Systems Directions for Pervasive Computing

Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth,
Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello,
Steven Gribble, David Wetherall
University of Washington
one@cs.washington.edu

Abstract

Pervasive computing, with its focus on users and their tasks rather than on computing devices and technology, provides an attractive vision for the future of computing. But, while hardware and networking infrastructure to realize this vision are becoming a reality, precious few applications run in this infrastructure. We believe that this lack of applications stems largely from the fact that it is currently too hard to design, build, and deploy applications in the pervasive computing space.

In this paper, we argue that existing approaches to distributed computing are flawed along three axes when applied to pervasive computing; we sketch out alternatives that are better suited for this space. First, application data and functionality need to be kept separate, so that they can evolve gracefully in a global computing infrastructure. Second, applications need to be able to acquire any resource they need at any time, so that they can continuously provide their services in a highly dynamic environment. Third, pervasive computing requires a common system platform, allowing applications to be run across the range of devices and to be automatically distributed and installed.

1. Introduction

Pervasive computing [10, 26] promises a computing infrastructure that seamlessly and ubiquitously aids users in accomplishing their tasks and that renders the actual computing devices and technology largely invisible. The basic idea behind pervasive computing is to deploy a wide variety of smart devices throughout our working and living spaces. These devices coordinate with each other to provide users with universal and immediate access to information and support users in completing their tasks. The hardware devices and networking infrastructure necessary to realize this vision are increasingly becoming a reality, yet precious few applications run in this infrastructure. Notable exceptions are email for communication and the World Wide Web as a medium for electronic publishing and as a client interface to multi-tier applications.

This lack of applications is directly related to the fact that it is difficult to design, build, and deploy applications in a pervasive computing environment. The pervasive computing space has been mapped as a combination of mobile and stationary devices that draw on powerful services embedded in the network to achieve users' tasks [9]. The result is a giant, ad-hoc distributed system, with tens of thousands of devices and services coming and going. Consequently, the key challenge for developers is to build applications that continue to provide useful services, even if devices are roaming across the infrastructure and if the network provides only limited services, or none at all.

As part of our research into pervasive computing, we are building *one.world*, a system architecture for pervasive computing [14]. Based on our experiences with this architecture, we believe that existing distributed computing technologies are ill-suited to meet this challenge. This is not to say that discovery services [1, 2, 8] or application-aware adaptation [19] are not useful in a pervasive computing environment. On the contrary, we consider them clearly beneficial for pervasive computing applications. However, they are not sufficient to successfully design, build, and deploy applications in the pervasive computing space.

Moreover, we argue that current approaches to building distributed applications are deeply flawed along three axes, which — to express their depth — we call fault lines. In the rest of this paper, we explore the three fault lines in detail; they are summarized in Table 1. First, Section 2 makes our case against distributed objects and outlines a more appropriate approach to integrating application data and functionality. Next, Section 3 discusses the need to write applications that continuously adapt in a highly dynamic environment. Finally, Section 4 argues for a common pervasive computing platform that spans the different classes of devices. We conclude this paper in Section 5.

Problem	Cause	Proposed Solution
Objects do not scale well across large, wide-area distributed systems	Encapsulation of data and functionality within a single abstraction	Keep data and functionality separate
Availability of application services is limited or intermittent	Transparency in a highly dynamic environment	Programming for change: Applications need to be able to acquire any resource they need at any time
Programming and distributing applications is increasingly unmanageable	Heterogeneity of devices and system platforms	Common system platform with an integrated API and a single binary format

Table 1. Overview of the three fault lines discussed in this paper, listing the problem, cause, and proposed solution for each fault line.

2. Data and Functionality

The first fault line concerns the relationship between data and functionality and how they are represented. Several distributed systems, such as Legion [16] or Globe [25], are targeted at a global computing environment and have explored the use of objects as the unifying abstraction for both data and functionality. We are skeptical about this use of objects for distributed computing for two reasons.

First, objects as an encapsulation mechanism are based on two assumptions: (1) Implementation and data layout change more frequently than an object's interface, and (2) it is indeed possible to design interfaces that accommodate different implementations and hold up as a system evolves. However, these assumptions do not hold for a global distributed computing environment. Increasingly, common data formats, such as HTML or PNG, are specified by industry groups or standard bodies, notably the World Wide Web Consortium, and evolve at a relatively slow pace. In contrast, application vendors compete on functionality, leading to considerable differences in application interfaces and implementations and a much faster pace of innovation.

Second, it is preferable to store and communicate data instead of objects, as it is generally easier to access passive data rather than active objects. In particular, safe access to active objects in a distributed system raises important issues, notably system security and resource control, that are less difficult to address when accessing passive data. This is clearly reflected in today's Internet: Access to regular HTML or PDF documents works well, while active content results in an ever continuing string of security breaches [17]. Based on these two realizations, we argue that data and functionality should be kept separate rather than being encapsulated within objects.

At the same time, data and functionality depend on each other, especially when considering data storage and mobile code. On one hand, data management systems already rely on mobile code for their services. For example, Bayou propagates updates as procedures and not simply as data [23]. The Oracle8*i* database not only supports SQL stored procedures, but also includes a fully featured Java virtual machine [11]. On the other hand, mobile code systems have seen limited success in the absence of a standard data model and the corresponding data management solutions. For example, while many projects have explored mobile agents [18], they have not been widely adopted, in part because they lack storage management. Java, which was originally marketed as a mobile code platform for the Internet, has been most successful in the enterprise, where access to databases is universal [21].

The result is considerable tension between integrating data and functionality too tightly — in the form of objects — and not integrating them tightly enough. *one.world* resolves this tension by keeping data and functionality separate and by introducing a new, higher-level abstraction to group the two. In our architecture, data is represented by tuples, which essentially are records with named and optionally typed fields, while functionality is provided by components, which implement units of functionaly. Environments serve as the new unifying abstraction: They are containers for stored tuples, components, and other environments, providing a combination of the roles served by file system directories and nested processes [5, 12, 24] in more traditional operating systems. Environments make it possible to group data and functionality when necessary. At the same time, they allow for data and functionality to evolve separately and for applications to store and exchange just data, thus avoiding the two problems associated with objects discussed above.

To summarize, we are arguing that data and functionality need to be supported equally well in large distributed systems, yet also need to be kept separate. We are not arguing that object-oriented programming is not useful. *one.world* is implemented mostly in Java and makes liberal use of object-

oriented language features such as inheritance to provide its functionality.[1] At the same time, our architecture clearly separates data and functionality, using tuples to represent data and components to express functionality.

3. Programming for Change

The second fault line is caused by transparent access to remote resources. By building on distributed file systems or remote procedure call packages, many existing distributed systems mask remote resources as local resources. This transparency certainly simplifies application development. From the programmer's viewpoint, accessing a remote resource is as simple as a local operation. However, this comes at a cost in failure resilience and service availability. Network connections and remote servers may fail. Some services may not be available at all in a given environment. As a result, if a remote service is inaccessible or unavailable, distributed applications cannot provide their services, because they were written without the expectation of change.

We believe that this transparency is misleading in a pervasive computing environment, because it encourages a programming style in which a failure or the unavailability of a resource is viewed as an extreme case. But in an environment where tens of thousands of devices and services come and go, the unavailability of some resource may be the common (or at least frequent) case. We are thus advocating a programming style that forces applications to explicitly acquire all resources, be they local or remote, and to be prepared to reacquire them or equivalent resources at any time.

In *one.world*, applications need to explicitly bind all resources they use, including storage and communication channels. Leases are used to control such bindings and, by forcing applications to periodically renew them, provide timeouts for inaccesible or unavailable resources. While leases have been used in other distributed systems, such as Jini [2], to control access to remote resources, we take them one step further by requiring that *all* resources be explicitly bound and leased. Furthermore, resource discovery in *one.world* can use late binding, which effectively binds resources on every use and thus reduces applications' exposure to failures or changes in the environment [1].

This style of programming for change imposes a strict discipline on applications and their developers. Yet, programming for change also presents an opportunity by enabling system services that make it easier to build applications. *one.world* provides support for saving and restoring application checkpoints and for migrating applications and their data between nodes. Checkpointing and migration are useful primitives for building failure resilient applications and for improving performance in a distributed system. Furthermore, migration is attractive for applications that follow a user as she moves through the physical world.

Checkpointing and migration affect an environment and its contents, including all nested environments. Checkpointing captures the execution state of all components in an environment tree and saves that state in form of a tuple, making it possible to later restore the saved state. Migration moves an environment tree, including all components and stored tuples, from one device to another. Since applications already need to be able to dynamically acquire resources they need, both checkpointing and migration eschew transparency and are limited to the resources contained in the environment tree being checkpointed or migrated. As a result, their implementation in *one.world* can avoid the complexities typically associated with full process checkpointing and migration [18], and migration in the wide area becomes practical.

To summarize, the main idea behind programming for change is to force developers to build applications that better cope with a highly dynamic environment, while also providing primitives that make it easier to implement applications.

4. The Need for a Common Platform

The third fault line is rooted in the considerable and inherent heterogeneity of devices in a pervasive computing environment. Computing devices already cover a wide range of platforms, computing power, storage capacity, form factors, and user interfaces. We expect this heterogeneity to increase over time rather than decrease, as new classes of devices such as pads or car computers become widely used.

Today, applications are typically developed for specific classes of devices or system platforms, leading to separate versions of the same application for handhelds, desktops, or cluster-based servers. Furthermore, applications typically need to be distributed and installed separately for each class of devices and processor family. As heterogeneity increases, developing applications that run across all platforms will become exceedingly difficult. As the number of devices grows, explicitly distributing and installing applications for each class of devices and processor family will become unmanageable, especially in the face of migration across the wide area.

We thus argue for a single application programming interface (API) and a single binary distribution format, including a single instruction set, that can be implemented across the range of devices in a pervasive computing environment. A single, common API makes it possible to develop applications once, and a single, common binary format enables

[1] Though, for several features, including the implementation of tuples, mixin-based inheritance [4] and multiple dispatch as provided by MultiJava [7] would have provided a better match than Java's single inheritance and single dispatching of methods.

the automatic distribution and installation of applications. It is important to note that Java does not provide this common platform. While the Java virtual machine is attractive as a virtual execution platform (and used for this purpose by *one.world*), Java as an application platform does not meet the needs of the pervasive computing space. In particular, Java's platform libraries are rather large, loosely integrated, and often targeted at conventional computers. Furthermore, Java, by itself, fails to separate data and functionality and does not encourage programming for change, as discussed in Sections 2 and 3 respectively.

Given current hardware trends and advances in virtual execution platforms, such as the Java virtual machine or Microsoft's common language runtime [22], we can reasonably expect that most devices can implement such a pervasive computing platform. Devices that do not have the capacity to implement the full platform, such as small sensors [15], can still interact with it by using proxies or emulating the platform's networking protocols. Furthermore, legacy applications can be integrated by communicating through standard networking protocols, such as HTTP or SOAP [3], and by exchanging data in standard formats, such as XML.

A pervasive computing platform that runs across a wide range of devices does impose a least common denominator on the core APIs. Applications can only assume the services defined by the core APIs; they must implement their basic functionality within this framework. At the same time, a common platform does not prevent individual devices from exposing additional services to applications. It simply demands that additional services be treated as optional and dynamically discovered by applications.

As part of our research on *one.world*, we are exploring how to scale a common platform across the range of devices. Taking a cue from other research projects [6, 13, 15, 20], which have successfully used asynchronous events at very different points of the device space, our architecture also relies on asynchronous events to express control flow. All system interfaces are asynchronous, and application components interact by exchanging asynchronous events. The hope behind this design decision is that it will considerably aid with the scalability of the architecture. *one.world*'s implementation currently runs on Windows and Linux computers, and a port to Compaq's iPAQ handheld computer is under way.

5. Outlook

In this paper, we have argued that current approaches to distributed computing are ill-suited for the pervasive computing space and have identified three fault lines of existing distributed systems. First, while object-oriented programming continues to provide an attractive paradigm for application development, data and functionality should be kept separate for pervasive computing applications as they typically need to evolve independently. Second, applications need to be explicitly programmed to gracefully handle change. While this style of programming imposes a strict discipline on application developers, it also enables system services, such as checkpointing and migration, previously not available in distributed systems of this scale. Third, pervasive computing requires a common system platform, so that applications can run across (almost) all devices in this infrastructure and can be automatically distributed and installed.

We are exploring how to address these fault lines with *one.world*, a system architecture for pervasive computing. In an effort to better understand the needs of application developers, we have taught an undergraduate course that leverages *one.world* as the basis for students' projects. We are also building pervasive applications within our architecture and are collaborating with other researchers in the department to implement additional infrastructure services on top of it. Further information on *one.world*, including a source distribution, is available at `http://one.cs.washington.edu/`.

Acknowledgments

We thank David Notkin for helping us to refine our observations and Brendon Macmillan as well as the anonymous reviewers for their comments on an earlier version of this paper.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island Resort, South Carolina, Dec. 1999.

[2] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999.

[3] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. W3C note, World Wide Web Consortium, Cambridge, Massachusetts, May 2000.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '90*, pages 303–311, Ottawa, Canada, Oct. 1990.

[5] P. Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 250, Apr. 1970.

[6] P. Chou, R. Ortega, K. Hines, K. Partridge, and G. Borriello. ipChinook: An integrated IP-based design framework for distributed embedded systems. In *Proceedings of the 36th*

ACM/IEEE Design Automation Conference, pages 44–49, New Orleans, Louisiana, June 1999.

[7] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications '00*, pages 130–145, Minneapolis, Minnesota, Oct. 2000.

[8] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 24–35, Seattle, Washington, Aug. 1999.

[9] M. L. Dertouzos. The future of computing. *Scientific American*, 281(2):52–55, Aug. 1999.

[10] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges: Data-centric networking for invisible computing. In *Proceedings of the 5th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 256–262, Seattle, Washington, Aug. 1999.

[11] S. Feuerstein. *Guide to Oracle8i Features*. O'Reilly, Oct. 1999.

[12] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 137–151, Seattle, Washington, Oct. 1996.

[13] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 319–332, San Diego, California, Oct. 2000.

[14] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A system architecture for pervasive computing. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 177–182, Kolding, Denmark, Sept. 2000.

[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, Massachusetts, Nov. 2000.

[16] M. Lewis and A. Grimshaw. The core Legion object model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 551–561, Syracuse, New York, Aug. 1996.

[17] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. Wiley Computer Publishing, John Wiley & Sons, 1999.

[18] D. Milojičić, F. Douglis, and R. Wheeler, editors. *Mobility—Processes, Computers, and Agents*. ACM Press. Addison-Wesley, Feb. 1999.

[19] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint-Malo, France, Oct. 1997.

[20] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 199–212, Monterey, California, June 1999.

[21] A. Radding. Java emerges as server-side standard. *InformationWeek*, (987):121–128, May 22, 2000.

[22] J. Richter. Microsoft .NET framework delivers the platform for an integrated, service-oriented web. *MSDN Magazine*, 15(9):60–69, Sept. 2000.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, Colorado, Dec. 1995.

[24] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *Proceedings of the 8th ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, Sept. 1998.

[25] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, 1999.

[26] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.