To appear in the Proceedings of UbiSys'03 - System Support for Ubiquitous Computing Workshop, at the 5th Annual Conference on Ubiquitous Computing, UbiComp'03, Seattle, WA

Improving User-Awareness by Factoring it Out of Applications

João Pedro Sousa, David Garlan

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

Abstract. Computers support more and more daily activities for common users. However, user attention is taking a heavy toll when scaling the use of computers for tasks that span many locations, large periods of time, and that are constantly interrupted and resumed. To reduce such toll, computer systems must improve their awareness of user tasks, across multiple devices, and over time spans of weeks or even years. In this paper, we discuss the limitations of building such awareness into applications, and propose to factor the awareness of user tasks into a common infrastructure. We summarize the main features of such infrastructure and distill some of the lessons learned.

1 Introduction

Advances in technology are creating new expectations by users for capabilities delivered by emerging Ubiquitous Computing (Ubicomp) systems. Increasingly, ordinary artifacts and physical spaces offer computing power to the end user: phones, entertainment systems, cars, airport lounges, cafés, etc. A natural consequence of this abundance is that people increasingly expect to push the use of computing beyond the desktop, scaling that use both in space and in time [1]. For instance, a user may start watching a video clip at home, and continue on the bus; he may join a teleconference while walking down the hall, and participate in it while sitting in a smart room; or he may be preparing a conference paper on and off, in the free time between daily meetings and activities.

Consequently, Ubicomp systems will be increasingly required to address the following challenges:

– *Everyday computing*. Users simultaneously handle many *tasks*, such as preparing presentations, writing reports, or answering email, constantly shifting their attention between one and another. Today, computers are involved in more and more everyday activities, and systems are required to support the user in tasks that span weeks or even years, that are constantly interrupted and resumed, that span multiple locations, and that reuse or share information resources with other such tasks.

– *User Mobility*. Ubicomp users should be able to take full advantage of the devices and computing capabilities in their immediate environment: in contrast to a common premise of Mobile Computing, users should not have to rely primarily on devices that they carry with them. For instance, if a user joins a teleconference using his wireless PDA while walking down the hall, why shouldn't he be able to put aside the PDA and take advantage of the large display and wired connectivity in the smart room he just entered? If the user wishes to work on his conference paper both at the office and at home, he should be able to fully use the capabilities at each place, without worrying about the details of recovering the state of his task: opening applications, accessing up-to-date versions of the documents, or even initializing the application to the right places in those documents. However, by freely using the capabilities available at each location, the user will necessarily have to deal with heterogeneous devices, operating systems, and applications.

– *Environment change*.[1] The small devices and networking that are staples of Ubicomp expose the user to dynamic change much more than was the case in the desktop paradigm. To obtain a desired quality of service, users must be aware of the demand posed by alternative computing modalities on limited *resources*, such as battery and bandwidth. Moreover, a setup that corresponds to the user's expectations at one moment may become unacceptable later: for example, in heavily networked environments, remote servers constantly change their response times and even availability.

Handling these challenges with today's software imposes a severe drain on the user's attention. First, whether resuming a task interrupted the day before, or resuming a task suspended in another location, users currently have to find and restart the necessary applications, recover the appropriate settings, migrate or access the up-to-date files being worked on, and recover the previous work positions. Furthermore, users are required to directly manage applications to accommodate dynamically changing and limited resources in the environment, under penalty of obtaining less than ideal performance, or worse, of running out of resources midway through their tasks.

To do a better job at addressing these challenges with respect to user distraction, Ubicomp systems need to improve their awareness of the user's tasks and intent. By incorporating user-awareness, Ubicomp environments can in principle automatically configure and reconfigure themselves to support the relevant user tasks in optimal ways.[2] Ideally, reconfiguration should be triggered, at the user's request or proactively, whenever the user shifts between tasks or moves to a new location,[3] and when-

[1] Informally, the computing *environment* is the set of devices, applications, and resources that are accessible to a user at a particular location.

[2] Although somewhat related, this kind of *automatic configuration* is distinct from the automatic configuration being investigated in other research [3]. There, configuration is taken in the sense of *building and installing* applications into a new environment, whereas here, it is taken in the sense of *activating and recovering* the user-level state so that the user can resume his task with minimal distraction.

[3] More generally, reconfiguration should be triggered whenever the physical *context* of the user changes, be it location, activity (such as sitting, walking, driving…), social situation (alone, with a colleague, addressing a group of people…), etc.

ever the resources in the environment change sufficiently, so that the current configuration no longer best serves the user's intent. Specifically, with suitable knowledge of user intent, such automatic configuration of the environment to support a user's task can handle: finding and starting the necessary applications, recovering the appropriate settings, migrating or accessing the relevant information/files, recovering the previous work positions, monitoring the availability of resources in the environment, and translating the user's intent to appropriate resource-adaptation policies.

Clearly, some of these aspects need to be addressed above the level of individual applications, by a common infrastructure dedicated to the automatic configuration of Ubicomp environments. But then, we need to answer questions like: which user-awareness features should be kept in applications and which should be factored out into the infrastructure? What should be the APIs and knowledge about the user's task exchanged between the infrastructure and the applications? What are adequate semantic primitives to describe the user's task and intent?

In this paper, we describe our experience with factoring user-awareness out of the applications and into a common infrastructure, and argue how that approach supports the challenges above. In the remainder of this paper, Section 2 discusses the limitations of application-based user-awareness relative to those challenges. Section 3 summarizes our approach, as well as the main features of the architecture that supports it. For the sake of space, the formal specifications of the architecture and of the representation of user intent are left out of this paper [5]. Section 4 summarizes the lessons learned about incorporating applications into such architecture, and Section 5 enumerates some related research questions.

2 Application-based User-awareness

Currently, many applications incorporate some level of user-awareness. Typically this is done by having each application learn and store some user-level state, such as preferences, the last few files the user worked on, window size, and active options. Resource-adaptive applications take user-awareness in another direction, by applying user-specific policies for guiding their adaptation to dynamically changing resources. For instance, an adaptive speech recognizer might make tradeoffs between the accuracy of the recognition and the latency constraints expressed by the user, based on the available CPU cycles.

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

- *Software engineering costs*. Currently, user-awareness features are added to applications with little concern for generality, and often by intertwining those features with application code. This stovepiped approach makes it very hard to reuse solutions across different applications.
- *Awareness of user tasks*. In everyday computing, the *same* application may be used to support different user tasks in turn. For instance, a text editor may be used to

support writing a conference paper at one time, but writing a monthly report in another, each with its own files, options and window settings. Currently, applications store user-level state, at best, on a per-user basis (older applications store one user-level state, which all users share). Unfortunately, the user-level state that should be recovered can be different for each user *task*.

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

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

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

3 User-aware Infrastructure

Our approach factors user-awareness out of the applications and into a common infrastructure. Such infrastructure exploits lightweight models of user tasks (more on this below) to perform automatic configuration and reconfiguration of Ubicomp environments on behalf of the user, and according to the requirements of each user task.

There are two parts to the problem of automatic configuration of Ubicomp environments. First, before making any automatic configuration, the infrastructure needs to know *what* to configure for: what does the user need from the environment in order

to carry out his task. 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 work, 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 1. Summary of the software layers in the infrastructure

layer	*mission*	*subproblems*
Task Management	***what*** does the user need	• monitor the user's task, context and intent • map the user's task to needs for services in the environment • map user intent to QoS/resource tradeoffs • 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 environment-specific capabilities • 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

Table 1 summarizes the purpose of the software layers in the infrastructure. The top layer, Task Management 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 signals that he wishes to resume his work at a new location, Task Management coordinates the access to all the information related to the user's task, and negotiates the task support with the Environment Management. 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 interrupt the current task or switch to a new task, Task Management coordinates saving the user-level state of the interrupted task and instantiates the intended new task, as appropriate. The user-level state of a task refers to the user-observable set of properties in the environment that characterize the support for that task. Specifically, the set of *services* marshaled to support the task (more on this below), the user-level settings (preferences, options) associated with each of those services, the files being worked on, user-interaction parameters such as window size, cursors, etc. The user-level state of a task is captured and recovered at the granularity of each service supporting the task. The user-level state of a task includes a utility-based framework for expressing the user's preferences with respect to QoS tradeoffs and resource-adaptation policies.[4]

[4] Other work follows a utility-based approach to find out the optimal tradeoffs in configuring an application at *design-time* [2]. Yet another work follows a utility-based approach to guide the *run-time* resource-adaptation policies of an application on a per-operation basis [4]. Our

The intermediate 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. With respect to heterogeneity, when the user needs, for instance, *speech synthesis*, the Environment Management will find and map an application that can provide that *service*. 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. The mapping between user needs and concrete applications/devices can thus be automatically adapted at runtime. In contrast, in traditional environments where dynamic change is not an issue, this mapping is typically handled manually by the user. The Environment Management adjusts such mapping automatically, not only in response to changes in the user's needs (adaptation initiated by the Task Management), but also in response to changes in the environment's capabilities and resources (adaptation initiated by the Environment Management itself).

Finding the best match between what the user wants and what the environment has to offer in practice corresponds to maximizing a *utility function*. Such utility function is learned by the Task Management and it expresses the user's preferences and intent for the task at hand. It has three parts: first, *configuration preferences* capture the preferences of the user with respect to the set of services to support the task at hand. Second, *supplier preferences* capture which specific applications/components are preferred to supply the required services; and third, *QoS preferences* capture the acceptable Quality of Service (QoS) levels and preferred tradeoffs. To illustrate configuration preferences, suppose the user needs to prepare a review for a promotional video. For taking notes, the user may prefer 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, the user is willing to type or write the text. As another example, suppose the user is moving around carrying only his handheld, and he wants to watch a soccer game available from an on-line video feed. Since video and audio are competing for limited bandwidth, sometimes the video quality degrades so much that the user can no longer follow the game. When that happens, the user is willing to forego the video and have the meager bandwidth be allotted to provide acceptable audio. As an example of supplier preferences, for typing notes (*text editing* service), the user may prefer MSWord over Notepad or Emacs, and be unwilling to use the *vi* editor at all. As an example of QoS preferences, consider again watching a video over a network link. Suppose that the bandwidth suddenly drops: should the video player reduce image quality or frame-update rate? For the soccer game, frame-update rate should be preserved at the expense of image quality; however, when the user is watching the promotional video, he may prefer image quality to be preserved at the expense of frame-update rate.

The lower layer in Table 1, the environment, holds the applications and devices that provide the services that can be marshaled to support the user's task. Configura-

---

work exploits a utility-based framework to find, at *run-time*, the set of applications, the resource constraints on each, and the local resource-adaptation policies, that optimize the overall QoS for the user's task, according to his intent.

tion issues aside, these components interact with the user in the same way as they would without the presence of the infrastructure. The infrastructure steps in only to the extent of automatically configuring those components on behalf of the user. The specific capabilities of each component are manipulated by the Environment Management Layer, which acts as a translator for the environment-independent descriptions of user needs issued by the Task Management. The infrastructure can accommodate components with a wide range of sophistication in matters like resource adaptation and context-awareness.

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. The Environment Management acts at a time scale of a few seconds to evaluate the adequacy of the mapping between the requested services and specific components. Adaptive applications (QoS-aware and context-aware) choose appropriate computation tactics at a time-scale of milliseconds.

4 Lessons Learned

Legacy applications. An important feature for any new infrastructure is that it allows for the easy integration of the wealth of applications already written for all sorts of devices. Our infrastructure easily accommodates legacy applications as long as they expose an API for importing and exporting user-level state: the richer that API, the better job we can do in recovering the state of user tasks. Fortunately, providing such APIs is a growing tendency in the industry.

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

Controlling the policies of QoS-aware (resource-adaptive) applications is more challenging. These applications tend to fall into two fields: first, those coming from research or open-source projects, for which controlling the policies, although possible, can be an involved task. Second, commercial software, which either doesn't expose mechanisms to control the adaptation policies in the offered APIs, or for which we often can not observe a reliable correlation between the controls transmitted to the application and its actual behavior – consistently greedy. But here also there is reason for being optimist: recent versions of media streaming applications offer a rich API to control the resource demand and QoS tradeoffs of the application. Our experiments with, for instance, RealOne Player indicate a good correlation between the controls and the actual behavior with respect to resource adaptation and QoS tradeoffs.

Heterogeneity. A mobile user takes full advantage of the capabilities around him to support his task, but in doing so, he is exposed to the heterogeneity of devices and applications. For instance, the user may join a teleconference using his wireless PDA while walking down the hall, and then switch to the devices in a smart room, as soon as he reaches it; or the user may start editing a document on his desktop, and continue on his PDA, en route to a meeting.

In our experience, applications that provide the same *service*, say *video conferencing*, or *text editing*, share a common core of concepts that characterizes their user-level state. For instance, applications providing *text editing* share the concept of file(s) being worked on, cursor position, and some window settings, such as scrolling. However, even basic concepts like editing overstrike, or window size may not be supported by some applications (e.g. on a small device, the window size may be fixed to the size of the display). Not withstanding such variation of the capabilities of applications, we were able to define an ontology of concepts that offers a good leverage for defining the user-level state of each service.

The Environment Management layer is the natural place to perform the back and forth translation between such abstract user-level state and the concrete features of each application. Application wrappers hold the knowledge of *how* to control the application to recover *what* the user was working on. Such wrappers instantiate as much as possible from the description of the user-level state, leaving untouched the concepts that cannot be translated into the application's features. This way, when the same task is migrated to an environment with the required features, those aspects of the user-level state can be recovered. To represent the abstract user-level state of the services, we used (XML-based) markup formats. For the typical size of such descriptions (hundreds to a few thousand bytes) the overhead of markup as opposed to raw data formats is not significant when contrasted with the advantages of application- and platform-independence.

User distractions. During our research targeted at addressing the requirements entailed by everyday computing, user mobility, and environment change, we concluded that application-based user awareness has severe structural limitations (see Section 2). Such limitations result in added demand on the user's attention. Our research demonstrates that by adding an infrastructure aware of user tasks we can do a much better job at addressing the requirements mentioned above. By automatically searching, setting up and maintaining service configurations that best meet the user's needs, we argue that distractions are greatly reduced for users of Ubicomp environments.

5 Future work

Having a common infrastructure that exploits descriptions of user tasks holds great potential for increasing user-awareness in Ubicomp systems, and consequently to reduce user distractions. For the sake of space, some of the technical challenges that stem from such approach were left out of this paper [5]: how to represent user intent in an application-independent fashion? Which functionality to incorporate in the infrastructure for automatically configuring the environment? Exactly which assumptions

are shared between applications and the common infrastructure, and what is the distribution of responsibilities? What are the APIs that an application should expose to enable the role of the infrastructure, and what are the APIs that the infrastructure provides for improving the application's user-awareness?

Our present work targets related research challenges to be covered in future publications: which functionality to incorporate in the infrastructure for capturing descriptions of user tasks, and for explaining the configuration decisions to the user? Which are adequate metrics to evaluate the user's costs and benefits associated with configuring Ubicomp environments, under scenarios of user mobility, everyday computing, and environment change? Which are representative scenarios in those categories?

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. Candea, G., Fox, A.: A Utility-Centered Approach to Building Dependable Infrastructure Services. In: Proceedings of the 10th ACM SIGOPS European Workshop (EW'2002), Saint-Émilion, France (2002) 213-218
3. Kon, F., Yamane, T., Hess, C., Campbell, R., Mickunas, M.: Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. In: Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), San Antonio, Texas (2001)
4. Narayanan, D., Satyanarayanan, M.: Predictive Resource Management for Wearable Computing. In: Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys'03), San Francisco, CA (2003)
5. Sousa, J.P., Garlan, D.: The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. Carnegie Mellon Technical Report, CMU-CS-03-183, August 2003.