Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments

João Pedro Sousa and David Garlan
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh PA 15213 USA
{jpsousa,garlan}@cs.cmu.edu

Abstract: Ubiquitous computing poses a number of challenges for software architecture. One of the most important is the ability to design software systems that accommodate dynamically-changing resources. Resource variability arises naturally in a ubiquitous computing setting through user mobility (a user moves from one computing environment to another), and through the need to exploit time-varying resources in a given environment (such as wireless bandwidth). Traditional approaches to handling resource variability in applications attempt to address the problem by imposing uniformity on the environment. We argue that those approaches are inadequate, and describe an alternative architectural framework that is better matched to the needs of ubiquitous computing. A key feature of the architecture is that user tasks become first class entities. User proxies, or *Auras*, use models of user tasks to set up, monitor and adapt computing environments proactively. The architectural framework has been implemented and is currently being used as a central component of Project Aura, a campus-wide ubiquitous computing effort.

Key words: Ubiquitous computing, mobility, architectural framework, architectural style.

1. INTRODUCTION

Fuelled by Moore's Law, technology is moving towards a world populated with increasing numbers of heterogeneous computing devices, services and information sources. This emerging world of ubiquitous computing poses a number of significant challenges for software systems, and software architecture in particular.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35607-5_15

J. Bosch et al. (eds.), *Software Architecture*

One of the most important challenges for architectural design is to support the relatively new quality attribute of user mobility. Ideally, a ubiquitous computing infrastructure would allow users to move their computational tasks easily from one *environment*[1] to another. Moreover, users should be able to take full advantage of the local capabilities and resources within a given environment, even as other users and devices enter and leave that environment, and as resources (like available bandwidth) change [30].

Current approaches to user mobility are based on one of four techniques, none of which fully achieves these goals. One approach is to support as much of a user's computing needs as possible on a *mobile machine*. A second approach is to compute via *remote access* to a computing server that stores a users personal state and preferences, much as X-terminals do. A third approach is to provide *standard applications* that are ported to and installed in all environments. Those applications are extended to become aware of user intention and mobility. A fourth approach is to provide standard virtual platforms (such as the Java Virtual Machine) that enable *mobile code* to follow the user as needed.

There are two problems with these approaches. First, since to some degree they assume a homogenous computing baseline, they cannot take full advantage of the diverse capabilities of each environment, such as external displays, processors, and I/O devices. Second, they lack the ability to handle dynamic variations of capabilities and resources in the environment without overburdening the user with manual tuning and reconfiguration.

In this paper we propose an alternative approach that enables *mobile users* to make the most of ubiquitous computing environments, while shielding those users from managing heterogeneity and dynamic variability of capabilities and resources. Specifically, we describe an architectural framework[2] for ubiquitous computing applications with the following key features: first, user tasks become first class entities that are represented explicitly and autonomously from a specific environment. Second, user tasks are represented as coalitions of abstract services. Third, environments are equipped to self-monitor and renegotiate task support in the presence of run time variation of capabilities and resources.

As we will see, this architectural framework has a number of important benefits. By representing user tasks explicitly, we provide a placeholder to capture user intent. This knowledge is used to guide the search for suitable configurations in each new environment. By representing tasks as service

[1] In this paper, we define "environment" informally as the set of devices and applications that are accessible to a user standing at a particular location.

[2] By "architectural framework" we mean an architectural style for applications and services together with supporting run-time infrastructure (or middleware) that supports their invocation, interaction, and reconfiguration.

coalitions, the infrastructure can recognize when all the essential services in a task can be supported, instantiating them jointly, or otherwise provide early warning to the user that that is not possible. By providing an abstract characterization of the services in a task, the infrastructure can search heterogeneous environments for appropriate matches to supply those services. By providing the environment with self-monitoring capabilities, the infrastructure can detect when task requirements (such as minimum response time) are not met, and search and deploy alternative configurations to support the task.

Section 2 describes the proposed architectural framework, making it concrete how the intended features are supported by the architectural design. Section 3 illustrates the workings of the framework using a task migration scenario as an example. Section 4 details the current state of our research, discusses the benefits and limitations of what has been achieved, and outlines future research. Section 5 describes related work, while Section 6 summarizes the main results.

2. AURA'S ARCHITECTURE

The central architectural challenge in supporting computational needs of mobile users is to satisfy two competing goals. The first is to *maximize the use of available resources* – that is, effectively exploiting the increasingly pervasive computing and communication resources in our environments. The second is to *minimize user distraction and drains on user attention.*

Today, a major source of user distraction arises from the need for users to manage their computing resources in each new environment, and from the fact that the resources in a particular environment may change dynamically and frequently.

In Project Aura at Carnegie Mellon University we are developing a new solution to this problem based on the concept of personal *Aura.* The intuition behind a personal Aura is that it acts as a proxy for the *mobile user* it represents: when a user enters a new environment, his or her Aura marshals the appropriate resources to support the *user's task.* Furthermore, an Aura captures constraints that the *physical context* around the user imposes on tasks (more on this below). Examples of user tasks (or simply *tasks*) are: writing a paper, preparing a presentation or buying a house. Each of these tasks may involve several information sources and applications.

To enable the action of such a personal Aura, we need an architectural framework that clarifies which new features and interfaces are required at system- and application-level. The framework must also define placeholders for capturing the nature of the user's tasks, personal preferences, and intentions. This knowledge is key to configure and monitor the environment, thus

shielding the user from the heterogeneity of computing environments, as well as from the variability of resources.

Figure 1 shows a bird's-eye view of our architectural framework. There are four component types: first, the *Task Manager*, called *Prism*, embodies the concept of personal Aura. Second, the *Context Observer* provides information on the physical context and reports relevant events in the physical context back to Prism and the *Environment Manager*. Third, the Environment Manager embodies the gateway to the environment; and fourth, *Suppliers* provide the abstract services that tasks are composed of: text editing, video playing, etc. From a logical standpoint, an environment has one instance of each of the types: Environment Manager, Context Observer and Task Manager.[3] Although the boundaries of an environment are defined administratively, they typically correspond to some physical area, like a floor or a building. Each environment may have several service Suppliers: the more it has, the richer the environment is. Much like naming servers on networks do today, Environment Managers cooperate to find and marshal remote Suppliers when that is required by the user's task.

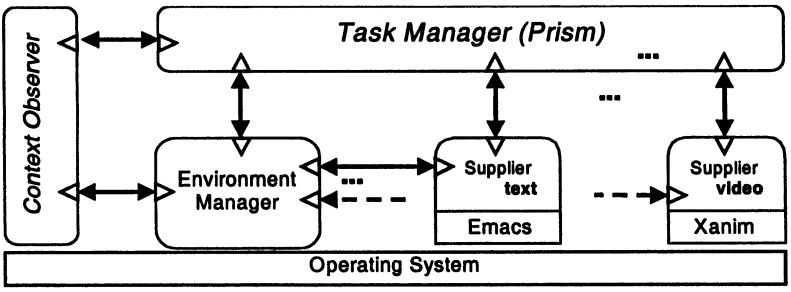

Figure 1. Aura bird's-eye view

2.1 Task Manager (Prism)

Prism embodies the concept of a personal Aura. It strives to minimize user distractions in the face of the following four kinds of change:

- The user moves to another environment: Prism coordinates the migration of all the information related to the user task to the new environment, and negotiates the task support with the new Environment Manager.
- The environment changes: Prism monitors Quality of Service information provided by the Suppliers supporting the user's task. Whenever that information becomes incompatible with the requirements of the current

[3] An environment may have redundancy of these components for the sake of robustness.

task, or the monitored Supplier fails, Prism queries the Environment Manager to find an alternative configuration to support the task.

- The task changes: Prism monitors explicit indications from the user and events announced by the Context Observer. Upon getting indication that the user intends to interrupt the current task or to switch to a new task, Prism coordinates saving the state of the interrupted task and instantiates the intended new task, as appropriate.
- The context changes: task descriptions include constraints on the context, capturing requirements on privacy, user activity (sitting, driving...) etc. When these constraints are not met, Prism coordinates the suspension of the executing task, or adjusts the parts that are affected by the context change; for instance, hiding the display of sensitive data when someone else comes into the user's office.

The key idea behind Prism is a platform-independent description of user tasks [29]. Earlier research in this area treated tasks as a cohesive collection of applications. When a user refers to a particular task, the system automatically brings up all the applications (and files) associated with that task. This mechanism relieves the user from finding files and starting applications individually [18]. In our work, we extend this notion by describing a task as a coalition of abstract *services*, such as "edit text" and "play video." This form of abstraction allows such tasks to be successfully instantiated in different environments using different supporting applications. For example, in a Windows environment Microsoft Word and Media Player might be used to provide the edit text and play video services, whereas in a Unix environment Emacs and Xanim could be used.

2.2 Service Suppliers

Suppliers provide the abstract services that tasks are composed of. In practice, these abstract services are implemented by wrapping existing applications and services to conform to Aura APIs. For instance Emacs, Microsoft Word and Notepad can each be wrapped to become a supplier of *text editing* services.

Such wrappers play a fundamental role while instantiating a task based on its platform-independent description: the wrappers map the abstract service descriptions into application-specific settings. Note however, that different suppliers for the same type of service will typically have different capabilities. For instance, a basic text editor may not support spell checking, or even be aware of what spell checking means. Therefore, the description of the service must be such that a Supplier is able to extract the information it can recognize, without having to deal with information it does not know how to handle.

We address this requirement by using markup formats, specifically XML-based, for the description of services. The underlying assumption is that Suppliers of a given service type share a vocabulary of tags and the corresponding interpretation. Naturally, each service type is characterized by a distinct vocabulary of tags corresponding to the information relevant for the service, although there are some commonalities across service types.

2.3 Context Observer

Context Observers provide information about the physical context and report events in the physical context back to Prism and the Environment Manager. Examples of such information are user location, recognition (authentication,) activity, other people in the vicinity, etc. Context Observers in each environment may have different degrees of sophistication, depending on the sensors deployed in that environment. The more sophisticated a Context Observer, the less Prism has to rely on explicit indications from a user concerning his intentions. For the purpose of the points illustrated in this paper, we will not discuss Context Observers in further detail.

2.4 Environment Manager

The Environment Manager component embodies the gateway to the environment: it is aware of which Suppliers are available to supply which services, and where they can be deployed. It also encapsulates the mechanisms for distributed file access.[4]

When Suppliers are installed in an environment, they become registered with the local Environment Manager. Such a registry is the base for matching requests for services. For Suppliers with limited sharing capacity, such as those that involve input/output devices, the registry also keeps track of the available capacity. When instantiating a task in a new environment, the registry is consulted by location mechanisms for abstract services. Those mechanisms are built on top of currently available tools [1,4].

In addition to individual service discovery, a sophisticated Environment Manager evaluates each alternative configuration of service suppliers to select the one that presents a better match to the user's preferences.

2.5 Addressing Ubiquity

When Prism migrates a task from one environment to another, the deployment of the Suppliers across devices may be very different. Moreover,

[4] The choice of the actual mechanisms for file access is an implementation issue: one Environment Manager might require the files to be sent over some protocol like ftp, while another might rely on a distributed file system.

even within the same environment, that deployment may change dynamically, as component reachability changes.

For example, suppose the user stops typing at a desktop, takes hold of a wireless PDA, and goes down the hall for coffee. Initially, Prism and the supplier of text editing were probably both running on the desktop. When the user leaves the office, Prism has to communicate with a supplier for text editing on the PDA. From a task viewpoint, however, Prism is still coordinating a supplier of text editing, regardless of the particular application that is providing the service or on which device that application is deployed. Furthermore, in one environment the available interaction mechanism may be CORBA, while in another environment it may be COM or RPC.

We use a technique, similar to stub generation, to insulate the components both from dynamic redistribution and from alternative interaction mechanisms. That technique is the explicit implementation of connectors.

There are four types of connectors in Aura: between Prism and an arbitrary Supplier, between Prism and the Environment Manager, between the Context Observer and Prism, and between the Context Observer and the Environment Manager. Each of these connector types is defined by an interaction protocol appropriate to the component type it connects. For instance, the connector type between Prism and the Suppliers supports protocols to capture and recover the execution state of services.[5] All the component types in Aura's architecture have standard interfaces, or *ports* (represented by the triangles in *Figure 1*). For instance, all the ports of Prism that attach to Suppliers have the same API.

Each connector type may have many implementations, each appropriate to a specific low-level interaction mechanism and to the distribution of the components it connects. For example, if the two ends of the connector are deployed on the same device, an implementation that uses local method calls is appropriate. If the connector is between two different devices, its implementation is comprised of two code stubs, one in each device. Each of the stubs makes local method calls to the corresponding port in the attached component, and uses environment-specific communication mechanisms to pass control and data to the other end of the connector.

When Prism requests support for a task to the Environment Manager, the latter annotates each service request with three things: a handle for the appropriate connector to reach the supplier, supplier location information, and a handle for the supplier proper. Prism uses the first handle to dynamically load its end of the connector, and then uses the second and third pieces of information to initialise that end of the connector. Thereafter, Prism communicates with the supplier through the connector, oblivious of distribution issues. If a supplier becomes unable to continue to support the task (e.g.,

[5] For space reasons, we do not detail the protocols further.

because the user left his desk) Prism just requests for another supplier subject to the new context constraints to the Environment Manager – and again initialises it and uses it seamlessly. Thus, Aura components need not be aware of distribution issues: the Environment Manager takes charge of assembling and adapting the configurations using the appropriate connectors.

3. AURA AT WORK

To illustrate how the Aura architecture achieves its goal of supporting user mobility, we now describe a simple scenario of task migration, focusing on the interactions among the components identified in Section 2.

Fred is at home working on the organization of a conference in a remote place. He's gathering information on possible venues and getting budgets for catering. The web pages of some of the hotels include short videos featuring virtual visits to the premises and Fred already downloaded some of these for reference. Fred is also taking notes on a spreadsheet concerning his appraisal of each venue along with the alternative catering budgets.

Fred leaves home and heads to his office. Since Fred intends to continue working on the organization of the conference, Aura sets up that task at Fred's office so that he can resume his work as soon as he is recognized entering the office: a web browser over the recently visited pages, the downloaded videos paused at the same places, and a spreadsheet containing all the entered figures. Since there is a big screen on the wall of Fred's office, that is preferred to stage the video and web browsing, releasing monitor space for the spreadsheet.

Fred is working at home when the Home Context Observer[6] notices Fred leaving the house. The Context Observer lets Prism know that Fred is leaving – interaction (1) in *Figure 2*, and that causes Prism to undergo state transition (a), where it realizes it should suspend the task ongoing at home. Prism then requests to checkpoint the state of each of the services being provided as part of the ongoing task – interaction (2). In interaction (3), the Home Prism tells the Home Environment to deallocate those services and to store all the involved files back into a globally accessible file server – interaction (4).

After checking Fred's schedule, Prism infers that he is likely to head to the office, and (5) conveys that information along with an estimated time of arrival to the TM at the office. That triggers state transition (b) in the TM at the office, causing it to request the Office EM (6) to retrieve the updated description of the tasks Fred has been working on – interaction 7.

[6] For convenience, we refer to "*component* at *location*," for instance "Home CO," meaning "the Context Observer at Fred's home."

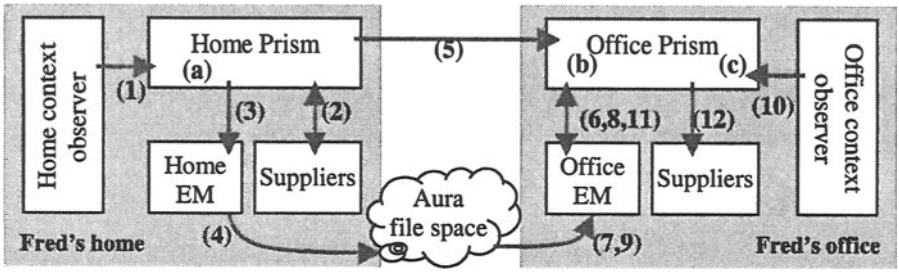

Figure 2. Fred goes from home to the office

Given that description, Prism at the office extracts which files will be necessary for Fred to work on, and requests the Office EM to retrieve them – interaction (8). The Office EM checks if the copies stored locally are up-to-date, retrieving updated copies as necessary (9). As soon as the Context Observer at the office recognizes Fred coming into his office, it informs Prism of that (10) causing Prism to undergo state transition (c). This triggers the request of suppliers for the services involved in the task (11) and the subsequent restoring of the execution state at the allocated suppliers (12).

Upon instantiating a task, Prism slices the task description in order to pass the relevant service descriptions to each of the suppliers. *Figure 3* shows an example of the service description exchanged between Prism and a supplier of text editing services. Notice the two top-level elements, one describing the service, the other the data that the service must access. Within the service description, there are elements that are specific to the service type (in the example, pane settings, spelling etc.), and others that are common to all service types: an estimate of the duration of the service supply. In the example, the user will be happy if the service is provided for 30 minutes or more, but would rather seek an alternative if it cannot by provided for at least 10 minutes. This kind of indication can be used to manage finite resources like battery charge in mobile platforms [10], giving the user an early warning when the requirements of the task cannot be met. The material description identifies the origin of the data – typically a file name or URL – and format. Additionally, the material element includes a description of where the user left off: in the case of text, the cursor, scrolling and zoom factor in effect when the editing was interrupted.

Service suppliers parse these descriptions when instantiating a task, extracting as much information as they can map to the settings of the specific application. Such descriptions are updated when Prism requests a service checkpoint. The update process is conservative with respect to the existing information. For instance, even if a simple text editor could not instantiate the spell-check settings, it preserves that part of the description so that a more sophisticated editor in another environment can use it down the line.

```
<auraTask id="demo">
  <service type="editText">
    <duration unit="minutes" bad="10" good="30"/>
    <settings pane_height="360" pane_width="200">
      <spelling enabled="yes" ignoreAllCaps="yes"/>
      <editing overstrike="no" replaceSelection="yes"/>
    </settings>
  </service>
  <material origin="myTextFile" format="txt">
    <state cursor="104" scroll="28" zoom="100"/>
  </material>
</auraTask>
```

Figure 3. Description of a text editing service

4. DISCUSSION AND FUTURE WORK

The current implementation of the architecture in *Figure 1* supports the migration of simple user tasks interchangeably between personal computers running Windows or Linux. As a proof of concept, we have wrapped Microsoft Word and Emacs as suppliers of text editing services, and Media Player and Xanim as suppliers of video playing services. The current implementation of the Environment Manager has rudimentary service registry abilities, and relies on distributed file systems like Coda or AFS [23,24] for file access across environments. We have not yet integrated research on context observation: Prism reacts to explicit task suspend and resume commands issued by the user.

By describing tasks as coalitions of abstract services, we rely on the ability to migrate those descriptions between resource-rich environments. This approach imposes fewer requirements on platform compatibility than an approach that relies on the ability to migrate executable code.

While the current implementation shows the feasibility of automated task migration, it is limited by the granularity of the task components (full applications working separately from each other) and by its inability to anticipate or infer what the user wants to do next. To address these problems, we have begun to develop support for finer-grained tasks and richer models of user intent. In its ultimate form we anticipate the need for Aura to support a spectrum of task models ranging from simple invocation of applications to sophisticated models that can anticipate immediate needs of users, or even assist them in accomplishing some complex multi-step activity (like financial planning, travel assistance, or health management.) Ongoing work on Project Aura builds on research in computer-human interaction and machine learning, exploring semiautomatic learning of richer models of tasks [16,26].

A key enabler both for capturing sophisticated models of tasks and for enacting them is the integration of physical context observation: the user location, what activities are competing for a user's attention, who else is in the vicinity, etc. [5,8,14]. If the user has to specify every detail of a task, then no one will use Aura. On the other hand, systems are notoriously poor at automatically capturing user intent. Hence, Aura must strike a balance between user involvement and automatic inference of user intent. Our assumption is that an Aura should prove useful even with no deeper knowledge of the task beyond the coalition of services currently being used. Furthermore, our approach should prove useful only with rudimentary context awareness, specifically recognizing a user entering and leaving a given environment.

The self-awareness and adaptability of the environment is addressed at two levels. At the higher level, the infrastructure monitors the availability and performance of whole components and of the communications infrastructure, evaluating possible alternatives for supporting a user task when the requirements for such a task are not met by the current configuration. This coarse-grain adaptation builds on monitoring and adaptation mechanisms like the ones described in [9] and is currently subject of research for integration into the architecture described in Section 2.

At the lower level, system components themselves are endowed with the ability to adjust their operation following the variation of available resources like CPU, bandwidth, battery charge, etc. Aura's architecture addresses the problem of representing the adaptation policy that is appropriate to a user's intent using the notion of utility functions – see for instance [22]. Suppose, for example, that a user is viewing a video over a network connection for which the bandwidth suddenly drops. A fidelity-aware component can deal with resource limitations by reducing the fidelity of (the results of) the computation, but in the example should it reduce the frame-update rate or the image quality? For watching a sports video, it should preserve higher framerates at the expense of image quality; but for watching a tour of a museum, it should do the opposite.

We are currently working on the integration of mechanisms for coarse-grained adaptation of configurations [9], and for fine-grained adaptation of computation fidelity in components [6]. The latter is closer to being fully integrated into the architecture described in Section 2. Both mechanisms are driven by representations of user intent that reside at the task level. By providing a placeholder to capture user intent, task descriptions enable a clean separation of concerns between determining the appropriate fidelity-adaptation policies, at task description level, and the mechanisms to enact those policies, at the level of applications and operating system extensions.

5. RELATED WORK

Flexible partitioning of applications in a wide-area setting is addressed by research in distributed computing [11,17,28]. However, applying those results in ubiquitous computing environments is likely to lead to systems that are hard to deploy and manage. This is due to scale, heterogeneity, and rate of change within those environments. Other infrastructures that specifically target ubiquitous computing take the approach of deploying standard virtual platforms in every device [12,13]. Such infrastructures enable code mobility and therefore enable applications to follow and serve mobile users. It is not clear, however, how much such mobile applications will be able to leverage the diversity of devices and available interaction modes in local environments. On the other hand, trying to build super-applications that deal with a multitude of device capabilities and interaction modalities has obvious software engineering implications.

Applications can also be extended to capture models of user intent [2,15]. However, addressing this problem at the application level has obvious limitations in the face of user mobility through heterogeneous environments. For instance, if the user intent information concerning text editing is captured within Microsoft Word, it cannot be used when the user comes into an environment where Emacs is the only available text editor. By having the knowledge captured in an application-independent way by the infrastructure, we are able to use that knowledge in heterogeneous environments.

Another example comes from research in fidelity-aware computing. With the goal of providing better quality of service to the user and better resource management, applications are commonly extended to incorporate the mechanisms for resource adaptation [10,20]. Determining the adaptation policy that best serves the intent of the user then becomes a hard problem. We claim that such problem is best addressed at the task level [6].

To the authors' best knowledge, Aura's approach is novel in building high-level, application-independent models of user tasks, and in using those models to setup and adapt ubiquitous computing environments.

Aura's architecture uses connectors as first-class entities not only at the design level, but also at the implementation level [25,27]. The explicit encoding of connectors delivers encapsulation of interaction mechanisms and of distribution issues, making it much easier to design and build the components. Of course, middleware and distributed computing infrastructures have addressed such issues in a generic form [7,21]. However, we pull the *use* of such generic mechanisms out of the application and infrastructure components, and into architecture-specific connectors. By doing so, we create added flexibility to adapt to the existence of different interaction mechanisms in different environments, and to dynamically choose the most appropriate mechanism to reach a particular component.

6. CONCLUSION

In this paper we have described an architectural framework that solves two of the hard problems in developing software systems for ubiquitous computing. First, it attacks the problem of allowing a user to preserve continuity in his/her work when moving between different environments. The key advantage of this framework over other traditional approaches is that it allows the system to tailor the user's task to the resources in the environment. Second, it attacks the problem of adapting the on-going computation of a particular environment in the presence of dynamic resource variability. As resources come and go, the computations can adapt appropriately.

The key ingredients of the architectural framework are explicit representations of user tasks as collections of services, context observation that allows the task to be configured in a way that is appropriate to the environment, and environment management that assists with resource monitoring and adaptation. Each of these capabilities is encapsulated in a component of the architectural framework (the task manager, environment manager, and context observer, respectively). The services needed to support a user's task are carried out by a set of components termed *service suppliers.* Service suppliers typically are implemented as wrappers of more traditional applications and services. Finally, interactions between the parts are carried out by explicit connectors that hide details of distribution and heterogeneity of service suppliers.

The architecture has been implemented in prototype form, permitting task migration for a small set of services between Unix- and Windows-based environments. While this implementation is only a first step, already it demonstrates that certain kinds of task migration and adaptation can be supported in the Aura architecture. However, complete evaluation of the architecture will only be possible once we have populated the environment with additional service suppliers, increased the number of environments supported by the framework (e.g., PDAs and smart rooms), and developed a number of more complex task descriptions.

ACKNOWLEDGEMENTS

We thank Takahide Matsutsuka and Tadashi Okoshi for implementing the supplier wrappers in the current prototype. We would also like to thank Rajesh Balan, Jason Flinn, Dushyanth Narayanan, SoYoung Park, Mahadev Satyanarayanan, and Bradley Schmerl for fruitful discussions. This research is supported by DARPA under Grants N66001-99-2-8918 and F30602-00-2-0616. Views and conclusions contained in this document are those of the

authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA.

REFERENCES

1. W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, J. Lilley. The design and implementation of an intentional naming system. Proceedings of the Seventeenth Symposium on Operating System Principles. Kiawah-Island Resort, North Carolina, December 1999.
2. D. Albrecht, I. Zukerman, A. Nicholson, A. Bud. Towards a Bayesian model for keyhole plan recognition in large domains. Proc. 6th Int. Conference on User Modeling (UM '97), pp 365-376. SpringerWien, Jameson, Paris and Tasso (Eds.) New York, 1997.
3. V. Ambriola, P. Ciancarini, C. Montenegro. Software Process enactment in Oikos. Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments. SIGSOFT Software Engineering Notes, pp 183-192, Irvine, California, 1990.
4. K. Arnold, B. O'Sullivan, R. Scheifler, J. Waldo, A. Wollrath. The Jini Specification. Addison-Wesley, 1999.
5. S. Baker, T. Kanade. Hallucinating faces. Proceedings of the Fourth International Conference on Automatic Face- and Gesture-Recognition, Grenoble, France, March 2000.
6. R. Balan, J. Sousa, M. Satyanarayanan. Meeting the Software Engineering Challenges of Adaptive Mobile Applications. Submitted for publication, March 2002.
7. A. Birrell, B. Nelson. Implementing remote procedure call. ACM Transactions on Computer Systems, 2(1), pp 39-59, ACM Press, New York, February 1984.
8. P. Castro, P. Chiu, T. Kremenek, R. Muntz. A Probabilistic Room Location Service. Proc. Ubicomp 2001: Ubiquitous Computing. Atlanta, Georgia, September 2001.
9. S. Cheng, D. Garlan, B. Schmerl, J. Sousa, B. Spitznagel, P. Steenkiste, N. Hu. Software Architecture-based Adaptation for Pervasive Systems. International Conference on Architecture of Computing Systems Trends in Network and Pervasive Computing, Karlsruhe, Germany, April 8-11, 2002. To appear in LNCS, Volume 2299.
10. J. Flinn, M. Satyanarayanan. Energy-aware adaptation for mobile applications. Proceedings of the 17th ACM Symposium on Operating Systems Principles, Kiawah Island Resort, South Carolina, December 1999.
11. I. Foster, C. Kesselman. Globus: A metacomputing infrastructure toolkit. International Journal of Super-computer Applications and High Performance Computing, 11(2), pp 115–128, 1997.
12. J. Gosling, B. Joy, G. Steele. The Java Language Specification. Addison-Wesley, 1996.
13. R. Grimm, T. Anderson, B. Bershad, D. Wetherall. A system architecture for pervasive computing. Proceedings of the 9th ACM SIGOPS European Workshop, pp 177-182, Kolding, Denmark, September 2000.
14. A. Harter, A. Hoper, P. Steggles, A. Ward, P. Webster. The Anatomy of a Context-Aware Application. Proceedings of the Fifth ACM/IEEE International Conference on Mobile Computing and Networking, pp 59-68, Seattle, Washington, August 1999.
15. E. Horvitz, J. Breese, D. Heckerman, D. Hovel, K. Rommelse. The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, pp 256-265, Madison, Wisconsin, 1998.
16. N. Kushmerick, S. Hanks, D. Weld. An Algorithm for Probabilistic Least-Commitment Planning. Proceedings of the Twelfth National Conference on Artificial Intelligence. Seattle, Washington, July 1994.

17. M. Lewis, A. Grimshaw. The core Legion object model. Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, pp 551–561, Syracuse, New York, August 1996.
18. B. MacIntyre, E. Mynatt, S. Voida, K.Hansen, J. Tullio, G. Corso. Support For Multitasking and Background Awareness Using Interactive Peripheral Displays. Proc. ACM User Interface Software and Technology (UIST'01), Orlando, Florida, November 2001.
19. H. Nii. Blackboard Systems. AI Magazine, 7(3), pp 38-53 and 7(4), pp 82-107, 1986.
20. B. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, K. Walker. Agile Application-Aware Adaptation for Mobility. Proceedings of the 16th ACM Symposium on Operating System Principles, October 1997, St. Malo, France.
21. Object Management Group. The Common Object Request Broker: Architecture and Specification, 2.6 edition, http://www.omg.org/technology/documents/formal/corba_iiop.htm, 2001.
22. R. Rajkumar, C. Lee, J. Lehoczky, D. Siewiorek. Practical Solutions for QoS-Based Resource Allocations. Proceedings of the 19 th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998.
23. M. Satyanarayanan. Mobile Information Access. IEEE Personal Communications, Vol. 3, No. 1, February 1996.
24. M. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. IEEE Computer, May 1990, Vol. 23, No. 5.
25. M. Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. Studies of Software Design, Proc. 1993 Workshop, LNCS No. 1078, Springer-Verlag, D.A. Lamb (Ed.), 1996.
26. S. Shearin, H. Lieberman. Intelligent Profiling by Example. Proc. International Conference on Intelligent User Interfaces (IUI 2001). Sante Fe, New Mexico, January 2001.
27. B. Spitznagel, D. Garlan. A Compositional Approach for Constructing Connectors. Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA'01), Royal Netherlands Academy of Arts and Sciences Amsterdam, The Netherlands, August 2001
28. M. van Steen, P. Homburg, A. Tanenbaum. Globe: A wide-area distributed system. IEEE Concurrency, 7(1), pp 70–78, 1999.
29. Z. Wang, D. Garlan. Task Driven Computing. Carnegie Mellon University Technical Report CMU-CS-00-154, http://reports-archive.adm.cs.cmu.edu/cs2000.html, May 2000.
30. M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, pp 94-100, September 1991.