Application-Aware Adaptation
for Mobile Computing

M. Satyanarayanan, Brian Noble, Puneet Kumar, Morgan Price

July 28, 1994
CMU-CS-94-183

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

Appeared in: *Proceedings, 6th ACM SIGOPS European Workshop*,
September 12-14, 1994 Dagstuhl Castle, Wadern, Germany.

Abstract

This paper identifies *application-aware adaptation* as an essential capability of mobile clients, and provides an overview of *Odyssey*, an architecture that supports this capability. Functionality that has hitherto been implemented monolithically must now be split between the operating system and individual applications. The role of the operating system is to sense external events, and to monitor and allocate scarce resources. In contrast, the role of individual applications is to adapt to changing conditions by using the information and resources provided by the operating system.

Research is sponsored in part by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F19628-93-C-0193, the National Science Foundation (Graduate Fellowship), the IBM Corporation, and Intel Corporation.

1. Introduction

In this position paper we put forth the view that *access to shared data in mobile computing systems is best achieved by a collaborative partnership between the operating system and individual applications.* This strategy characterizes the design space between two extremes. At one extreme, the system provides total transparency: applications are completely unaware of mobility. At the other extreme, there is no system support: each application copes with mobility on its own. It is our thesis that a collaborative partnership will better cope with the problems induced by mobility. We refer to such a strategy as *application-aware adaptation.*

2. Functional Partitioning for Mobile Computing

Data access in mobile environments is complicated by a number of factors. First, *scalability* is a concern both because of the expected ubiquity of mobile computers and because of the increasing volume of data in shared repositories. Second, we have to cope with certain *fundamental constraints of mobility:* mobile hosts tend to be resource poor because of size, weight, and power constraints; they are physically vulnerable to loss, theft and other hazards; and they have to cope with considerable variation in the performance and reliability of wireless communication. Third, data is increasingly *diverse* in format: in addition to the well-understood data formats of Unix-like hierarchical file systems and SQL databases, there is growing interest in video images, maps and other spatial images, and hypertext-like data as in the World-Wide Web.

The resource poverty, physical vulnerability and scalability concerns of mobile hosts suggest a client-server architecture, with servers being the true home of data and clients merely being caching sites. At the same time, the uncertainty in wireless communication requires clients to adapt dynamically, reducing their dependence on servers as the quality of communication degrades. The need to support diverse types of data suggests that the adaptation cannot be universal, but needs to be customized for each type of data.

These considerations lie behind our advocacy of application-aware adaptation as the paradigm of data access in mobile environments. In our model, the role of the operating system is to sense external events (such as connectivity and physical location changes), and to monitor and allocate resources (such as network bandwidth, cache space, battery power, and communication budget). In contrast, the role of individual applications is to adapt to changing conditions by using the information and resources provided by the operating system. The nature of the adaptation is specific to the application. For example, in accessing video data, degrading the display from full-motion color to black and white, and thence to slow-scan might be appropriate. But when accessing map data, the meaningful form of degradation might be to lower the resolution and minimum feature size represented. The overall goal is to minimize the need for active intervention by users to cope with the consequences of mobility. For example, it is inappropriate to require the user to use a different video display program when bandwidth drops.

3. Example: Application-Specific Conflict Resolution

A simple instance of this collaborative paradigm is already operational in the Coda File System [2, 3]. Coda uses an optimistic replica control strategy to allow updates to cached data while disconnected. Responsibility for monitoring connectivity, resynchronizing cache state upon reconnection with servers, and detecting update conflicts is vested in the Coda cache manager, *Venus.* But once a conflict is detected on a file, Venus invokes an *application-specific resolver (ASR)* and lets it resolve the conflict. If the ASR succeeds, the conflict is transparent to the user; otherwise the conflict becomes visible and has to be repaired manually.

The importance of application assistance can be seen by considering the example of a calendar management program. Suppose an executive and her secretary both make appointments while the former is disconnected. Upon reconnection, Venus detects that the file containing appointments is in conflict. But it has no knowledge of the format of file contents, nor of whether there is really a scheduling conflict. Only code specific to the calendar program can tell, for instance, that appointments for an hour each at 8am and 10am on the same day pose no problem if they are in the executive's office, while those same appointments are impossible to keep if they are in New York and San Francisco.

At the same time, the centralization of resource control provided by Venus is essential to managing cache space and network bandwidth across applications. There is also an important performance benefit during reintegration: the common case of no conflicts is efficiently handled by Venus using the syntactic approach of version comparison rather than by the more expensive semantic approach of invoking application-specific code on each cached file.

One can think of this partitioning of responsibility in terms of the end-to-end argument [4]. Correctness requires help from the end points (in this case, the applications). But efficiency and resource management considerations argue for support at lower levels of the system (in this case, Venus).

4. Odyssey: A Platform for Mobile Data Access

We are currently exploring the paradigm put forth in this position paper. The vehicle for our work is *Odyssey,* a set of Unix extensions for mobility. The extensions are both at the system call level and internal to the operating system. The Odyssey architecture is a generalization of the Coda architecture [5]. Data is stored in a shared, hierarchical name space made up of typed volumes called *tomes.* There are three components of Odyssey at each client:

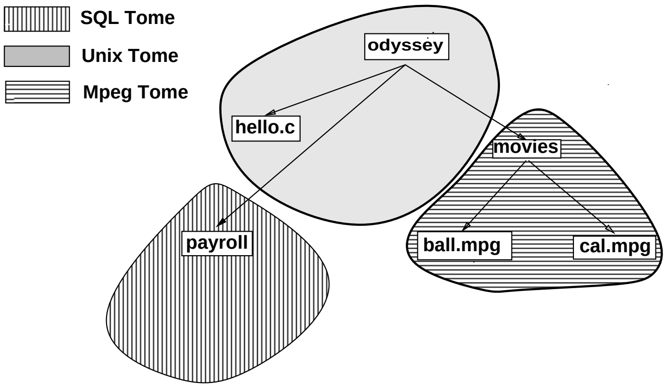
- the *Odyssey API,* a set of new system calls;
- the *viceroy,* which provides type-independent functionality and resource management; and,
- multiple *wardens,* which handle type-specific functionality and resource management.

4.1. Tomes

Tomes are conceptually similar to *volumes* [1] in AFS and Coda. They are the unit of system administration functions such as backup and disk quota enforcement. But they differ from volumes in that they carry with them the notion of type. Each tome type, referred to as a *codex,* identifies a unique combination of characteristics and policies such as naming, consistency, forms of graceful degradation, and caching strategies. Examples of codices might be "Unix", "SQL", "Mpeg video", "Grass", and so on. Servers do not have to be dedicated to particular codices — tomes of different codices can reside on the same server.

Figure 1 illustrates the Odyssey name space. Each tome is stored in its entirety on a server. As in AFS and Coda, the name space is location transparent and tomes are glued together at *mount points.* Although the root of every tome has a hierarchical name, the name space within a tome need not be hierarchical. An SQL tome, for example, might only support associative naming.

Data is typed by its location in the name space: for example, video data must be in a video tome, and so on. Although this is a larger granularity than one might like, it does obviate the need to provide individual type tags to Odyssey objects. This, in turn, preserves upward compatibility with existing data.



This figure shows a hypothetical Odyssey name space. It has three tomes, each from a different codex. The root tome is from a UNIX file codex. Embedded in it are mount points to an Mpeg tome, with root `movies`, and an SQL tome, with root `payroll`. The internal name structure of the Mpeg tome is hierarchical: a directory with two movies, `ball.mpg` and `cal.mpg`, is visible. The SQL tome, however, has no internal naming visible at this level.

Figure 1: Odyssey Name Space

4.2. Odyssey API

The Odyssey API consists of two classes of extensions to the 4.4 BSD Unix system call interface. One class, pertinent to this paper, is for *resource negotiation* and *change notification.* The other class, not discussed here, supports an abstraction called *dynamic sets* [6].

Resources in Odyssey are divided into two classes: generic and codex-specific. Examples of generic resources include disk space, network bandwidth and battery power. An example of a codex-specific resource might be the number of RPCs to a stock-quotation server, assuming each user is restricted to a certain number of calls per day.

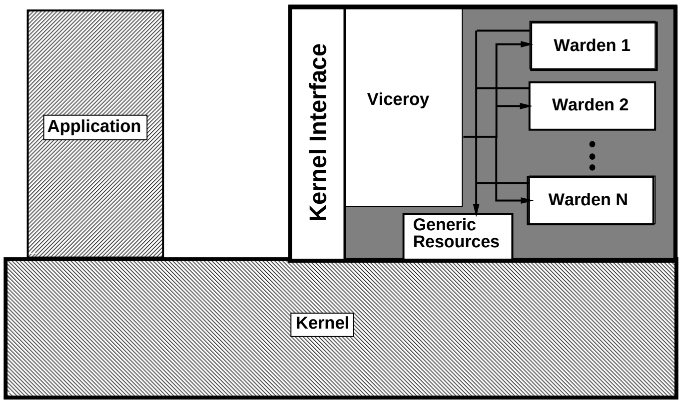
An application can negotiate and register a window of tolerance with Odyssey for a particular resource. If availability of that resource rises above or falls below the limits in the tolerance window, Odyssey will notify the application. Once notified, it is the application's responsibility to renegotiate a new tolerance window and to adapt its behavior accordingly. The request and notification interfaces are similar to those of the `signal` facility in Unix. To place a request, an application passes a resource identifier, the lower and upper bounds of the tolerance window, and the address of an upcall procedure to be invoked if the tolerance window is exceeded.

4.3. Viceroy and Wardens

The viceroy and wardens may be collectively viewed as a generalization of the Coda cache manager, Venus. Their combined role is to provide the generic and specialized functionality and cache management needed by the client. Thus their relationship is collaborative in nature, as shown in Figure 2. Applications are, of course, unaware of the existence of wardens and the viceroy; all they see is the Odyssey and Unix APIs.

The viceroy is responsible for monitoring external events and levels of generic resources, and notifying clients who have placed requests on those resources. It also provides functionality that is common to all tomes, such as authentication, tome location and system administration. Other operations are forwarded by the viceroy to the appropriate warden. Finally, the viceroy is responsible for type-independent cache management functions.

Type-specific functionality is delegated to wardens, one per codex. Each warden implements caching and naming support specific to its codex, and manages codex-specific resources. As its name implies, a warden only has a custodial relationship to the resources it manages — ultimate control of these resources always resides with the



This figure shows the internal structure of a typical Odyssey client. Applications make requests on Odyssey objects. These requests are redirected through the kernel to the kernel interface of the Odyssey cache manager. The viceroy fields each request, handles generic operations, and passes all other operations on to individual wardens. Resources, such as cache space, are loaned to wardens by the viceroy. When necessary, the viceroy notifies applications of changes in resource availability.

Figure 2: Structure of an Odyssey Client

viceroy.

The structure of Odyssey has many desirable properties. First, it provides a balanced approach to resource management. Although responsibility is delegated to the wardens, they share a common point of resource control in the viceroy. This allows the wardens to work effectively together, in contrast with current state of affairs, where resource management across data types is uncoordinated. Second, functionality is implemented where it is needed: type-specific code is contained within the wardens; resource sharing and key interfaces are in the viceroy. Finally, the system is extensible: adding a new codex and its warden are relatively straightforward.

5. Conclusion

Accessing shared data from mobile clients poses many challenges. We believe that neither a fully transparent approach, nor a purely application-by-application approach can meet these challenges adequately. Hence our approach is to expose mobility in a controlled and limited manner to applications, while reserving ownership and control of key resources to the system. This is the essence of application-aware adaptation.

The Odyssey architecture embodies this philosophy. Its goal is to be a flexible and efficient base for application-aware adaptation in mobile computing environments. An implementation of this architecture is under way, and we hope to be able to report positively on its effectiveness in the near future.

References

[1] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J.
Scale and Performance in a Distributed File System.
ACM Transactions on Computer Systems 6(1), February, 1988.

[2] Kumar, P., Satyanarayanan, M.
Supporting Application-Specific Resolution in an Optimistically Replicated File System.
In *Proceedings of the 4th IEEE Workshop on Workstation Operating Systems*. Napa, CA, October, 1993.

[3] Kumar, P., Satyanarayanan, M.
Flexible and Safe Resolution of File Conflicts.
July, 1994.
Submitted for publication.

[4] Saltzer, J.H., Reed, D.P., Clark, D.D.
End-To-End Arguments in System Design.
ACM Transactions on Computer Systems 2(4), November, 1984.

[5] Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.
Coda: A Highly Available File System for a Distributed Workstation Environment.
IEEE Transactions on Computers 39(4), April, 1990.

[6] Steere, D.C., Satyanarayanan, M.
A Case for Dynamic Sets in Operating Systems.
June, 1994.
Submitted for publication.