svc);
    // query for supp based on svc type
    suppList = query(SuppReg, svc, suppPref);
    suppListsBySvc.put(svc.type, suppList);
  }
  // 2. GENERATE configs, compute supp pref
  List configs = GenConfigs(suppListsBySvc);
  configs = sort(configs);
  // 3. EXPLORE the QoS space
  int indexBestConfig;
  float overallUtilBest = 0.0;
  for each i from configs.size-1 to 0 do {
    Config cCur = configs.get(i);
    if (overallUtilBest > cCur.suppPrefUtil)
        break;
    cCur = searchQoS(cCur, QoSPrefs);
    if (cCur.overallUtil > overallUtilBest){
        indexBestConfig = i;
        overallUtilBest = cCur.overallUtil;
    }
  }
  return configs.get(indBestConfig);
}

GenConfigs(Map suppListsBySvc){
  List configs = new List(MAX_INT);
  int depth = 0;
  Config partialConfig = null;
  GenConfigsRecur(depth, configs,
       suppListsBySvc, partialConfig);
}
GenConfigsRecur(int d, List configs,
   Map suppListsBySvc, Config partialConfig){
  if ( d == suppListsBySvc.size()){
    configs.add(new Config(partialConfig));
    return;
  }
  List suppList = suppListsBySvc.getByInd(d);
  for each supp in suppList do{
    partialConfig.add(d, suppList);
    GenConfigsRec(d+1, configs,
              suppListsBySvc, partialConfig);
    partialConfig.remove(d);
  }
}
```

Fig. 4. Pseudocode of the algorithm

TABLE III
STRUCTURE OF THE SEARCH SPACE

a_1, b_1	a_1, b_2
a_2, b_1	a_2, b_2

problem, resources map to knapsack dimensions and the capability space of one service maps to one set of similar items. This is a well-studied problem in the optimizations research, and is at the core of such optimization problems as winner determination in combinatorial algorithms. Lee *et al.* [18] and Pisinger [25] show the problem to be NP-complete and give approximation algorithms. Pisinger [25] gives an exact solution that is demonstrably fast on inputs drawn from certain probability distributions.

TABLE IV
NUMBER OF CONFIGURATIONS GENERATED AND EXPLORED FOR VARIOUS VALUES OF N AND Λ, MAXIMUM UTILITY ACHIEVED

	n					
	1	2	3	4	…	8
Generated	10	10^2	10^3	10^4	…	10^8
Λ = .9	2	3	4	5	…	8
Λ = .81	3	6	10	15	…	36
Λ = .73	4	10	20	35	…	120
Λ = .66	5	15	35	70	…	330

One of the approximating algorithms to the problem uses utility to resource ratio as a metric for ranking the capability points, it then applies greedy branch-and-bound and LP-relaxation to find a near-optimal answer. In the multiple resource case, quadratic weighted-average is used to compute a single resource currency from multiple resources and the solution to the single resource case is reused iteratively [18].

In our solution, SearchQoS invokes a third-party library called Q-RAM, the package described in [18].

2) Analysis: To analyze the running time of the algorithm, let

1) $\mathbf{n}$ be the number of requested services;
2) $\mathbf{P}$ be the total number of available suppliers;
3) $\mathbf{p}$ be the number of suppliers for a given service type;
4) $\mathbf{q}$ be the size of the capability space of a supplier.

$\mathbf{P}$ and $\mathbf{p}$ describe the *richness* of the environment and can potentially increase as more applications, hardware, and devices are made available. $\mathbf{q}$ describes the capability richness of a supplier. It is reasonable to assume that the size of the user task is limited to a small number of applications. Thus $\mathbf{n}$ is bounded.

Next we analyze the running time of the three phases.

1) The query phase retrieves items from a hashtable. Retrieving one item is logarithmic in the size of the hashtable. $\mathbf{n}$ retrievals from a hashtable of size $\mathbf{P}/\mathbf{p}$ take $O(\mathbf{n} * \log(\mathbf{P}/\mathbf{p}))$.
2) The generate phase is a recursion of depth $\mathbf{n}$, with a loop of size $\mathbf{p}$ at each level. Thus, it takes $O(\mathbf{p}^n)$.
3) The explore phase in the worse case takes $O(\mathbf{p}^n) * O$ (searchQoS). The size of the QoS space of a configuration of $\mathbf{n}$ suppliers each of which has a capability space of size $\mathbf{q}$ is $O(\mathbf{q}^n)$. The approximation algorithm we use can search that space in time $O(\mathbf{n} * \mathbf{q} * \log \mathbf{q})$ [18], [25]. Thus the explore phase takes $O(\mathbf{p}^n) * O(\mathbf{n} * \mathbf{q} * \log \mathbf{q})$ in the worst case and dominates all other terms. The first term $O(\mathbf{p}^n)$ presents a possible scalability bottleneck.

Let us demonstrate how the exploration strategy described earlier helps tackle this bottleneck. Recall the break condition in the explore phase (illustrated in the example introduced in Section IV-C1). The number of configurations that are explored will depend on the distribution of the supplier preference values and Λ, the highest achievable utility value. Let us assume an average number of suppliers per service $\mathbf{p} = 10$ and a specific distribution of supplier preference values that is uniform, i.e., the most preferred supplier scores 0.9°, the next one scores 0.9^1, etc. Table IV shows the number of configurations generated and the

TABLE V
SERVICES REQUIRED FOR THE TASK, THEIR QUALITY DIMENSIONS, AND AVAILABLE SUPPLIERS FOR EACH SERVICE*

Service	QoS dimensions	Available suppliers
Play Video	Frame rate, frame size, audio quality	Real One, Windows Media Player
Edit Text	None	TextPad, WordPad, NotePad, MS Word, Emacs
Browse Web	Latency, Content	Internet Explorer, Netscape, Opera

*These suppliers jointly allow a total of 2*5*3=30 combinations.

number of configurations that are actually explored depending on the value of maximum achievable utility Λ, and number of services in the task $\mathbf{n}$.

The first row shows the number of services. The second row shows the number of configurations generated, which is $\mathbf{p}^n$, in this case, 10^n. In each subsequent row, we show the number of configurations that are sufficient to explore, if the maximum utility shown in the first column in that row is actually achieved by some configuration. For instance, for a task with 4 requested services, even if the maximum utility achievable is as modest as $\Lambda = 0.6$, then the number of supplier configurations explored is 126, which is two orders of magnitude smaller than 10^4, the total number of configurations.

3) Reconfiguration: The algorithm also handles reconfiguration scenarios described in Section III-D. When there is a running configuration, the utility from the best computed configuration is compared with the *observed* utility of the running configuration and a switch is made if the latter is lower than the former. The cost of change introduces a kind of hysteresis, giving the currently running configuration preference. Because a user's tolerance to change might depend on a type of service, the model explicitly provides means to specify this.

V. EVALUATION

A. Case Study

In this section we report on a case study of automatically configuring an environment for reviewing a documentary video clip. The user watches the clip, taking notes, while browsing the net for information. Table V lists the services in the task and the QoS dimensions of each service.

We performed the case study in two steps. In the first step we collected application profile data, specified preferences, and identified resource limits. In the second step, we ran a prototype implementation of the algorithm.

1) Input Data Collection: As an experimental platform, we chose an IBM Thinkpad 30 laptop, equipped with 256 MB of memory, 1.6-GHz CPU, WaveLAN card, and Windows XP Professional. In power-saving mode, the CPU can run at a percentage of the maximum speed, effectively creating a tight CPU constraint.

The model requires three inputs: 1) user preferences; 2) application profiles; and 3) resource availability. For this experiment, we used synthetic preferences intended to be representative of the task. We identified several applications that supported various facets of the task. These applications were installed on the laptop. To obtain application profiles, we measured resource usage corresponding to a small set of capability points. We performed this profiling *offline*, with each supplier running separately. Resource availability is as follows: 400 MHz of processing power, when the CPU is running at one-fourth of the baseline speed, 64 MB of free memory after excluding the memory taken by the operating system and other essential critical systems, and 512 kb/s of bandwidth, provided by an 802.11 wireless access point backed by a DSL line.

The last column of Table V lists the applications used in the experiment.

We measured CPU and physical memory load using Windows Performance Monitor. We used *percent processor time*, *working set* counters of the *Process* performance object to measure CPU and memory utilization, respectively. We took the sampling *average* over a period of time. The performance monitoring API does not provide per process network statistics and so the mechanism for measuring bandwidth demand was different in each case, as explained below.

For a representative clip to watch, we obtained a 2-min trailer of a movie in Windows native .wmv and Real Networks native .rpm in several different bitrates. Where cross-player compatibility is supported, we obtained additional capability points. For example, RealOne plays .wmv format. Also, players provide quality knobs, allowing improved quality in exchange for higher CPU utilization. For example, Windows Media player supports video smoothing that provides higher frame rate than the rate encoded in the stream. For each player, 32 quality points were sampled. To measure bandwidth demand, we consider the bit-rate of the stream and cross-check with the application-reported value. The CPU consumption of different players are widely different for the same quality point.

We measured the CPU and memory used while typing and formatting text for 2 min with each text editor. The memory consumption of the text editors is widely different.

All browsers surveyed support a text-only mode, providing two points in the page richness dimension. To obtain different levels of latency, we used a bandwidth-limiting http proxy and pointed the browser to the proxy. We measured latency by allowing the following bandwidth limits: 28, 33, 56, 128, 256, 512 kb/s. Our script included a sequence of approximately 15 pages with a mix of both text and graphics on the internet. By starting with a clean browser cache, we sampled 16 quality points. We found that the browsers have very similar resource consumption patterns.

Although we realize that the methods for obtaining resource consumption measures are not precise, we believe that they yield good enough approximations for this feasibility analysis.

Note that the capability space of a configuration of suppliers has approximately 500 points $(32 * 16 * 1)$, based on the samples taken. Thirty configurations together provide a capability space of approximately 15 000 points.

2) Prototype Evaluation: The algorithm is guaranteed to find an optimal assignment of suppliers. Furthermore, it will obtain the optimal set of quality points for the

TABLE VI
SUMMARY OF THE EXPERIMENT RESULTS

Configuration latency (average)	Percent CPU used during long-term utilization	Virtual memory used
531 ms	3.8%	8 MB

suppliers, as long as Q-RAM finds the optimal point inside each quadrant. Whenever Q-RAM returns a near-optimal answer, our algorithm will return a near-optimal set of quality points.

Additionally, we evaluated a prototype implementation of the algorithm according to two metrics: 1) latency and 2) system overhead. Latency measures the time it takes to compute an optimal configuration, from the time that a client program requests it. Overhead measures percent CPU and memory utilization of the algorithm. To adapt the configuration in response to environment changes, it is necessary to run the algorithm periodically. Thus, the overhead of the periodic invocation provides a useful metric.

The latency of computing the best configuration averaged over ten trials was 531 ms. In the query and generate phases, the algorithm spends less then 10 ms each. In the explore phase, it is slightly less than 500 ms (approximately 10 ms was due to parsing the request and formatting the answer). The bulk of the time in the explore phase was due to external process invocation and file input–output (Q-RAM package is an external executable). Thus, the latency can be significantly reduced by linking into Q-RAM in-process.

We invoke the algorithm a total of 50 times in 5-s intervals over a period of 250 s and measure *average CPU utilization*. Average CPU utilization is 3.8%. This overhead is fairly low and can be further lowered by running the algorithm less frequently, e.g., once per 10 or 25 s.

Memory usage of the process running the algorithm is approximately 8.8 MB. While this is a significant overhead, most of it is due to the Java virtual machine. Table VI summarizes the key results of the experiments.

B. Lessons Learned and Design Guidelines for Applications

The form of self-adaptation that we address in this work is targeted at support for everyday computing in ubiquitous computing environments. An essential component of this work is to integrate applications into the infrastructure. However, using existing applications is a challenge since these applications are in general not designed for self-configuring capabilities, such as those that we are attempting to provide.

To integrate legacy applications as suppliers of Aura, we have written wrappers around the applications. These wrappers mediate communication with Aura so that the application state can be set and retrieved and the resource usage and QoS can be monitored. Our experience in writing over a dozen suppliers for applications in both Windows and Linux environments has proven that it is easy to implement wrappers to get basic set/get state functionality. However, it is much more challenging to control application adaptation policies.

In order to facilitate smooth integration of applications into the infrastructure, we have identified the following two groups of desirable requirements.

1) To support mobility and coarse-grained adaptation, we require applications to provide mechanisms to get and set the task-level state of each application.
2) To support fine-grained QoS adaptation, we require applications to report aggregate resource usage and QoS information and to provide mechanism to restrict usage of certain resources.

1) Mobility: As described in Section III-B, the task layer requires that the user-level state of applications be retrieved and set. This facilitates task transfer between environments and enables task suspend and resume by a user. Our approach requires some consensus about the meaning of the state of a particular generic service, such as text editing. However, not all applications need to handle all the details of the task state: certain additional properties can be treated as optional. If a supplier cannot interpret these properties, they are simply ignored, but preserved for future instantiations of the task.

In our experience with developing suppliers, we have had mixed success with getting state information from applications. While more recent applications allow reflective access to get and set this information through programmatic interfaces such as .NET, it is not as easy with older applications. Even when applications provide a programmatic interface, it is possible that they do not expose the required information. For example, to restore the state of web browsing, it is desirable to set and get the history of the browser so that backward and forward browsing state can be maintained. Internet Explorer, while providing an interface for setting the current web page, does not provide these additional APIs.

We argue that our requirements for a programmatic interface to set and get the state of the task are not unreasonable. Applications increasingly allow access to such information. In our experience, it has always been possible to get and set some form of the state; the challenge has been in the varying mechanisms that we have had to use, and the issue has been the scope of the information that we have access to.

Our experience has also demonstrated the need for applications to "sandbox" the set of materials belonging to one task. For example, suppose that one user task requires two spreadsheets to be edited, while another task requires one spreadsheet to be edited. If these tasks are simultaneously active, the originating task of each spreadsheet needs to be recorded. Applications provide varying support for such sandboxing. Microsoft Excel supports directly such functionality because there can be multiple physical instances of the Excel process running, each with its own set of files. On the other hand, Microsoft PowerPoint makes this difficult, because only one process instance can be running on a given workstation. This makes the design of the supplier wrapper much more complex and time-consuming.

2) Adaptation: To allow for adaptation to changing resources, in Section IV-B we described a formalism that can provide optimal configurations or reconfigurations based on the available supply of resources and the expected resource usage

of the applications. For the mathematical formalism to work in practice, the Aura infrastructure requires information about resource usage and QoS of applications, resource supply in the environment, as well as a certain level of cooperation from applications about expected resource usage.

In practice, the following set of requirements need to be satisfied in order for the mathematical model of Aura to produce accurate outcomes: 1) ability to monitor application-provided QoS; 2) ability to monitor and report application resource usage; 3) ability to monitor available resource supply; and 4) ability to enforce resource usage limits on applications.

Let us discuss how each of these requirements can be translated into design guidelines for application and system developers. Requirement 1) can be satisfied directly by application developers by exposing rich APIs that report the QoS. For example, many of the commercial and open-source video/media players report the richness of the stream in bit rates, the frame update rate of the video, the size of the frame, the color depth, etc. Requirements 2) and 3) can be satisfied by a shared service that is either provided by the operating system or third-party middleware. While modern operating systems generally provide reasonable performance and resource-monitoring hooks, there is room for improvement. However, some resources can be more difficult to account for on a per-process basis (e.g., battery). Notice that there are research systems, such as the Nemesis operating system [19] and Odyssey adaptive platform [24] that specifically provide accurate resource usage and supply estimates.

With respect to requirement 4), we believe that a twofold approach is needed. First, applications can provide various adaptation strategies (e.g., more CPU-intensive video stream decoding or less CPU-intensive decoding); we believe that applications should provide the ability to comply with resource usage limitations. Some of the video players on the market provide such ability directly, e.g., with respect to network bandwidth. However, it is also desirable to have an operating system-provided mechanism for ensuring that resource limitations are enforced if an application proves to be uncooperative. For example, some video players aggressively prefetch, thereby using all available bandwidth despite being told to use a low bit rate stream. In such cases, a mechanism external to application (e.g., bandwidth throttling) can enforce resource limitations imposed on applications.[3]

VI. Conclusion and Future Work

In this paper we have described an approach to self-configuring capabilities for everyday computing environments. Motivated by the challenges of supporting heterogeneity, resource variability, mobility, ubiquity, and task-specific user requirements, we have developed a self-adaptation infrastructure that has three distinctive features. It allows explicit representation of user tasks, including preferences and service qualities. It provides an EM capability to translate user-oriented task and preference specifications into resource allocations that match the intended environment. Finally, it provides a formal basis for understanding the resource allocation and derived algorithms that support optimal allocation at runtime.

While providing a good starting point, this work also suggests a number of important future directions. First is the extension of task specification so that it can express richer notions of task, such as work flow, cognitive models, and goal-driven task realization. Second is the extension of resource allocation algorithms to take advantage of future predictions. This entails much richer notions of utility, such as those prescribed by options theory. Finally, there are many directions that one could pursue in the area of user interface design to make it even easier for users to create and reuse task descriptions.

[3]Notice that we are not advocating here the need for applications to interfere with the low level scheduling of resources by the operating system. We simply advocate that on the level timescale of seconds the resource usage by applications should be consistent with the expected quality of service delivered.

References

[1] R. K. Balan, J. P. Sousa, and M. Satyanarayanan, "Meeting the software engineering challenges of adaptive mobile applications," Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-03-11, 2003.

[2] C. Boutilier *et al.*, "Cooperative negotiation in autonomic systems using incremental utility elicitation," in *Proc. 19th Annu. Conf. Uncertainty in Artificial Intelligence (UAI-03)*, Acapulco, 2003, pp. 89–97.

[3] L. Capra, W. Emmerich, and C. Mascolo, "A micro-economic approach to conflict resolution in mobile computing," in *Proc. Foundations of Software Eng. (ACM SIGSOFT/FSE)*, 2002, pp. 31–40.

[4] ——, "Reflective middleware solutions for context-aware applications," in *Proc. Int. Conf. Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION)*, New York: Springer-Verlag, 2001, Lecture Notes in Computer Science.

[5] S. W. Cheng *et al.*, "Software Architecture-based adaptation for pervasive systems," in *Proc. Int. Conf. Architecture of Computing Systems: Trends in Network and Pervasive Computing*, Lecture Notes in Computer Science, vol. 299, H. Schmeck, T. Ungerer, and L. Wolf, Eds. New York: Springer-Verlag, 2002.

[6] F. Cristian, "Understanding fault-tolerant distributed systems," in *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.

[7] The DAML Services Coalition (multiple authors), "DAML-S: Web service description for the semantic Web," in *Proc. Int. Semantic Web Conf. (ISWC)*, published as I. Horrocks and J. Hendler (Eds.), Berlin, Germany: Springer-Verlag, Lecture Notes in Computer Science, vol. 2342, pp. 348–363.

[8] J. Flinn *et al.*, "Reducing the energy usage of office applications," in *Proc. IFIP/ACM Int. Conf. Distributed Systems Platforms*, Middleware, 2001.

[9] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, "Project Aura: Towards distraction-free pervasive computing," *IEEE Pervasive Comput.*, vol. 21, no. 2, Apr.–Jun. 2002.

[10] D. Garlan, J. Kramer, and A. Wolf, Eds., *Proc. ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, Charleston, SC, Nov. 18–19, 2002.

[11] D. Garlan, S. W. Cheng, and B. Schmerl, "Increasing system dependability through architecture-based self-repair," in *Architecting Dependable Systems*, R. Lemos, C. Gacek, and A. Romanovsky, Eds. New York: Springer-Verlag, 2003.

[12] I. Georgiadis, J. Magee, and J. Kramer, "Self-organising software architectures for distributed systems," presented at the ACM SIGSOFT Workshop on Self-Healing Sys. (WOSS'02), Nov. 2002.

[13] M. Hiltunen and R. Schlichting, "Adaptive distributed and fault-tolerant systems," *Int. J. Comput. Syst. Sci. Eng.*, vol. 11, no. 5, pp. 125–133, Sep. 1996.

[14] Jini (2003, Sep.) [Online]. Available: www.jini.org

[15] M. Jones, D. Rosu, and M. Rosu, "CPU reservations and time constraints: Efficient, predictable scheduling of independent activities," presented at the ACM Symp. Operating Syst. Principles (SOSP), 1997.

[16] J. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Comput.*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[17] E. de Lara, D. S. Wallach, and W. Zwaenepoel, "Puppeteer: Component-based adaptation for mobile computing," presented at the 3rd USENIX Symp. Internet Technologies and Systems (USITS), 2001.

[18] C. Lee *et al.*, "A scalable solution to the multi-resource QoS problem," presented at the IEEE Real-Time Syst. Symp. (RTSS), 1999.

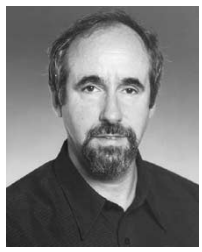
[19] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," *IEEE J. Selected Areas Commun.*, vol. 14, no. 7, pp. 1280–1297, Sep. 1996.
[20] J. Kephart and M. Parashar, Eds., in *Proc. Int. Conf. Autonomic Computing*, New York: IEEE Press, May 17–18, 2004.
[21] F. Kon *et al.*, "Dynamic resource management and automatic configuration of distributed component systems," presented at the USENIX Conf. OO Technologies and Systems (COOTS), 2001.
[22] D. Narayanan, J. Flinn, and M. Satyanarayanan, "Using history to improve mobile application adaptation," presented at the 3rd IEEE Workshop Mobile Computing Systems and Applications (WMCSA), 2000.
[23] R. Neugebauer and D. McAuley, "Congestion prices as feedback signals: An approach to QoS management," presented at the ACM SIGOPS European Workshop, 2000.
[24] B. Noble *et al.*, "Agile application-aware adaptation for mobility," *Oper. Syst. Rev.*, vol. 31, no. 5, pp. 276–287, Oct. 1997.
[25] D. Pisinger, "An exact algorithm for large multiple knapsack problems," *Eur. J. Oper. Res.*, vol. 114, pp. 528–541, 1999.
[26] Service Location Protocol (2003, Sep.). [Online]. Available: www.srvloc.org
[27] J. P. Sousa and D. Garlan, "The Aura software architecture: An infrastructure for ubiquitous computing," Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-03-183, 2003.
[28] ——, "Beyond desktop management: Scaling task management in space and time," Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU-CS-04-160, 2004.

João Pedro Sousa received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 2005.

He is a Postdoctoral Fellow, Carnegie Mellon University. His research interests include software architecture, self-configuring systems, and human factors for ubiquitous computing environments.

Vahe Poladian received the B.S. degree in computer science and mathematics from the Macalester College, Saint Paul, MN.

He is a Doctoral Candidate, Carnegie Mellon University, Pittsburgh, PA. His research interests include applications of microeconomics, utility theory, and decision theory to automatic software configuration on mobile computing platforms.



David Garlan (S'82–M'85) received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1987.

He is a Professor of Computer Science, Carnegie Mellon University. His research interests include software architecture, formal methods, self-healing systems, and task-based computing.

Bradley Schmerl (S'96–M'03) received the Ph.D. degree in computer science from Flinders University, Adelaide, South Australia, in 1997.

He is a Systems Scientist, Carnegie Mellon University, Pittsburgh, PA. His research interests include dynamic adaptation, software architectures, and software engineering environments.

Mary Shaw (M'78–SM'81–F'90) received the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1972.

She is the Alan J. Perlis Professor of Computer Science, Carnegie Mellon University. Her research interests include value-based software engineering, everyday software, software engineering research paradigms, and software architecture.