Beyond Desktop Management: Scaling Task Management in Space and Time

João Pedro Sousa, David Garlan

August 2004
CMU-CS-04-160

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

This material is based upon work supported by the National Science Foundation (NSF) under Grant CCR-0205266, and by DARPA under Grant N66001-99-2-8918. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, DARPA, or Carnegie Mellon University.

Keywords: task management, desktop management, ubiquitous computing, software architecture, task-oriented computing, everyday computing, self-configurable systems, adaptive systems, modeling user preferences, utility-based adaptation, resource-adaptive applications.

Abstract

Computers support more and more daily activities for common users, and users increasingly take their activities to different locations. Rather than being bound to a specific device, users would like to take full advantage of the computer systems accessible to them, much like they take advantage of the furniture in each physical space. However, user attention takes a heavy toll when scaling the use of computers to tasks that are constantly interrupted and resumed, and that span many locations and long periods. In this report we describe an infrastructure that provides users with easy access to their tasks as a logical unit, across multiple devices, and over time spans of years. The infrastructure handles platform and application heterogeneity, as well as dynamic adaptation to resource variations. We validate that the infrastructure's overhead is small compared to normal application startup, and that the approach scales.

Acknowledgments

The ideas contained in this technical report owe much to Mahadev Satyanarayanan, Peter Steenkiste, Bonnie John, Brad Myers, and Scott Hudson for asking the right questions. Finding and polishing the answers to those questions emerged out of detailed discussions with Vahe Poladian, and Bradley Schmerl. In addition, a number of others have contributed in various ways to the work underlying this report (in alphabetical order): Yady Guitana, Lalit Jina, Peter Kim, Takahide Matsutsuka, Tadashi Okoshi, Bhuricha Sethanandha, Chris Tuttle, Wei Zhang.

Contents

1 INTRODUCTION **5**

2 RELATED WORK **6**

3 DEFINING USER TASKS AND SUSPEND-RESUME **8**

4 SUPPORTING SUSPEND-RESUME **10**

5 STORING AND FINDING TASKS **13**

6 DEFINING AND EXPLOITING USER PREFERENCES **14**

7 IMPLEMENTATION AND EVALUATION **16**

8 DISCUSSION AND FUTURE WORK **19**

REFERENCES **19**

1 Introduction

It is well known that computer users may simultaneously handle several tasks, such as preparing presentations, writing reports, or answering email, constantly shifting their attention between them. This fact was observed twenty years ago [5], and it certainly holds today [10].

One important property of such tasks is that they typically involve several applications and information resources. For instance, for preparing a presentation, a user may edit the slides, refer to a couple of papers on the topic, check previous related presentations, and browse the web for new developments. Existing work on desktop management has addressed this property, from early work in Rooms [7], through recent work such as the GroupBar [22].

Another, increasingly important property of user tasks is that they may span multiple locations. Advances in ubiquitous computing are prompting people to change their expectations towards the availability of computing [1], and this issue is especially relevant for users working at large company sites, research campuses, or service-oriented facilities such as hospitals. There, users carry out their work across many locations, moving from their office to their colleague's, to common areas, to meeting rooms, etc. Rather than being bound to a specific device, users may want to take full advantage of the computer systems accessible to them, much like they may take advantage of the furniture in each physical space. In the example above, the user may start working on the presentation while in his or her office, move to the office of a collaborator, and pick the task up later at home. Users should not have to carry a personal machine around, although they may, just like people don't have to carry their own chair around. Ideally, users should be able to continue their tasks, on demand, with whatever computing resources are available.

However, realistically, we cannot expect that a mobile user will encounter a uniform set of devices over multiple locations, even among the same class of equipment, say desktops or meeting room equipment. In a wider ubiquitous computing setting, where users may wish to access some tasks at home, or at a coffee shop, or at the airport, the types of applications and devices available to a user will vary widely. Therefore dealing with device and software *heterogeneity* becomes an important issue in addressing the scalability of task management in space.

Yet another important property of user tasks is their duration and recurrence. Users may work on some tasks for days or even months. Tasks may need to be resumed after the user thought they were done. And tasks may recur periodically; or to be more precise, users may periodically carry out distinct instances of the same kind of task. For instance, if a user prepares monthly reports, although such tasks share some characteristics, each has its own identity and may diverge from the common pattern: in July the user needed to include some slides for the top management, but not in August.

Unfortunately, existing desktop management systems lack a notion of task that can be carried across different devices, and that can be referred to independently of a set of active application windows that represent it. Specifically, users should be able to refer to not only currently active tasks, but also the ones defined long ago, such as *the report on xyz that I wrote last year*. And note that this is not a question of finding one file that resulted from the task, but a question of finding the task (definition) itself, so that it can be reactivated, if necessary, or used as a template to create a similar task.

Our research experiments with a notion of user tasks that scales beyond a single desktop, and beyond the set of currently active windows. Specifically, by capturing at a semantic high-level what the user is doing, we can reactivate the task in another machine, where the task was never

activated before. By making such representation independent of specific applications, we can reactivate the task on different platforms. For instance, for the task of preparing a report, we capture the fact that the user needs to *edit a text document*, not that MS Word is part of the task. By using available application interfaces (APIs,) we capture a high-level representation of the user-perceived state: things such as cursor position and application settings, in addition to window layout and files being worked on. By giving tasks a semantic identity (which goes beyond a name,) we enable users to find, manipulate, and resume tasks at will, regardless of how long ago, or where, or in which machine those tasks were defined.

While other research in ubiquitous computing shares the goal of supporting the notion of mobile user tasks, it commonly custom-builds or significantly extends existing applications. In contrast, our approach does not assume, or preclude, that applications are mobile, or distributed, or otherwise built with ubiquitous computing in mind. We focus on mechanisms that are external to existing applications, and that exploit the applications' APIs to support the notion of mobile user tasks. Our approach to representing user tasks gracefully handles the heterogeneity of devices and applications that a mobile user will inevitably encounter.

In this report we present an infrastructure that supports the notion of user *task* as a first class entity. Users can browse their tasks, swap active tasks, as well as suspend working at one location and resume at another location, while fully utilizing the locally available set of devices (whether or not they carry around a personal device). The infrastructure automatically finds and configures devices and applications on the user's behalf, optimizing the support for the relevant tasks. For that, adequate knowledge about the user's preferences plays a key role.

The Aura infrastructure has been developed as part of the Aura Project at Carnegie Mellon University [12], and has been evolving in prototype form for over a year. Ultimately, the goal of this research is to demonstrate that such an infrastructure reduces the distractions incurred by mobile users, allowing them to focus on their tasks rather than on the end-user configuration of the computer systems to support those tasks. In addition to supporting suspend-resume of user tasks across heterogeneous devices, the infrastructure is also prepared to handle (i) dynamic variation of resources, such and bandwidth and battery, and (ii) dynamic variations in the availability of applications and devices.

This report describes how a user may interact with the infrastructure to define a task, to suspend-resume tasks, and to browse his tasks (Sections 3 and 5). It also provides a high-level description of the architecture and workings of the infrastructure (Section 4 – a detailed discussion and formal specification is available in [23].) This report does not address validating the claims of reducing user distractions, of the optimality of task instantiation, or of dynamic adaptation, although Section 6 describes how user preferences are captured and represented, and hints at how they are used to drive both optimal task resume and dynamic adaptation.

In this report, we validate the following claims: (a) it is feasible to implement such an infrastructure, (b) the infrastructure's overhead for automatic suspend-resume is small compared to the average startup time of applications, and (c) the approach scales for large numbers of user tasks and available services.

2 Related Work

Existing desktop managers fall short in scaling in space, or time, or both. Some solutions to address user mobility (scalability in space) rely on thin clients enabling the user to remotely access a remote computing server (e.g. X Windows or PC Anywhere). However, such solutions have two serious drawbacks: first, remote clients rely on a stable, fairly high-bandwidth, connection –

something that is frequently not available in a ubiquitous computing setting. Second, as thin UI-oriented clients, these solutions fail to take advantage of local resources, and, in particular, of the ever-increasing capabilities of mobile devices, smart spaces, etc. A partial solution to this limitation is provided by mobile context-aware applications, which target user mobility by providing a mobile piece of code that follows the user around. That piece of code examines the capabilities available at each location it is migrated to, and using internal logic, chooses appropriate interaction modalities. However, this strategy assumes a certain degree of uniformity in the platforms that the code will be migrated to, and relies on the mobile application's ability to recognize and handle the characteristics of each device and other software components in the system. Therefore there is no guarantee that the application will be able to migrate to every device that the user chooses to utilize. Even if it does migrate, there is no guarantee that the mobile application will provide better service than a local, custom built, application. Most importantly, current solutions based on mobile code are application-centric, and do not support a first-class notion of user task.

Early work in ubiquitous computing environments experimented with the idea that users are mobile, and may utilize available devices in their vicinity. That work uses OS-level mechanisms driven by location-sensing components to automatically "teleport" (make accessible) a user's desktop to the nearest display within a smart space, such as an augmented home [6]. Other early work experimented with the idea that a user's task encompasses a set of applications *independently* of device-driven metaphors, such as a desktop [15,24]. In that work, applications are visually aggregated and can be moved between the foreground and background of the user's attention as a unit. Subsequent work targeted making smart spaces amenable to cooperative tasks [18] and supporting the tasks of mobile users in very specific domains, such as biology labs or hospitals [3,9]. Yet other research extends operating systems for supporting the notion of mobile user tasks, where tasks are user-defined collections of applications [19]. Commonly, the above cited research custom-builds or significantly extends existing applications to work over an infrastructure that supports distributed data exchange and application mobility.

Others have adopted a lightweight approach for suspend-resume, in the sense that no modifications are required to existing applications. There, the state of the whole virtual memory in the user's current machine is captured and migrated to the target machine [14]. However, such a solution is limited to situations where the user tasks are supported by a single machine and where the user only moves among machines with a compatible hardware architecture, and identical system and application software.

In contrast, our approach handles heterogeneity by representing user tasks at a high level and then mapping each task to the applications and devices available at each location. Nevertheless, our approach is still lightweight in the sense that light wrapping is enough to integrate existing applications.

In addition to heterogeneity, another important challenge is coping with dynamic change. Here we build on research on model-based dynamic reconfiguration of software systems [8], and smoothly integrate results on fidelity-aware applications [16]. Over the latter, our work adds the advantage that the resource-adaptation policies can be dynamically tuned to match the user's preferences for each task.

Other results our research builds on, although they are not covered in this report, are service ontology and discovery mechanisms [2,11], mobile information access [20], and sensing the physical context around the user [13,21].

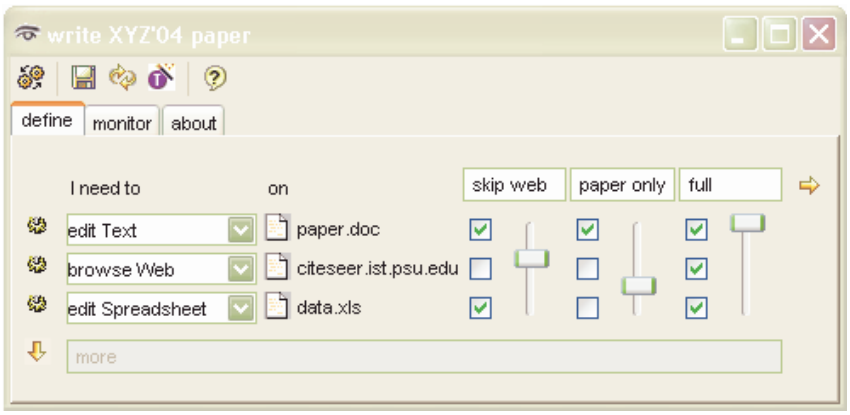

Fig. 1. Fred's task definition for writing XYZ'04 paper

3 Defining User Tasks and Suspend-Resume

To illustrate the lifecycle of tasks, we present a simple scenario of a user, Fred, who is about to write a paper. Fred considers opening the relevant files and applications on a need-to basis, using standard OS mechanisms. However, since this task will persist for the next few weeks, Fred decides to define it with the infrastructure (Fig. 1). Initially, Fred includes only editing the paper, and he does that by pressing the down arrow at the bottom of the (empty) task definition window and selecting *edit text*. The text editor activated by the infrastructure brings up a (default) blank document and Fred starts working. As Fred browses the web, he decides to associate an especially relevant page with the task, so that it is brought up automatically every time the task is resumed. For that, 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 entering the performance data on a spreadsheet. Again, Fred simply drags the file produced by the data gathering tool, from the file system explorer into the *more* field and selects *edit spreadsheet* for it.

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.

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

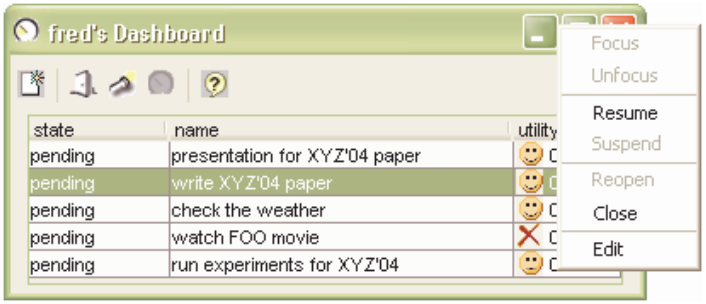

Fig. 2. Fred's list of pending tasks on the dashboard

Suppose now that Fred enters a new location, such as a collaboration area, and authenticates with the infrastructure.[1] A dashboard becomes available to Fred, showing all of his tasks that are both pending and enabled (Fig. 2, and more on this below). The infrastructure actively monitors the availability of devices, applications and resources in Fred's vicinity and matches that information against the pending task definitions, such as the one in Fig. 1, and against Fred's preferences (see below and Section 6). How well each of these tasks can be supported in the current circumstances is represented as the *utility* of the computing environment around Fred for the particular task. In simple terms, the higher the utility of the environment, the better the match between Fred's needs for the task and the capabilities of the environment. The infrastructure evaluates the utility for each alternative operating mode, weighs it with Fred's preference as expressed in the happiness slide bar under each configuration, and presents Fred with the best option on the dashboard. The *monitor* tab in Fig. 1 presents details on the alternatives for supporting each configuration, and the corresponding utility. The user may decide to activate a configuration other than the one with the best calculated utility, thus overriding the dashboard's default. Because of space limitations, these features are not shown here.

By choosing an option in the popup menu associated with each entry in the dashboard, Fred tells the infrastructure which tasks he wishes to work on (Fig. 2). Tasks change state as a result of Fred's choices. After a task is created it becomes *pending*. The user can associate enabling constraints with a task, constraining it (its appearance in the dashboard) to certain locations, timeframes, or other context properties (Section 5). Note however that *enabled* is not a state, but rather a selection (a subset) of pending tasks determined by the user's context. A pending task becomes *active* after the user decides to resume it. The user may switch an active task between the *foreground* and the *background* of his attention by selecting the focus/unfocus actions. An active task can be suspended, returning to the pending state, and then closed, when the user doesn't intend to work on the task any more. Closed tasks by default do not show on the dashboard but can be browsed and (re)opened, if necessary (Section 5). The state transition diagram for tasks is shown in Fig. 3, where the arc labels correspond to the first letter of the actions described above.

When the user resumes a task, the infrastructure takes the configuration with the best utility and activates applications that support the task's activities, such as those in Fig. 1. Furthermore, the infrastructure uses a task snapshot captured the last time that the task was suspended to direct the applications to recover user-level settings such as cursors, window size and placement, application options, etc. The focus and unfocus actions, provide a lightweight mechanism for swapping

[1] Authentication mechanisms can range from smart id tags, to fingerprint scanning, to typing in a user name and password. For the purposes of this discussion, the result is equivalent.

among active tasks without deactivating the applications. For instance, applications with a GUI may react to an unfocus directive by minimizing their windows, data streaming servers may react by not streaming data (without closing the connection,) etc.

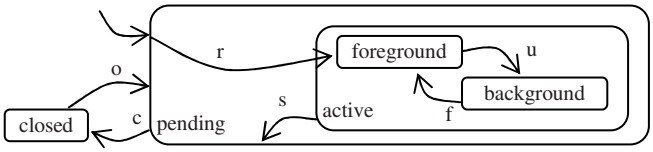

Fig. 3. State transition diagram for a task

4 Supporting suspend-resume

This section presents a layered view of the infrastructure and describes the roles of each layer with respect to task suspend-resume and dynamic adaptation. Table 1 summarizes the relevant terminology.

Table 1. 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	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-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.

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

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. For example, the features in Figures 1 and 2 reside in the Task Management layer.

Table 2 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 accesses a new environment, TM coordinates access to all the information related to the user's task, and negotiates task support with Environment Management (EM). Task Management 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 instantiates the resumed task, as appropriate. Task Management may also capture complex representations of user tasks (out of the scope of this report,) 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).

Table 2. 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
Environment	support the user's task	• monitor relevant resources • fine grain management of QoS/resource tradeoffs

The EM layer holds abstract models of the environment. These models provide a level of indirection between the user's needs, expressed in environment-independent terms, and the concrete capabilities of each environment.

This indirection is used to address 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 the ones available in the environment. With respect to dynamic change, the existence of explicit models of the capabilities in the environment enables automatic reasoning upon dynamic changes in those capabilities. Environment Management adjusts such a mapping automatically in response not only to changes in the user's needs (adaptation initiated by TM, see Section 3), but also to changes in the environment's capabilities and resources (adaptation initiated by EM). In either case, adaptation is guided by the maximization of a *utility function* representing the user's preferences (see Section 6).

The Environment layer holds the applications and devices that can be configured to support the user's task. Configuration issues aside, these suppliers 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 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. That knowl-

edge is very hard to obtain at the application level, but once it is determined at the user level – by Task Management – it can easily be communicated to the applications selected to support the user's task (see [4]).

The infrastructure can accommodate suppliers with a wide range of sophistication in matters like fidelity- and context-awareness. For communication between layers, tagged formats have the advantage over raw data formats that they make it easier to deal with heterogeneity. Specifically, tagged descriptions can be processed by suppliers with different degrees of sophistication. For example, suppose the user requires a *text editing* service, and would prefer spell checking to be activated. Although finding a suitable supplier in a rich environment may not be a problem, a basic text editor on a small platform might not support spell checking, or even be aware of what "spell checking" means. Therefore, the description of the user-level state must be such that a given supplier is able to extract the information it can recognize, without being thrown off by information it does not know how to handle. Naturally, the layers of the infrastructure need to share a vocabulary of tags, or otherwise be able to resolve symbol equivalences (this is a topic of other research [11]).

The same reasons that make a tagged format desirable to represent the user-level state apply to the overall task description. Given the distributed, heterogeneous, and dynamic nature of ubiquitous computing environments, any component in the infrastructure may have to deal with different versions of some other components at some point. Therefore the overall representation of user tasks, and all communication among the layers in Table 2, is XML-based.

Furthermore, all communication between layers is asynchronous (non-blocking). Typical ubiquitous computing environments are heavily distributed –suppliers, especially, may be scattered across different devices, some of which may be remote to the user's location. Connectivity varies widely, from high-speed wired connections to fluctuating wireless (radio or infrared) connections. In synchronous communication, the originating (calling) component blocks on the reply of the target (called) component. However, in our case, each layer should keep up with its responsibilities in real-time, doing the best it can with the available information, and without blocking on another component's reply. For example, EM should not stop monitoring the capabilities of the environment, or replying to TM's requests, on account of being blocked on the reply of a remote supplier – which might have become disconnected. Likewise, TM should not stop responding to changes in the user's task, when waiting for the reply of some other component.

Each layer reacts to changes in user tasks and in the environment at a different granularity and time-scale. Task Management acts at a human perceived time-scale (minutes), evaluating the adequacy of sets of services to support the user's task. Environment Management acts at a time scale of a few 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. A detailed description of the architecture, including the formal specification of the interactions between the several components in the three layers defined above, is available in [23].

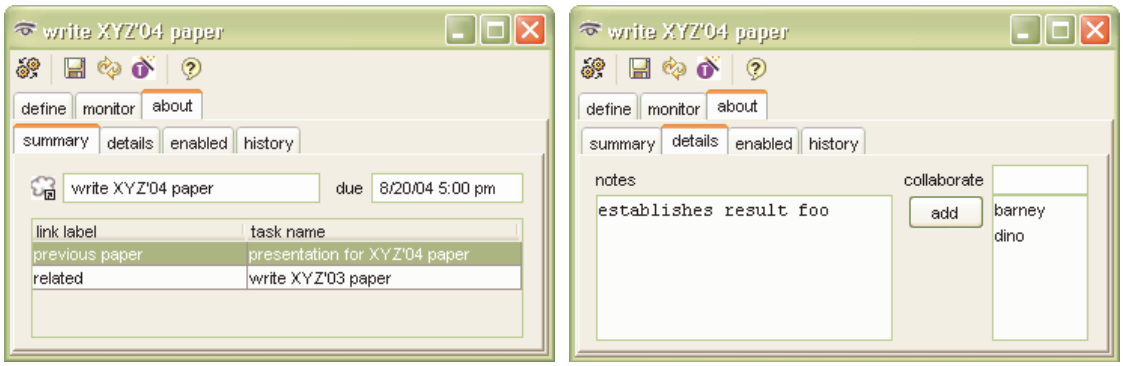

Fig. 4. Information about Fred's task of writing the XYZ'04 paper

5 Storing and Finding Tasks

Fully supporting the notion of user task entails enabling the user to browse and refer to his tasks as first class entities in the system. For that, the infrastructure stores information *about* each task, such as name, due date, relevant notes, or names of people collaborating on the task (Fig. 4). Note that this information is stored in addition to the user-level state of a task, as defined in Section 4.

Storing information about a task enables the user to search for that task later on, based on *anything* he remembers about that task. One can think of each term entered in the information about the task as enabling one classification scheme, which distinguishes all the tasks referring to that term from those that don't. This is in stark contrast with the single hierarchical classification scheme offered by the directory structure in file systems, and much closer to the approach used by web search engines (more on this below).

The user is free to enter as much or as little information about the task as he feels appropriate, with no concerns about name uniqueness. In the extreme, even the name may be omitted, since the infrastructure keeps an internal unique id for each task of a given user. Of course, the more information the user provides, the easier it will be for him to find that task later. The user can establish links between a task *t* and related tasks by dropping a task reference into the links table on *t*'s summary tab.

A task's reference can be obtained anywhere the task is shown: in the entries on the dashboard (Fig. 2) or on the browsing results (Fig. 5), from the arrow to the left of the name in the summary tab, in other task's links table, etc. Task links can be followed by double clicking, which shows the corresponding task information window. Most importantly, task references can be dropped onto the dashboard, thus enabling the user to act upon them as described in Section 3.

The infrastructure's browsing component, *lamp*, builds an index of the terms entered anywhere in the task information window (Fig. 5). During a search, this index is matched (formally an inner product) against the index of searched keywords. Dates are matched by the *before* and *after* criteria: for instance, *10/12/03* matches *before 1/1/04*. Each match scores one point, and search results are presented sorted by score.

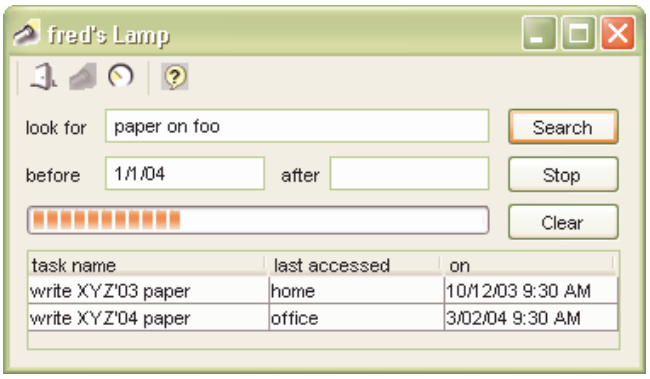

Fig. 5. Search for papers on *foo* written before 1/1/04

Note two tabs in Fig. 4, *history* and *enabled*, not shown for the sake of space. The history tab shows the list of dates and places where the task was activated (in the sense discussed in Section 3). This information is kept for auditing purposes, in addition to the search purposes discussed above. The enabled tab lists the context properties, such as locations and timeframes, under which the task is enabled. Recall that only enabled (pending) tasks show in the dashboard after a user authenticates. Nevertheless, the user may browse disabled (or closed) tasks and explicitly drop them onto the dashboard. The infrastructure allows any task in the dashboard to be activated, logging that fact in the task's history.

6 Defining and Exploiting User Preferences

Suppose that Fred needs to prepare a review of a promotional video. For taking notes on the video, the Fred prefers to dictate the text. However, if the environment lacks the capabilities (microphone, speech recognition software…) or resources (CPU cycles, battery charge…) to support dictation satisfactorily, Fred is willing to type or write the text. These are examples of *configuration preferences* (see also Section 3).

For typing the notes (*text editing* service), Fred may prefer *MSWord* over *Emacs*, and be unwilling to use the *vi* editor at all. These are examples 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.

Consider now that Fred will be watching the video over a network link. Suppose that the bandwidth suddenly drops: should the video player reduce image quality or frame-update rate? The answer depends on Fred's *QoS preferences* for the current task. For watching a video with a lot of motion, such as a sports video, Fred may prefer to preserve the frame-update rate at the expense of frame quality. However, for watching a painting documentary, Fred may prefer image quality to be preserved at the expense of frame-update rate. As another example of a QoS tradeoff, when using speech recognition with limited resources, would Fred prefer more accurate recognition or snappy response times?

Computing the best match between what a user needs for a given task and what the environment has to offer corresponds to maximizing a utility function. The utility functions used in our work express formally, in a computable way, the user's preferences for the task. The environment's capabilities and available resources act as constraints in the maximization process. User preferences (and their formal reification, utility functions) used in our work have three parts: first, *con-*

figuration 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 Quality of Service (QoS) levels and preferred tradeoffs.

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, 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. 6). Sigmoids are easily encoded by just two points: the values corresponding to the knees of the curve; that is, the limits *good* of the good-enough interval, and *bad* of the insufficient interval. The case of when more-is-better (e.g. *accuracy*) is just as easily captured as the case where less-is-better (e.g. *latency*) by flipping the order of the *good* and *bad* values. In the case studies evaluated so far, we have found the expressiveness of the forms above to be sufficient.

The *utility space* provides a formal representation of how useful is each aspect of the preferences, and ultimately of the whole environment, relative to a specific task. In other words, utility is a measure of user's happiness with respect to possible outcomes. Formally, we encode utility in the interval [0,1] of the real numbers, where 0 utility 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 usefulness for the specific task.

Fig. 6 shows an example of QoS preferences for the speech recognition service. The service has three QoS dimensions: *latency*, *accuracy* and *vocabulary* size. The first two are numeric: the latency of recognizing each utterance is expressed in seconds, and accuracy reflects the percentage of words that are recognized accurately. 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 don't expect every user to handle this kind of detail. Rather, the infrastructure provides a set of templates for each service type, corresponding to frequent situations. For instance, for the speech recognition service, it includes the *snappy recognition* template shown in Fig. 6, as well as the *accurate recognition* template, where the latency thresholds are relaxed, and the accuracy and vocabulary more strict. The user can choose which preference template to apply to each service when defining a task (Fig. 1) or, by selecting the advanced tuning, manipulate the preferences directly.

---

[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.

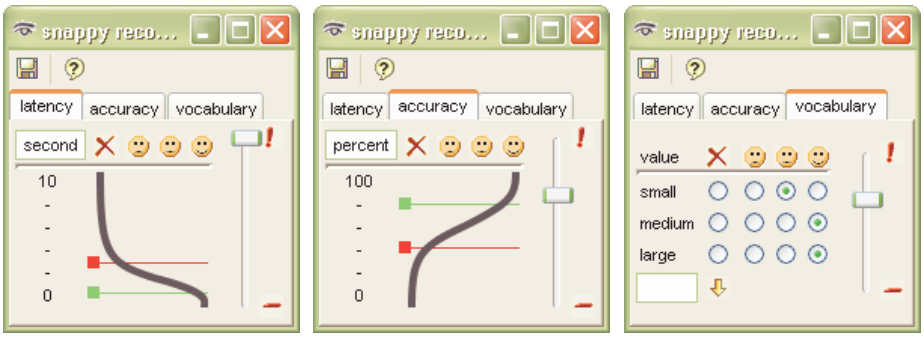

Fig. 6. QoS preferences for the *speech recognition* service

The framework for representing user preferences described above is used to find the optimal initial configuration, as well as to address the ongoing reconfiguration of the environment, dynamically optimizing the support for the user's tasks. Fig. 7 shows the internal representation of the preferences captured in Fig. 6. Note that the infrastructure creates user interfaces like the one in Fig. 6 dynamically, based on the internal representation. See [23] for a complete description of the formats of user preferences and corresponding formal semantics. A description and evaluation of the maximization algorithm used by the infrastructure is available in [17]. This algorithm exploits the structure of the utility functions to achieve an aggressive, but provably correct, pruning of the search space, resulting in an expected complexity orders of magnitude smaller than standard constrained maximization heuristics.[3]

```
<utility combine="product">
        <QoSdimension name="latency" type="float">
                <function type="sigmoid" weight="1">
                        <thresholds good="1" bad="3" unit="second"/>
                </function>
        </QoSdimension>
        <QoSdimension name="accuracy" type="float">
                <function type="sigmoid" weight="0.7">
                        <thresholds good="90" bad="40" unit="percent"/>
                </function>
        </QoSdimension>
        <QoSdimension name="vocabulary" type="enum">
                <function type="table" weight="0.7">
                        <entry x="small" f_x="0.8"/>
                        <entry x="medium" f_x="1"/>
                        <entry x="large" f_x="1"/>
                </function>
        </QoSdimension>
</utility>
```

Fig. 7. Internal representation of the QoS preferences in Fig. 6

7 Implementation and Evaluation

The current version of the infrastructure implements the Task Management (TM) and Environment Management (EM) layers (see Section 4) in Java. For the Environment layer, we have implemented suppliers that wrap Internet Explorer, MSWord, GNU Emacs, Media Player, Xanim (media player), PowerPoint, Sphinx (speech recognizer), Festival (speech synthesizer), and BabelFish (web-based translator). 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.

[3] The worse case complexity is the same as the complexity of standard heuristics.

The effort for developing a new supplier is about 2 weeks time-on-task for an experienced student for basic capture and recovery of user-level state: for example, for a web browser, the navigation history, current page, window position, size and scroll, etc. In our experience, controlling the resource adaptation policies of an application proved to be more challenging. These applications tend to fall into two categories: first, those coming from research or open-source projects, for which controlling the policies, although possible, can be an involved task. Second, commercial software, which either doesn't expose APIs to control the adaptation policies, or for which we could not observe a reliable correlation between the controls transmitted to the application and its actual behavior – consistently greedy. However, the application market seems to be maturing in this respect: in recent experiments with RealOne Player we could observe a good correlation between the control knobs for the resource-adaptation polices and the application's actual behavior.

For practicality, the current implementation relies on the following assumptions:

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

We have tested the infrastructure on Windows and Linux platforms, including the migration of user tasks between the two.[4] The results below were obtained on a IBM ThinkPad 30 laptop running Windows XP Professional, with 512 MB of RAM, 1.6 GHz CPU, and WaveLAN 802.11b card. The TM and EM each run on a Hot Spot JRE from Sun Microsystems, version 1.4.0_03.

To test the scalability of the infrastructure with the number of tasks, we populated a large number of task summaries (see Fig. 4) using data extracted from random text documents. Since we expect the number of tasks for the average user to range in the hundreds of new definitions per year of usage, we went up to about 10,000 task definitions. We then repeatedly divided the task directory size in half to obtain the variation of the performance with the number tasks.

Fig. 8 shows the latency between authentication and the availability of the user's dashboard containing an up-to-date list of the pending tasks. The diamond-shaped points correspond to the latency in reading the user's task directory, currently implemented over the file system, and the square-shaped points to the latency of searching the pending tasks after the directory was read. As expected, the latency grows linearly with the number of tasks, being under 1 second for well over 2000 task definitions. Fig. 9 shows the latency of task browsing as well as the memory footprint of the TM. As expected, both grow linearly with the number of tasks, at least after a significant number of tasks. The current implementation keeps the task directory in memory, after it has been read after authentication. Of course, the penalty in memory footprint is compensated by the swift search times: less than 1 second for a search such as the one illustrated in Fig. 5, even against 10,000 task definitions.

---

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

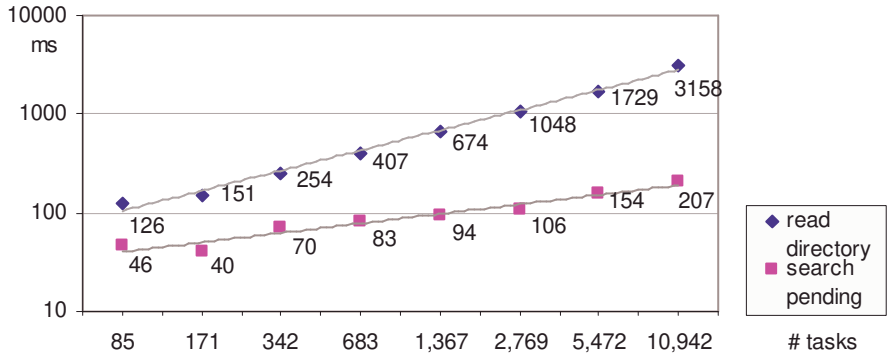

Fig. 8. Latency of dashboard availability after user authentication

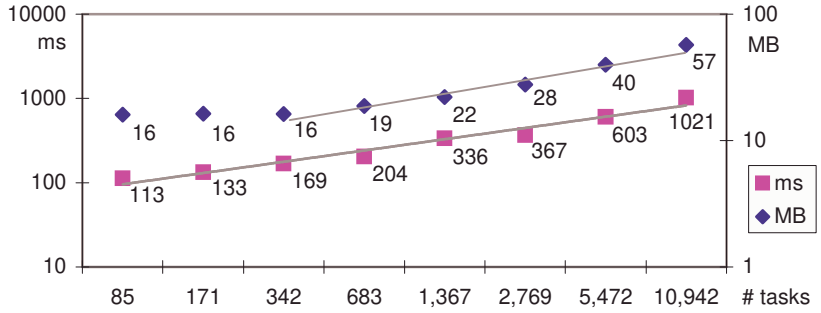

Fig. 9. Latency of task searching and TM memory footprint

Should the memory footprint become an issue, for instance when deploying the infrastructure on a handheld computer, the task directory can be read for every search. The memory footprint of the TM would drop to 16 MB, and the latency of each search would be increased by the latency of reading the directory (Fig. 8).

The memory footprint of the EM ranges linearly from 7 MB to 15 MB when it caches the descriptions of 20 up to 400 services in the environment. Note that a "hello world" Java application has a memory footprint of 4.5 MB, and that a Java/Swing application that shows a "hello world" dialog box has a memory footprint of 12 MB.

Once a task is included in the dashboard, the infrastructure takes an average of 200 ms (standard deviation 50 ms) to find the best configuration. Recall that this involves the constrained maximization of the utility function for each of the alternative configurations (see example in Fig. 1). These numbers were obtained from tasks that range from 4 up to 24 alternative combinations of suppliers for the required services. The performance variation is due to the different numbers of alternative configurations, number of services in the task, and QoS profile of the suppliers.

Once the user decides to resume a task in the dashboard, the infrastructure takes an average of 700 ms (standard deviation 200 ms) to confirmedly activate all the relevant suppliers. Once the user decides to suspend an active task, the infrastructure takes an average of 170 ms (standard deviation 20 ms) to obtain a snapshot of the user-level state in all relevant suppliers.

8 Discussion and Future Work

In this report we presented an infrastructure that supports the notion of user *task* as a first class entity. This infrastructure has a number of important benefits:

- Enables mobile users to browse their tasks and swiftly resume their work on a previous interrupted task, regardless of when and where that task was interrupted.
- Supports the description of alternative ways of supporting the same task, such as taking notes either by dictation or by editing a text document, and assists the user in choosing the one for which the currently available devices and applications offer the best support.
- Handles heterogeneity by describing user tasks in terms of the required *services*, such as *editing slides*, *viewing a text document*, or *browsing the web*, rather than in terms of particular applications.[5]
- Easily accommodates legacy applications by wrapping, taking advantage of the increasingly rich APIs for controlling the applications' behavior and for capturing its user-level state, such as settings and open files.
- Optimizes resource allocation across all the applications involved in a user's task, taking into account the relative importance of each application within the task.
- Actively monitors the QoS that applications provide to the user.
- Smoothly integrates work in fidelity-aware applications by handling dynamic change at two levels: (a) at a time scale of every few seconds it optimizes the choice of applications and the task-wide resource allocation. And (b), it passes user preferences (in the form of a utility function) to fidelity-aware applications, which then can micromanage resource utilization at a time scale of milliseconds.
- Enables fidelity-aware applications to enforce the resource-adaptation polices and QoS tradeoffs that are appropriate for each task.

Ultimately, the goal of the ongoing research at Carnegie Mellon University is to demonstrate that such an infrastructure reduces the distractions incurred by mobile users, allowing them to focus on their tasks rather than on the end-user configuration of computer systems. Such claim needs to be validated both with respect to the instantiation of tasks, and with respect to the dynamic adaptation to variations in the capabilities and resources accessible to the user. Although such claim is not yet validated at the present date, the results obtained so far are encouraging. Our implementation demonstrates that it is feasible to obtain the benefits listed above (see [4,17,23]).

We demonstrated that the infrastructure's overhead for automatically configuring the environment on the user's behalf ranges on the few hundreds of milliseconds. This kind of overhead is mostly imperceptible when coupled with starting up applications. What the user clearly perceives is that applications instantaneously recover the user-level state where the task was previously interrupted, and that all services associated with a task start up as a unit. Future validation will demonstrate that such benefit quickly amortizes the cost of defining the task's constituents and user preferences.

We also demonstrated that the infrastructure scales well with the number of task definitions, and with the number of services in the environment.

References

1 Abowd, G., Mynatt, E.: Charting Past, Present and Future Research in Ubiquitous Computing. In: ACM Transactions on Computer-Human Interaction, Vol. 7(1) (2000) 29-58

2 Arnold, K., O'Sullivan, B., Scheifler, R., Waldo, J., Wollrath, A.: The Jini Specification. Addison-Wesley, 1999.

3 Arnstein, L., Sigurdsson, S., Franza, R.: Ubiquitous Computing in the Biology Laboratory, Journal of Lab Automation (JALA). 6(1), March 2001.

[5] This approach is not incompatible with taking advantage of mobile applications, but it builds on that without limiting the user to one application, no matter how mobile or pliable the application.

4 Balan, R., Sousa, J.P., Satyanarayanan, M. Meeting the Software Engineering Challenges of Adaptive Mobile Applications. Carnegie Mellon U. Tech. Report, CMU-CS-03-11, 2003.
5 Bannon, L., Cypher, A., Greenspan, S., Monty, M. Evaluation and analysis of user's activity organization. Proceedings of CHI'83, pp 54-57, ACM, New York, 1983.
6 Brumitt, B. et al.: EasyLiving: Technologies for Intelligent Environments. Proc HUC2000 – 2nd Int Symposium on Handheld and Ubiquitous Computing. LNCS 1927 pp 12-29, Springer-Verlag, Gellersen, Thomas (eds.), September 2000.
7 Card, S.K. & Henderson, A.H. Jr. (1987). A multiple, virtual workspace interface to support user task switching. Proceedings of CHI+GI'87, pp 53-59, ACM. New York, 1987.
8 Cheng, S., Garlan, D., Schmerl, B., Sousa, J. P., Spitznagel, B., Steenkiste, P., Hu, N.: Software Architecture-based Adaptation for Pervasive Systems. Intl Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing, Karlsruhe, Germany. LNCS 2299. Schmeck, Ungerer, Wolf (eds.), 2002.
9 Christensen, H., Bardram, J.: Supporting Human Activities – Exploring Activity-Centered Computing. UbiComp 2002: Ubiquitous Computing, Procs of the 4th Intl Conference, Borriello, Holmquist (eds.), LNCS 2498, pp 107-116, Göteborg, Sweden, 2002.
10 Czerwinski, M., Horvitz, E., Wilhite, S. A diary study of task switching and interruptions, Proceedings of the 2004 conference on Human factors in computing systems, p.175-182, April 24-29, 2004, Vienna, Austria
11 The DAML Services Coalition (multiple authors), "DAML-S: Web Service Description for the Semantic Web", Proc Int'l Semantic Web Conference (ISWC), 2002.
12 Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P. Project Aura: Toward Distraction-Free Pervasive Computing. IEEE Pervasive Computing, April-June 2002.
13 Hightower, J., Borriello, G.: Location Systems for Ubiquitous Computing. Computer 34(8), pp 57-66, 2001.
14 Kozuch, M., Satyanarayanan, M.: Internet Suspend/Resume. Presented at the Fourth IEEE Workshop on Mobile Computing Systems and Applications, Calicoon, NY. Available as Intel Research Report IRP-TR-02-01, Jun. 1, 2002.
15 MacIntyre, B., Mynatt, E., Voida, S., Hansen, K., Tullio, J., Corso, G.: Support For Multitasking and Background Awareness Using Interactive Peripheral Displays. Proc. ACM User Interface Software and Technology (UIST'01), Orlando, Florida, November 2001.
16 Noble, B.: System support for mobile, adaptive applications, IEEE Personal Computing Systems, vol. 7, no. 1, pp 44-49, Feb. 2000
17 Poladian, V., Sousa, J.P., Garlan, D., Shaw, M. 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.
18 Ponnekanti, S., Lee, B., Fox, A., Hanranhan, P.: Icrafter: A Service Framework for Ubiquitous Computing Environments. UbiComp 2001: Ubiquitous Computing, Proceedings of the 3rd International Conference, Abowd, Brumitt and Shafer (Eds.), LNCS 2201, pp 56-75, Atlanta, Georgia, September 2001.
19 Román, M., Hess, C. K., Cerqueira, R., Ranganathan, A., Campbell, R. H., Nahrstedt, K.: Gaia: A Middleware Infrastructure to Enable Active Spaces. IEEE Pervasive Computing, pp 74-83, Oct-Dec 2002.
20 Satyanarayanan, M.: Mobile Information Access. IEEE Personal Communications, Vol. 3, No. 1, February 1996.
21 Schmidt, A. et al.: Context Acquisition based on Load Sensing. UbiComp 2002: Ubiquitous Computing, Proceedings of the 4th International Conference, Borriello and Holmquist (Eds.), LNCS 2498, pp 333-350, Göteborg, Sweden, September 2002.
22 Smith, G., Baudisch, P. et al. GroupBar: The TaskBar evolved. Proceedings of OZCHI'03, Brisbane, Australia, 2003.
23 Sousa, J.P., Garlan, D. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. *Carnegie Mellon Univ. Technical Report CMU-CS-03-183*, 2003.
24 Wang, Z., Garlan, D.: Task Driven Computing. Carnegie Mellon University Technical Report CMU-CS-00-154, http://reports-archive.adm.cs.cmu.edu/cs2000.html, May 2000.