Mobile Data Access

Brian D. Noble

May 11, 1998

CMU-CS-98-118

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
M. Satyanarayanan, Chair
Daniel P. Siewiorek
Hui Zhang
Randy H. Katz, University of California, Berkeley

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA), Air Force Materiel Command, USAF under agreement numbers F19628-96-C-0061 and F19628-93-C-0193, the Intel Corporation, the National Science Foundation, AT&T, and Lucent Technologies.

Keywords: Mobile computing, adaptive systems, application-aware adaptation, agility, fidelity, trace modulation, Odyssey, UNIX

Abstract

Mobile devices and wireless networks are becoming more powerful and affordable, leading to the growing importance of mobile data access. Unfortunately, mobile environments are inherently turbulent; the resources available to mobile clients change dramatically and unpredictably over time.

This dissertation puts forth the claim that clients must *adapt* their behavior to such turbulence by trading quality of fetched data for performance in fetching it. Such adaptation is best provided by *application-aware adaptation* — a collaboration between the operating system and its applications. In this collaboration, the system is responsible for providing the mechanisms for adaptation, while applications are free to set adaptive policies.

The dissertation next describes the design and implementation of *Odyssey*, a platform for mobile data access. This discussion focuses on the mechanisms provided by the system, the architecture comprising those mechanisms, and the application programming interface from which applications construct adaptive policies. The dissertation then presents three applications that make use of these mechanisms: a video player, a web browser, and a speech recognition system. These applications adapt their behavior to changes in available network bandwidth.

There are three questions to ask of this prototype and its applications. First, how *agile* can the prototype be in the face of changing network bandwidth? Second, does adaptation to substantial changes in bandwidth provide benefit to individual applications? Third, is the collaboration between the system and applications necessary when several applications are run concurrently?

These questions cannot be answered simply by subjecting the prototype to a real wireless network. Such networks provide neither stable nor repeatable performance, and hence are not suitable for comparative evaluations. Instead, the prototype is evaluated using *trace modulation*. This technique allows one to capture the performance characteristics of a wireless network over a short period of time, and reliably recreate that performance in an otherwise live system. Evaluating the system under modulation shows that Odyssey has good agility with respect to changes in network bandwidth, that individual applications can benefit from adaptive strategies, and that the system's involvement in adaptation is crucial for concurrent applications.

Acknowledgements

The common perception of a dissertation is as a solitary work, produced by a lone scholar privately toiling away. Nothing could be further from the truth. I could not imagine undertaking a task this substantial without the help, support, and guidance of many people.

First and foremost, I thank my advisor, Satya, for his guidance and counsel. His supervisory style was an excellent match for me. He always set optimistic goals, and offered praise when they were met and understanding when they were not. In addition to his superb technical guidance, he is an excellent writer and a better editor; the importance of such skills can not to be overstated. I consider it an honor to have worked with and known him.

Despite their incredibly busy schedules, the members of my committee were always available for feedback and advice. Dan Siewiorek offered my first taste at building truly mobile devices through his wearable computing course, and offered solid guidance throughout the course of my research. I have known Hui Zhang the longest of any of my committee, dating from our time together at Berkeley. In addition to his advice, the opportunity to watch him develop as a new faculty member has provided me with an excellent model for my own career. My external member, Randy Katz, was instrumental in the design and construction of trace modulation, and always available for feedback and advice.

I am in the unique and fortuitous position of having worked with all of Satya's previously graduated students, as well as those who are now supervised by him. Having six successful role models — Jay Kistler, Puneet Kumar, Qi Lu, Lily Mummert, David Steere, and Maria Ebling — has been an incredible benefit. Special thanks go to Jay, who patiently introduced me to the Coda code, and David, who as an officemate was my closest confidant for more years than either of us cares to admit.

As for the four who remain, I could not have completed this dissertation without them: Jason Flinn, Dushyanth Narayanan, Eric Tilton, and Kip Walker. Their help in building the code and keeping me honest is greatly appreciated. Morgan Price, an early member of the group, helped me build the foundation upon which the current system is based. I am sure that all five of them will continue to meet with great success.

Past and present group members, including Bob Baron, Peter Braam, Henry Pierce, and Josh Raiff, were all instrumental in keeping me sane and productive. Likewise, members of the CMU facilities staff — particularly Mike Accetta, Gary Aranyos, Barry Eisel, Jeliza Patterson, and Mark Puskar — helped keep my computing environments up and running. Bob Baron deserves special recognition; his technical talents, particularly with device drivers, saved me months of work.

My friends here at CMU, both past and present, are far too numerous to even begin listing. The Cache Cows, NP Completions, and Pittsburgh Hash House Harriers all kept me moderately healthy, and completely entertained. The Baseball Consortium, especially Mark Stehlik, who ran it, and Bob Wheeler, who went to both Pirates and Steelers games with me, helped me to spend more money on sporting events than I could reasonably afford. With Wayne Sawdon and the Brew Crew, I practiced alchemy combining chemistry and biology. The administrative staff, particularly Sharon Burks, Maria Fischer, Patty Mackiewicz, and Terri Stankus, all helped remind me that there is indeed life outside of computer science.

While I deserve the blame for this dissertation, perhaps my father deserves the most credit. He was the first person to suggest that I might be able to earn a Ph.D., if only I was willing to work hard enough. I doubt that I could have found the strength to do so without my father's and mother's constant support, encouragement, and instruction.

Last, but most importantly, I wish to thank my wife, Maureen. Her unfailing love, support, and companionship has made the process of writing this dissertation tolerable. In the many times I have deeply doubted myself and my work, she has provided the encouragement to push on. I could not have reached this point without her, nor can I imagine a better soul with whom to face future challenges.

Contents

List of Figures

Chapter 1

Introduction

Mobile devices are becoming increasingly prevalent, as are the wireless networks that connect them to the outside world. Together, these trends are driving the increasing importance of *mobile data access,* the ability to access data from anywhere at any time. Unfortunately, the environments in which mobile devices operate are very turbulent; the resources available to a mobile host change dramatically and unpredictably. In response to these changes, mobile devices are forced to *adapt* their behavior.

This dissertation puts forth the thesis that such adaptation — the trading of data quality for resource consumption — is best provided through a collaboration between the system and its applications. In this collaboration, called *application-aware adaptation,* the system provides the mechanisms of adaptation, while the applications are free to specify policy.

The dissertation establishes this claim through the design and implementation of *Odyssey*, a system that supports application-aware adaptation. Three applications, a video player, a Web browser, and a speech recognition system, have been modified to make use of these adaptive services. An evaluation of Odyssey and its applications demonstrates both the feasibility and utility of application-aware adaptation.

This chapter begins with a brief scenario introducing the challenges presented to mobile clients. It then gives a statement of the thesis, and presents the steps required to substantiate the thesis. Finally, it concludes with a road map to the remainder of the dissertation with an eye toward how it supports the individual claims.

1.1 A Mobile Scenario

A tourist visiting a city carries with him a wearable computer. This computer has access to a variety of networks that differ in bandwidth, coverage, cost, and reliability. The higher-bandwidth alternatives are more sensitive to fading and signal loss as the user moves in and out of the radio shadows of buildings.

As he walks, the user interacts with his computer through spoken commands; he receives output through a head-mounted display or synthesized speech. The *speech software exploits remote compute servers when connected, but is capable of degraded interactions using a tiny vocabulary when disconnected. One application provides a video narration of local history, that is delivered from a remote server. Another application is a Web browser that can respond to queries about the local environment.*

The client monitors resources such as bandwidth, CPU cycles, and battery power, and decides how to best exploit them. For example, when high-bandwidth connectivity is lost due to a radio shadow, the client detects the change and reacts to it. The video application begins to conserve bandwidth by lowering the frame rate, while the Web application displays degraded versions of large images. When the user emerges from the radio shadow, the client detects the improvement in bandwidth, and the applications revert to their original behaviors.

Although the user is aware of changing application behavior during his walk, he does not have to initiate adaptation or be involved in its details. Rather, he can delegate these decisions to the client, confident that reasonable tradeoffs will be made.

While illustrative, this scenario is only one of many presenting similar problems. For example, an emergency response team entering a disaster site would encounter areas where the in-place mobile infrastructure is no longer available. Even in mundane situations, variations in service are unavoidable. At Carnegie Mellon some campus buildings are covered by a 2 Mb/s wireless network, while users in other locations must resort to some alternative wireless technology; moving between these technologies yields significant variation.

There is one thread common to all of these scenarios: in each, the environment in which a mobile must operate is turbulent. This recognition brings to light several important questions. Are mobile environments significantly more volatile than static environments? Does this volatility justify adaptation on the part of mobile clients? How best can one provide such adaptation? What properties should such an adaptive system have?

1.2 The Thesis

The thesis directly answers the questions raised above:

Mobile environments are inherently turbulent, requiring clients to adapt. This adaptation is best provided for diverse, concurrent applications through a simple, efficient, and agile collaboration between the system and its applications.

1.3 Road Map for the Dissertation

This dissertation establishes the thesis in the following steps:

- It argues that mobile systems are more volatile than their static counterparts, and that this will not change despite the exponential rate of improvement in hardware capability. This argument is based on first principles and economic necessities.
- It argues that this volatility requires some form of adaptation on the part of mobile clients. It defines that adaptation as the trading of *quality of data* for *performance* and *resource consumption* that occurs as a reaction to changes in the *environment* of the client.
- It introduces a taxonomy of adaptive systems based on the degree to which the applications and the operating system of a mobile client are responsible for making adaptation decisions. It argues that, in the presence of diverse and concurrent applications, such decisions must be a collaboration between the system, which is best positioned to monitor the environment, and applications, which are best positioned to specify adaptation policy. This collaboration is called application-aware adaptation.
- It presents the design and implementation of Odyssey, a small set of extensions to a standard operating system that provide a simple, efficient API supporting application-aware adaptation. It also presents the design and implementation of three applications that have been modified to adapt to changes in available network bandwidth.
- It introduces a novel experimental method used to quantify agility, and adaptive systems generally. Borrowing the technique of *transient response analysis* from control systems, an adaptive system is subject to simple, idealized changes in available resources, called reference waveforms. In order to carry out such experiments, a test system is augmented with *trace modulation*, a system capable of delaying or dropping all packets to or from a host based on a simple network model. Parameters to this model can be generated synthetically, producing reference traces, or empirically, allowing the faithful reproduction of the performance of a real wireless network.
- It presents the result of experiments that demonstrate three things. First, the Odyssey prototype is sufficiently agile to track changes in bandwidth within a few seconds. Second, adaptive application strategies are superior to static ones in the face of changing network bandwidth. Third, system involvement in adaptation is critical in helping applications meet their adaptation needs.

The rest of this dissertation comprises seven chapters. Chapter 2 examines the constraints on mobile systems and the infrastructure that supports them. These constraints render mobile environments more volatile than their static counterparts. It then presents the need for diverse, concurrent applications on mobile hosts and argues that, in the context of mobility, such applications must adapt their behavior to changes in their environments by trading data quality for resource consumption. The chapter then presents the taxonomy of adaptation and the case for application-aware adaptation. It concludes with an introduction to the concept of agility — the speed with which an adaptive client reacts to some change in its environment.

Chapter 3 presents the detailed design of the Odyssey prototype. It begins with a set of guiding principles, and identifies the historical context from which the design borrows many facets. It then provides an overview of the major components of the design and presents each of them in turn.

The implementation of Odyssey is detailed in Chapter 4. The implementation effort was driven by a specific set of goals. The chapter begins with these goals, and then presents some background details of the host operating system, and proceeds to describe the implementation of each of the basic components. It concludes with a description of three adaptive applications: a video player, a web browser, and a speech recognition system.

Wireless networks, by their very nature, present performance that is both complex and irreproducible. This lack of experimental control requires some environment in which to conduct experiments other than a live, wireless network. To address this problem, Chapter 6 presents the design, implementation, and evaluation of *trace modulation*. Trace modulation allows network traffic to be delayed or dropped in accordance with a simple, time-varying network model yielding reproducible performance.

Chapter 7 presents three experiments that evaluate the prototype and its applications. The first experiment measures the limits to agility imposed by the system itself. The second demonstrates the benefit that applications obtain from adaptive strategies over similar, static ones. Finally, the third experiment confirms that system involvement in application-aware adaptation is critical in supporting concurrent, competing applications.

Related work is presented in Chapter 8. Chapter 9 concludes the dissertation with a summary and the identification of key contributions. Finally, the chapter addresses the possible avenues of further research stemming from this work.

Chapter 2

Characteristics of Mobile Systems

To motivate the design of Odyssey this chapter introduces a set of characteristics that are intrinsic to mobile systems. These characteristics demand that mobile systems provide some form of *adaptation* to changes, both to the environment in which they operate as well as in their demands of that environment.

The chapter begins by presenting a set of constraints placed on mobile systems, but not their fixed counterparts. It argues that these constraints, *scarcity of local resources*, *variability of the supporting infrastructure*, and *poor security and robustness*, are inherent to mobile systems and will not be eliminated by technological progress.

The chapter then argues that mobile hosts must support concurrent operation of applications with diverse data and resource needs. The resulting variations in demand for resources, together with the variation in the already scarce supply with those resources, lead to the requirement that mobile hosts adapt to these changes.

The chapter concludes with a discussion of adaptation in mobile systems. It first defines the sense in which these systems adapt to change: by trading resource usage for data quality. It then examines how adaptation decisions might be partitioned between system and applications, and argues that only a *collaborative partnership* can meet the needs of mobile computation. Finally, it addresses the issues in comparing adaptive systems.

2.1 Constraints on Mobile Systems

The design of mobile systems is driven by two sets of constraints. First, a mobile host's local resources and security are limited in comparison to fixed systems of similar costs. Second, the infrastructure supporting mobile systems is highly variable, especially in comparison to that supporting more static deployments. These constraints are inherent to mobile systems; they will not be eliminated by technological advances.

2.1.1 Constraints on Mobile Devices

There are two key constraints on mobile devices in comparison to their fixed counterparts; they are resource-poor, and they are less secure. Resource paucity is due to the additional design goals for mobile hardware that do not apply to immobile machines. Lowered security arises from the very nature of mobility.

Resource limitations spring from three requirements placed on the design of mobile hardware: low power consumption, light weight, and small physical size. These three requirements are in addition to the usual constraints on hardware design. Thus, in comparison to a similarly-priced machine without these design considerations, a mobile machine will always suffer from scarce local resources. Such poverty applies universally: to disk size, physical memory, processing power, memory cache size, and screen size.

Resource constraints are inherent to mobile hardware; technological progress will not erase the gap between mobile and immobile machines. While such progress will increase the resources available to a typical mobile host, those same advances will be brought to bear on desktop systems. Since the latter need pay little or no concern to size, power, and weight, they will continue to enjoy an advantage in resources provided at fixed cost.

Limits in security and robustness are inherent to the very nature of mobile devices. Because they are designed to be easily transported, they are more prone to loss or theft. They are more likely to be dropped and damaged than desktop systems, and often are operated in an environment less accommodating than the typical office building.

Resource and security constraints imply that mobile devices should not be the true home of data. Resource limitations make it difficult to keep all data of interest on a mobile device, and the risk of loss is substantial. Rather, data should be kept primarily on servers with plentiful resources and stronger security. Mobile devices then act as clients of these servers, accessing and manipulating data from them.

2.1.2 Constraints on Mobile Infrastructure

The infrastructure supporting mobile clients — network connectivity and remote resources — will be diverse, with areas of high and low concentration. This is due to the *overlay argument*, which was first applied by Katz and Brewer [37] to wireless networks. This section recounts their basic argument, and then broadens it to apply to elements of a mobile infrastructure generally.

Mobile devices must rely on wireless networks for connectivity while in transit or outdoors, and often will take advantage of them while in-building. Such networks tend to have lower bandwidth and longer latencies than their wired counterparts. Their lower performance is also highly variable, due to physical effects such as multi-path interference. Due to the scale of multi-path effects, seemingly insignificant changes to position may have large effects on performance. Environmental interference, which also effects wireless performance, changes both over time and through space; such changes are unpredictable, and can be large. This variance may render connectivity, at any speed, sporadic at best.

The variance of a single wireless network is likely to be only a small part of the variance in connectivity seen by a mobile client. The use of overlay networks, where different areas are served by wireless devices of very different quality, dramatically increases such variance. Physical and economic considerations render the use of these networks unavoidable in support of mobility.

Mobile clients operate in a much wider set of places than their immobile counterparts. This requires a much broader wireless infrastructure than that necessary for wired networks. Further, the presence of a particular mobile client in any one place is typically fleeting, complicating the capacity planning of a wireless infrastructure based on *a priori* knowledge of demand.

It is unrealistic to expect to provide all locales with a universal, high-quality, wireless network. Given the scope of such a network, and the unpredictability of the demands placed upon it, even providing modest support in all places would be prohibitively expensive. Any low-cost, global solution will necessarily be low-bandwidth, high-latency, or both. A given base station has a limited spectrum available to it, and would require some number of expected users in its service area to justify its deployment. Areas likely to be sparsely if at all populated by mobile nodes will have few base stations, and therefore little or no connectivity to support them; there will always be such areas of poor connectivity.

Such coverage would be inadequate in an urban environment, where the concentration of mobile users justifies a larger wireless investment. For example, the downtown area of a city might be served by a cellular network with re-use of available bandwidth between cells covering a few blocks each. In individual buildings with high populations of mobile nodes, one might even imagine a per-office cellular network, where each office has the full bandwidth of a base station available. However, due to the unpredictable nature of mobility, even such well-supported locales may see transient demand beyond that for which they were designed.

The intermittent nature of mobile networks implies that while mobile clients should not be the true home of data, neither can they be continuously dependent on servers for support. Mobile clients must be capable of operating disconnected, perhaps for extended periods of time. This eliminates systems that assume constant connectivity such as Wit [84] and the Infopad [70]. While such systems provide interesting in-building or campus-area solutions, they do not apply to wider use.

Systems such as the Pilot [80] are designed to operate isolated almost exclusively, with only infrequent connection to a network. However, taking advantage of connectivity whenever possible is to the client's advantage. Both the commercial and research worlds have turned to this often-connected model for applications such as package tracking, the Coda File System [67], and Bayou [77]. These systems use bandwidth when it is available, but can cope with disconnection when it is not.

The overlay argument easily extends to mobile infrastructure in general. Consider printer availability. When visiting a university computer science department, there are a variety of printers locally: large-format, high-resolution, color, and so on. In an airport, there may be only a fax machine or perhaps a low-quality printer available. The concentration of users in a given area will drive the degree of infrastructural support.

This highly variable infrastructure can have a severe impact on mobile applications and users. As a mobile client's circumstances change, the resources available to it will change as well; such change is unpredictable and can be dramatic. The client must react to decreased resource availability, and should take advantage of increases. Such reaction, adapting to changes in the supply of resources, is widely recognized as central to the support of mobile computing [19, 23, 66, 78, 85]. This notion of adaptation is defined in Section 2.3.

2.2 Demands on Mobile Clients

The demands placed on a mobile client are a function of the applications run by that client. There are two broad application characteristics that must be supported by such a client, application diversity and concurrency. These two requirements form the basic needs of a platform for mobile computing, and further motivate the need for such a platform to be adaptive.

2.2.1 Application Diversity

Mobile clients operate in a wide variety of locations, and therefore serve a wide variety of needs. For example, consider a salesperson who travels to customer sites. When arriving in a new city, the salesperson needs information about contacts in that city as well as restaurants, weather, and the like. While at the customer site in that city, the salesperson will need access to company resources such as inventory, pricing, and shipping information. The limited resources available to a mobile host, combined with the time-varying, shared nature of this data, suggests that a mobile node will have to interact with remote services.

A mobile user is likely to expect a rich set of such services — not just text, but also maps, video, audio, and so forth. In other words, data used by a mobile client is diverse. This is not strictly a function of mobility, but rather a general trend.

This data may be stored in one or more general-purpose repositories such as file servers, SQL servers, or Web servers. Alternatively, it may be stored in more specialized repositories such as video libraries, query-by-image-content databases, or back ends of geographical information systems. These servers and their clients have substantial semantic knowledge about the data items on which they operate. In many cases, such knowledge is necessary for efficient handling of a particular data type. For example, it is useful to know that video frames need not be resent if they are lost, since they will arrive too late to be of use. In contrast, database updates must be sent reliably.

Each distinct type of data places different demands on the mobile host's resources. For example, video data is relatively simple to render, since it uses compression methods designed for simple decoding. However, even with compression, video is expensive in bandwidth. Map data, on the other hand, is often entirely abstract, and rendering it into an image is computationally expensive, but the data itself is small and easily shipped.

2.2.2 Application Concurrency

The ability to execute multiple, independent applications concurrently is vital. Although this ability is taken for granted on desktop systems, there continues to be skepticism about its value in mobile clients. This skepticism is fueled by the popularity of devices such as the Pilot and other PDAs [5], which execute only one application at a time.

Despite the success of such devices, it seems clear that many mobile users will find it valuable to run background applications in addition to the foreground application that dominates their attention. For example, an information filtering application may run in the background monitoring data such as stock prices and alerting the user as appropriate. As another example, an application used in emergency response situations may monitor physical location and motion, and prefetch damage-assessment information for the areas to be traversed shortly.

The need to run applications with diverse resource needs concurrently suggests that mobile clients must be general-purpose computing engines, roughly equivalent to current desktop systems. Concurrent applications also place unique demands on a mobile client, which is already resource-poor. To mediate between conflicting resource demands, there must be some central point of control; without it, no opportunity exists to arbitrate between, and optimize for, multiple applications.

In the literature, adaptation has been motivated by variations in the supply of resources to a mobile client. However, varying demand for those resources, generated by diverse, concurrent applications also influences adaptation decisions.

2.3 Adaptation

The variable supply of resources, as well as the differing demands on them, suggest that the client must adapt to these changes. However, this broad notion of adaptation requires definition. In what sense does a mobile system adapt? Which parties in the system are responsible for adaptation decisions?

This section first defines adaptation as the trading of resources, either for other resources or for some *measure of quality* of accessed data. The section then explores models of adaptation — ways of partitioning adaptation decisions between system and application, and explores which of these models most effectively supports the demands outlined in Section 2.2. The section concludes with a discussion on evaluating adaptive systems through the property of *agility*.

2.3.1 The Nature of Adaptation

Mobile clients, through the constraints of mobility and the nature of the applications running on them, experience vast swings in the supply of and demand for resources. As the resources available to mobile applications change, they may wish to change the way in which they access data to either consume less of some newly-limited resource, or take advantage of a sudden abundance.

One way to change resource consumption is to trade one resource for another. For example, compressing data before shipping spends processing power to save bandwidth. A similar trade-off between computation and bandwidth is to ship only deltas of slowly changing data rather than re-ship the entire item with each change. Such a scheme is used in Banga's optimistic Web deltas [8].

Techniques that trade resources for one another are transparent to applications and users. The data delivered to them is the same, whether or not the trade-off is made. However, such trade-offs may not be sufficient, particularly on resource-poor mobile hosts; there may not be enough of any resource to provide adequate service without some degradation in the quality of data. The remainder of this section defines first what is meant by *full quality*, and then a notion of degradation.

For any data item, there is a representation of that item that is the most current, consistent, and detailed. For example, a full color, full frame-rate video stream that has been compressed by a lossless algorithm would be such a representation. We call such a full-quality representation the *reference copy* of that data item. The reference copy might not be physically stored on any particular server; it may be a composite of many different replicas, or it may exist only in the abstract because the item is generated on demand by a server.

When resources are sufficient, a mobile client may be able to operate with data indistinguishable from its reference copy. If some resources vary only slightly, the client may be able to trade other resources to maintain use of a reference-quality copy. However, when resources become so scarce as to make such high quality impossible, some degradation will be inevitable.

We define the *fidelity* of a data item presented on a mobile client as the degree to which the quality of the presented item matches that of the reference copy. Since there may be many factors that make up the quality of a particular data item, fidelity has potentially many different dimensions.

One dimension of fidelity that applies to all data items is *consistency*. Systems such as Coda, Ficus [59], and Little Work [29] cope with mobility solely by relaxing the consistency of files in a file system. Bayou applies similar techniques to the very different consistency model of databases.

These systems expose potentially stale data when network connectivity is poor or nonexistent, and allow conflicting updates that would not occur in the presence of a stronger consistency model. For example, when Coda clients are disconnected, they read and update only replicas of files that are cached locally. If a server replica is updated, the client does not see the update, and may potentially allow a conflicting update. Stale files are not replaced, nor are conflicts detected, until the client is re-connected. When connectivity is present, but too weak to maintain strict consistency, the window of vulnerability for stale reads and conflicting updates widens adaptively with network speed [50].

Since the quality of data is type-dependent, other dimensions of fidelity are also dependent on the type of data to be degraded. For example, video data has at least two dimensions in addition to consistency: frame rate and image quality of individual frames. One can reduce bandwidth requirements by reducing frame rate, while changing the compression of individual frames changes both bandwidth and computational requirements. Topographical maps and other spatial representations have dimensions of feature sets and minimum feature size. For example, one can exclude all features other than rivers and roads, or only examine roads which are divided highways or larger. For telemetry data, appropriate dimensions include sampling rate and maximum reporting delay.

These dimensions of fidelity are natural axes along which mobile clients can adapt resource usage to fluctuating supply and demand. By choosing to change one or more dimensions of fidelity in accessing or storing data, the client in turn can change its consumption of resources. Furthermore, degrading along different dimensions may conserve different resources. By taking into account what dimensions of fidelity are important to the task at hand, as well as the availability of various resources, the client can make informed decisions on how to adapt data access.

2.3.2 Models of Adaptation

What party is responsible for making adaptation decisions? Should responsibility lie with the operating system, the applications, or some combination of the two? This section explores each of these three possibilities, and argues that only a collaborative approach between system and applications can support the demands of concurrent, diverse applications.

Application-Transparent Adaptation

In the first model of adaptation, the system is wholly responsible for adapting to changes in the supply of and demand for resources. This model, called *application-transparent adaptation*, is embodied in systems such as Coda and Bayou. The system automatically handles changes in connectivity between hosts, and transparently decides when to propagate updates or invalidate and re-fetch stale data. Individual applications have no say in how to make use of available bandwidth, though applications in either system can provide specific functionality, such as conflict resolution.

Systems which perform application-transparent adaptation provide three important benefits: they allow legacy applications to run unmodified, they do not complicate the programming model for applications, and they provide a central point of resource control. This last benefit is a consequence of making adaptation decisions at a single place.

Despite these benefits, application-transparent adaptation does not adequately support application diversity. Specifically, it cannot support two applications that may wish to make different adaptation decisions for the same data. Since the system makes decisions in isolation, only one decision can be made for a given data item in a given situation.

As an example, consider two applications that operate on the same piece of video data: a video player and a scene editor. The player's primary goal is to preserve correspondence between movie time and real time, while secondarily maintaining quality. In times of plentiful resources, the player can meet both goals. When bandwidth is scarce the player may reduce frame quality and drop frames to maintain the pace of the movie.

The scene editor, in contrast, must fetch every frame in order to provide precise cuts and splices in the final product. In times of scarce bandwidth, it is willing to fetch frames late in preference to dropping them.

It is hard to see how any single policy can adequately service both of these applications' needs, even though they are accessing exactly the same data. Thus, regardless of the system's decision to change the fidelity of the stream it is retrieving, either the player or the editor – and quite possibly both – will not be satisfied. Since adaptation decisions are made without regard to an application's needs in application-transparent systems, such systems cannot possibly support diverse applications.

Laissez-Faire Adaptation

At the opposite end of the spectrum, applications are solely responsible for coping with the consequences of mobility. This approach, referred to as *laissez-faire adaptation*, has been taken by commercial software such as Eudora [60]. More recently, it has been pursued by research systems such as McCanne's RLM [44] and Cen's video player [31]. In such systems, applications monitor the availability of resources, and make their own adaptation decisions in isolation of other applications or the system.

The *laissez-faire* approach provides two substantial benefits. First, no system support is required — an essential attribute of commercial systems where the operating system is a fixed commodity. Second, applications get precisely the adaptation behavior they want; no approximations are necessary.

However, the *laissez-faire* approach, without a single point of resource control, does not support application concurrency. Applications, by virtue of being external to the system, are not well-positioned to monitor the availability of resources; what they see individually is often not representative of the machine as a whole. Furthermore, since each application reacts to changes in resource availability independently, two such applications running concurrently are likely to perform poorly.

As an example, consider two video players with *laissez-faire* adaptation, running concurrently on a mobile client. In reaction to a small but significant increase in bandwidth, both applications are likely to try to increase fidelity, even if there is only enough bandwidth for one to do so successfully. Without some amount of central coordination, it is unlikely that these two video players can seamlessly coexist.

Application-Aware Adaptation

The middle ground between these two extremes is a collaborative effort between system and applications. The nature of this partnership is a consequence of end-to-end considerations [63]. The system is best positioned to know what is available to the mobile client. Thus, is responsible for monitoring resource availability, enforcing resource allocation decisions, and optimizing the use of client-wide resources. An individual application, however, is the only party which can know fully what its own needs are. Hence, an application must be informed by the system of significant changes in the availability of resources, and react to those changes in whatever way it sees fit.

This division of responsibility directly addresses the issues of application diversity and concurrency. Diverse applications are accommodated by allowing applications to determine how changing resource availability should affect fidelity levels. Concurrent applications are supported by allowing the system to retain control of resource monitoring and arbitration. Application-aware adaptation is the only adaptation model that can support the sort of mobile computing envisioned in Section 2.2.

2.3.3 Comparing Adaptive Systems

Given two adaptive systems — both of which support concurrent, diverse applications — how does one compare them? There are two issues at hand. First, how long does it take each system to make an adaptation decision? Second, when that adaptation decision is made, is it the right one?

The latter question is one of policy; the correct adaptation decision depends on several things, not the least of which is user preference. The individual application goals, combined with the overall goals of the machine, dictate which of many possible adaptation decisions is the best. The former question, the time to react to change, is measured by a metric called *agility*.

Sound adaptation decisions require accurate and timely knowledge of resource availability, and quick arrival at the correct decision given that knowledge. Ideally, a mobile client should always have perfect knowledge of current resource levels. In other words, there should be no time lag between a change in the availability of a resource and its detection. Further, if this change is sufficient to warrant a change in client behavior, that too should be accomplished without delay.

Of course, no physical system can meet this ideal. The best one can hope for is to build close approximations through good design and engineering. Agility is the property of an adaptive system that measures the speed and accuracy with which it detects and responds to changes in resource availability. When changes are large and erratic, only a highly agile system can function effectively. In more stable environments, less agile systems may suffice. Agility thus also determines the most turbulent environment in which a system can function acceptably.

Agility is not a simple scalar metric; it is a complex property with many components. One source of complexity is differing sensitivity to changes in different resources. For example, a system may be much more sensitive to changes in network bandwidth than to changes in battery power level. This could be due to fundamental limits in the ability to measure battery life, or something as simple as checking bandwidth more frequently than power.

Another source of complexity is differing origins of changes in resource availability: changes in supply or demand. For example, consider network bandwidth. The bandwidth available to a mobile client might change because it has moved from a well-served locale to a more tenuously supported one. The resulting drop in bandwidth is a change in supply. In contrast, an application on the client might suddenly increase the amount of data it is attempting to fetch over the network. From the perspective of the system, this is a change in the demand for bandwidth. Since different mechanisms may be involved in detecting these two different causes of change, it may be necessary to distinguish supply-side and demand-side components of agility.

One can characterize the core adaptive system as a collection of reactive components, two components per resource. One of these components measures changes in the supply of that resource, the other measures change in demand. Viewed in this way, the agility of each component is a function of its *rise time*; the time it takes to recognized some particular change in resource availability. The goal of maximal agility is served by providing components with minimal rise times.

The goal of maximizing agility is pursued at the sacrifice of *stability*, the ability of the system to ignore transient changes . Clearly, this will not be acceptable in all situations. However, the core system must be as agile as possible, as it limits the agility of all applications run on that system. When stability is required, it should be provided in the context of individual applications. This notion is analogous to the construction of electrical circuits for amplification. The core of the circuit — the *operational amplifier* — should be constructed to provide infinite gain; the surrounding circuit adds what stability may be required by the particular task at hand.

2.4 Summary

This chapter presents three intrinsic constraints on mobile systems: they are resource-poor, they are subject to heightened security and robustness concerns, and their supporting infrastructure is highly variable. These constraints, particularly the last one, require that mobile systems somehow adapt to their environments. This adaptation takes the form of trading quality of data for resource consumption; more formally, it is trading fidelity for performance.

The nature of this adaptation is guided by two key requirements of mobile systems. First, they must support a diverse range of applications, with potentially different adaptive needs. Second, they must support concurrent, competing applications. These two requirements lead to providing adaptation as a collaborative approach between the system — which monitors and controls resources — and applications, which set adaptation policy. If the system were solely responsible for adaptation, then diverse applications could not be effectively supported. Likewise, an application-only approach would not effectively support concurrent applications.

Finally, the chapter addresses the issue of comparing adaptive systems. There are two axes along which such comparisons are made. First, does the system make correct adaptive decisions? Second, does the system arrive at these decisions quickly in the face of resource changes? The former is a matter of policy. The latter is a quantitative measure called agility.

To operate in a turbulent environment, a mobile system must be highly agile to change in demand for and supply of the resources making up that environment.

Chapter 3

Design

This chapter presents the design of Odyssey, with an emphasis on the provision of agile application-aware adaptation. The chapter begins by presenting the background to Odyssey's design. This background consists of the principles guiding the design of Odyssey, as well as the systems from which it borrows design elements.

The chapter then presents an overview of Odyssey's architectural components. It does so by outlining how an adaptive application would make use of these components during the course of its execution. After this brief introduction, the chapter presents each of these architectural components in more detail. It concludes with a summary of key points from the chapter.

3.1 Background

Odyssey's design was guided by a small set of principles, and influenced by the systems that came before it. This section first describes these guiding principles and how they affect Odyssey's design. It then describes what design elements were borrowed from Odyssey's philosophical ancestors, AFS [28] and Coda.

3.1.1 Guiding Principles

There are three principles that influenced the design of Odyssey. The first principle is *minimalism*. Rather than using a clean-sheet approach, Odyssey was designed as a minimal set of extensions to an existing system. This was done with the goal of understanding which abstractions would have to be added to a stock operating system to provide application-aware adaptation.

This goal of minimalism is pervasive, and affects many design decisions throughout the system. Whenever possible, Odyssey makes use of existing features in the system rather than invent new ones. Any new functionality or mechanisms that have been added to the system have been designed to be consistent with existing idiom. This reduces the number of fundamentally new abstractions that an application must accept in order to make use of application-aware adaptation.

The base system in which Odyssey has been built is NetBSD, a variant of the 4.4 BSD UNIX operating system [46]. NetBSD source code is publicly available without encumbrance, thus allowing free distribution of derivatives. The 4.4 BSD family of operating systems are very similar. They have a close, common ancestor in the 4.4BSD-Lite release, and have collectively met with wide acceptance in the systems research community.

The second principle is to *provide mechanism rather than prescribe policy*. This is directly reflected in the notion of application-aware adaptation itself. Applications are free to decide the fidelity at which data should be accessed given a particular set of environmental conditions. Odyssey provides the mechanisms to discover the state of the environment, and to affect fidelity changes. One important goal of building applications on top of Odyssey is to understand precisely which features are properly policy, and which can remain mechanism.

The third principle is to *respect the end-to-end argument*. Functionality or knowledge that must be at outer layers of the system for correctness should be present there alone. Odyssey duplicates these at inner layers only when there are either distinct performance advantages, or some fundamental capability is enabled by it. For example, an application must know the degree to which resources are available to it to correctly decide on a fidelity level. However, placing resource estimation in the system enables intelligent coordination between concurrent applications.

3.1.2 Historical Context

Many of the design decisions made in Odyssey are inherited from two previous systems: Coda and the Andrew File System, or AFS. Coda is a direct descendant of the second version of AFS, and borrows come code from it. Odyssey makes use of only a small subset of the code from these systems, but does inherit much of their design philosophy. This section discusses Coda and AFS, and how these two influenced the design of Odyssey.

Scalability, both in terms of performance and administration, is the central focus of AFS. Coda adds two goals in support of mobile computing. The first of these is to provide high availability in the face of network or server failures. The second is to provide application-transparent adaptation in the face of varying network quality.

AFS and Coda provide these capabilities within the context of the UNIX file system. They are transparent to applications, and behave much like a local file system. Both systems were designed for ease of development and experimentation, possibly at the expense of performance. For example, the bulk of these systems were implemented at user-level, at the cost of extra overhead in kernel boundary crossings and data copying.

To provide both scalability and security, these systems follow a strict client-server partitioning, with a small collection of dedicated server machines, and a larger group of untrusted clients. While the former are managed centrally and are physically secure, the latter are assumed to be owned by individuals and thus insecure.

For administrative simplicity, both AFS and Coda provide a single, global name space. This global name space is mounted at the same point in each client's local name space. Such uniformity provides name portability across machines and users. This single, global name space is broken into subtrees called *volumes* [72]. A volume is a collection of files that are viewed as a unit for administrative purposes. It is the unit for which quotas are enforced, is stored on a single server, and is backed up or relocated as a unit.

In order to build the complete name space, volumes are glued together at *mount points*, which themselves may occur within volumes. These are similar to the more familiar notion of UNIX file system mounts; however, the presence of a mount point and the name of the volume thus mounted are part of the global state rather than a side effect of client action. While the name space provided by Coda or AFS arises directly from the volume and mount point structure, it is seamless from the perspective of applications. Odyssey's design preserves the notion of a global name space, and inherits the volume substructure imposed on it. However, as discussed in Section 3.3.1 the name space is no longer entirely seamless; it is exposed through these tomes.

In both Coda and AFS, the global name space is mounted in the client's local name space as a new VFS file system [40]. Clients cache copies of files from servers, and then operate on them locally. Updated files are shipped back to servers as bandwidth permits. This process is managed by a local cache manager called *Venus* in both systems.

Neither Coda nor AFS place Venus in the kernel, but rather at user-level. This was done to simplify development and debugging, at a modest performance cost. To provide integration with VFS, these systems use a small *interceptor* which receives file operation requests from the VFS layer, and forwards them to Venus.

Odyssey borrows the notion of a user-level implementation as well as the code providing the in-kernel interceptor. However, Odyssey imposes some internal structure on the user-level cache manager as described in Sections 3.3.4 and 3.3.6.

3.2 Component Overview

Odyssey has incorporated a set of basic components to provide application-aware adaptation both effectively and efficiently. This section briefly outlines these basic components; they include abstractions, API extensions, and functional responsibilities. The discussion is centered around how an adaptive video player might make use of Odyssey; this serves as a simple example to illustrate Odyssey's key components and their interactions. Section 3.3 describes the design of each of these components in detail.

The video player adapts by trading fidelity for performance. Since fidelity is a type-specific notion, Odyssey must have some way of knowing the type of individual objects. It does so by associating a single type with all of the items in a volume.

In order to choose a particular fidelity level at which to play movies, the video player must know the current state of its environment. Odyssey must therefore provide applications with a way to name the *resources* that make up that environment; for example, the video player must be able to ask about the bandwidth available to the video server.

As the bandwidth to the video server changes in some significant way, the player will choose a different fidelity level. In application-aware adaptation, the video player does not directly monitor resource availability; that is the system's responsibility, as it is best positioned to do so. Furthermore the player need not know of every minute change in bandwidth; only sufficiently large changes will cause a change of fidelity. Odyssey provides an API extension called a *resource request*, which applications use to tell the system which resource changes would be significant.

The system-level component that monitors resource availability is called the *viceroy*. As the video player executes, the viceroy estimates the bandwidth available to the video server. If the bandwidth ever goes above or below the viceroy's requested bounds, the viceroy notifies the player of the new bandwidth level via an *upcall*.

When it receives the upcall, the video player will raise or lower the fidelity at which it is playing the movie. A change in fidelity may change the way in which data is represented. Thus, the application must be aware of fidelity changes. However, the end-to-end argument also dictates that the system components on the client should also have access to type information. The system provides the single point of resource control, and thus makes caching, transport, and consistency decisions. The type of an object may well influence those decisions. For example, it may be cheaper to recompute some item than to refetch it, changing the cost of flushing it from the client's cache.

Odyssey incorporates a set of components, called *wardens*, to provide type-specific functionality at the system level, one warden per type. Together with the viceroy and the interceptor, they form Odyssey's system support. There are potentially many types, each with potentially many fidelity-changing operations. Rather than attempt to enumerate each of these operations in the API, Odyssey instead provides a single, general mechanism called *type-specific operation*.

Together, these components form Odyssey's client architecture, shown in Figure 3.1. Odyssey focuses on the client for three reasons. First, applications are the entities responsible for forming adaptation policy; they make the decisions. Good software practices require co-locating the support for those decisions. Second, adaptation decisions are based on the availability of resources to a client application, from the application's point of view. The best place to provide support for such estimation is thus also at the client. Third, altering fidelity often requires very different data handling on the part of the client.[1]

3.3 Detailed Design

This section gives detailed descriptions of each of the components introduced in Section 3.2. Conceptual details, including API specifications, are spelled out in this section, but implementation details are left for Chapter 4.

[1]Of course, servers must also handle data differently as the fidelity of that data changes. However, this is not the focus of Odyssey *per se*, but rather can be determined solely between wardens and servers.

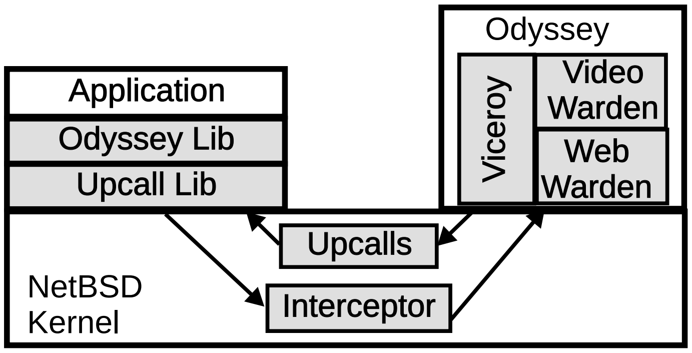


Figure 3.1: Odyssey Client Architecture

3.3.1 Tomes: Adding Types to Data

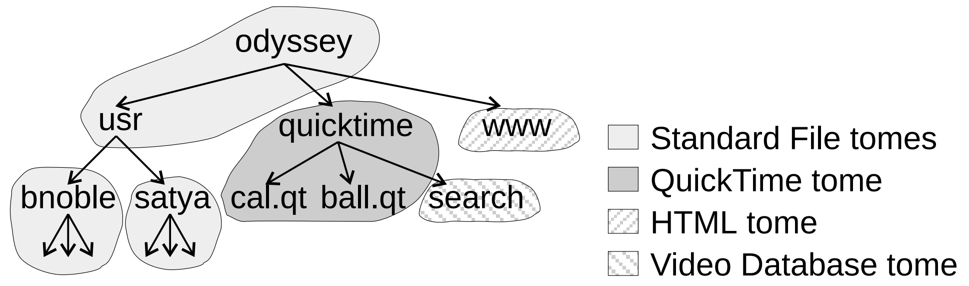
As with AFS and Coda, Odyssey provides a single, global name space to all clients. This name space is divided into sub-spaces which are the unit of administration in the repository. In order to simplify implementation and administration, these sub-spaces are also the unit of type. Borrowing from Odyssey's predecessors, they are called *tomes*, or typed volumes.

Each item within a given tome is of the same type, and a tome is stored in its entirety on a single server. Like the volumes of AFS and Coda, tomes are mounted within other tomes to form the global name space. This coupling of type to volume simplifies the administration of Odyssey. Good software practices suggest that objects of different type be served by different logical servers.[2] If a single volume were allowed to contain objects of multiple types, then that volume would be stored at multiple servers. Administrative tasks such as moving the volume would either require collusion between the servers storing the volume, or more likely could not be done atomically.

On the other hand, the binding of types to tomes exposes the types of objects in the name space. For example, a single directory cannot contain both video clips and map data; such exposure of type in the name space is obviously undesirable. However, adding a layer of indirection for naming, such as X.500 [12], or making use of aggregating objects, such as HTML documents, which name other objects transparently, would decouple naming from types, while preserving both the benefits of a convenient unit of administration as well as some logical directory structure.

An example name space of six tomes appears in Figure 3.2. The first tome, rooted at `odyssey`, is the *root volume*; a distinguished tome that is well-known to all clients. This tome is a standard file system tome. In this example, there are two other standard file system tomes, rooted at `bnoble` and `satya`, that hold the personal data for those two users. All of the other three tomes have distinct types. The tome rooted at `www` provides access to the World Wide Web. The tome rooted at `quicktime` contains quicktime-format video streams; the tome rooted at `search` is a database providing query and search capabilities over the movies in the `quicktime` tome.

[2]Of course, a single physical server may host more than one virtual server, each of a different type.



This figure depicts a simple example of an Odyssey name space. There are six tomes: three standard file system tomes, one video tome, one database tome indexing that video collection, and a tome providing access to the World Wide Web. From a client's perspective, all of this data is present in a single, global name space.

Figure 3.2: Example Tomes

Odyssey, unlike AFS and Coda, does not provide a strictly hierarchical name space. For example, the video database tome in Figure 3.2 provides a query-based naming scheme, while the World Wide Web tome uses URLs as names. Such extensions to the name space are similar in spirit to the Semantic File System [25]. The mechanism supporting these extensions is described in Section 3.3.6.

3.3.2 Resources: Naming the Environment

As they execute, applications adapt to their environment. To do so, they must have some way of naming salient features of the environment to which they adapt. The features of an Odyssey client's environment are the set of *resources* available to that client. For example, battery power is a resource, as is disk space available or the bandwidth along a particular network connection. As time progresses, the degree to which these resources are *available* will change; the battery will have drained, disk space may become more plentiful because some cache files were flushed, and the bandwidth may have gone up or down.

The environment of an application at any given point in time, then, is the degree to which each of the set of client resources is available to it. Each resource in the system is named by a unique identifier. The availability of each resource is represented in a 32 bit number, expressed in units appropriate to the resource. The units chosen should allow for a sufficiently large range and small granularity. Furthermore, they should not require an application to calibrate them to a particular machine in some way; if such calibration is necessary, the system should perform it. So, battery power is expressed in seconds of operation remaining, rather than joules; if the latter were used, an application would also need to know the rate at which the machine consumed power.

Resources are divided into two classes, *generic* and *type-specific*. Generic resources are meaningful to the client as a whole, while type-specific resources are meaningful only in the context of a specific kind of data. For example, battery power and bandwidth are both generic resources; they have meaning regardless of the type of data being accessed.

In contrast, the number of pre-paid queries on a particular database is a resource of interest only to applications using that database.

Resources are further classified by whether they are *universal*, or *item-specific*. Universal resources are defined without need for a frame of reference, while item-specific resources must be evaluated in the context of a particular Odyssey object. For example, battery life is a universal resource; there is no frame of reference needed to evaluate it. Bandwidth, on the other hand, is item-specific. Bandwidth is an end-to-end measure of a particular path from client to server. Paths to different servers may well have different bandwidths. Thus, bandwidth is estimated with respect to a particular Odyssey object; the result is the bandwidth available on the path to that object's server. By definition, all type-specific resources are also be item-specific.

Universal		Item-Specific	
Resource	Units	Resource	Units
Disk Cache Space	kilobytes	Network Bandwidth	bytes/second
CPU Cycles Available	SPECint95	Network Latency	microseconds
Battery Power	seconds		
Money	cents		

This figure lists the generic resources in Odyssey's design, and the units in which their availability is measured. The first column lists universal resources. The second lists the resources that are item-specific.

Figure 3.3: Generic resources in Odyssey

The generic resources are shown in Figure 3.3. There are four universal, generic resources: disk cache space, CPU cycles, battery power, and money. There are two item-specific, generic resources: network bandwidth and network latency.

Disk cache space is the most straightforward of the universal resources. Facilities for measuring available disk space already exist. Furthermore, applications can make obvious use of such measurements; when space is plentiful, applications can ask for more aggressive caching and prefetching policies than they might otherwise.

Available computational power is also of obvious importance, particularly in the face of adaptation. Many adaptive strategies involve compression, approximation, or regeneration of data. However, there may not be enough CPU available to the client to perform such functions. However, a simple measure such as fraction of idle time is insufficient; it would require the application to calibrate to the total processing power of the machine. Instead, Odyssey uses SPECint95 units to express CPU availability. This measure is not perfect, but it is a step in the right direction.

Mobile machines are rarely connected to a permanent power source, and are therefore highly dependent upon batteries. Knowing the state of those batteries is important for several reasons. There are several uses for battery information beyond saving critical information when a power failure is imminent. For example, clients can take advantage of variable-power radios. Using more power when transmitting can reduce the bit-error rate on the channel, but should be done only when power is plentiful. Power is expressed in units of seconds of operation, rather than joules, for much the same reason that available CPU is not measured in idle time. A unit such as joules requires the application have machine-specific knowledge; some machines may require more energy to operate than others. In contrast, seconds of operation is a machine-independent unit.

Money is of increasing importance to mobile clients. For example, cellular networks have steep costs that are often difficult to compute. Other remote services, such as printers and file servers, may charge foreign clients in exchange for their use. As infrastructures for electronic commerce improves, applications will increasingly have direct control over money. An application might make different decisions about what services to use given the cost of those services in light of user preference and budget.

The two item-specific, generic resources — bandwidth and latency — are measures of network quality. These quality metrics are end-to-end measures of an entire network path from the client to server, and can vary significantly from server to server. These two resources, then, must be item-specific; they must be measured with respect to the server storing a particular object.

Adapting to bandwidth is the focus of the current Odyssey prototype. When bandwidth drops, time-critical data must somehow be degraded to maintain performance goals; when bandwidth rises, quality can be increased. Latency information can be used to adjust buffering requirements on the client. It may also be used to determine how aggressive an application might be in adapting, since it is a measure of the minimum server response time.

Odyssey's design currently encompasses only a single value for bandwidth and latency between a client and a particular server, rather than different measures for inbound and outbound quality. Thus, Odyssey implicitly assumes that network performance is symmetric. The problem of asymmetry in networks, both mobile [7, 54] and wired [41], is a topic of recent interest. Despite this, the assumption of symmetry has proven adequate thus far.

3.3.3 Resource Requests: Expressing Expectations

Applications do not need continuous knowledge of all resources. It is likely that any given application will be concerned only with a small subset of all resources. Furthermore, most small changes in this set of resources will not be sufficient to cause a change in fidelity.

For example, consider a video player which changes the fidelity of its video stream as bandwidth rises and falls. For any given bandwidth level, the player selects a particular fidelity level to meet its performance constraints. There is an upper bound above which the player will raise fidelity. Likewise, there is some lower bound below which the player must lower fidelity. Within this range, called a *window of tolerance*, the player will not change its strategy; the exact value of bandwidth within this window is unimportant.

At any instant such windows exist for each resource of interest to an adaptive application. This collective set of windows are the *expectations* that the application has of its environment. So long as these expectations are met, the application will not change the fidelity at which it is accessing data.

If one of the resources in the set of expectations leaves its window of tolerance, the application must be told of its new value. The application is then free to choose a new fidelity level, and will pick a new window of tolerance on the changed resource. If the application should decide it no longer wishes to be notified for a particular resource, it can *cancel* a previously placed request.

Applications make resource expectations known to the client through *resource requests*. The system remembers these expectations, and promises to notify the application if any of them are violated. There are two forms of request; their function prototypes, along with associated types, appear in Figure 3.4.

Both forms of request follow the same general pattern. The application passes a window of tolerance for a particular resource. The viceroy checks the availability of that resource; if it is within the declared bounds, the viceroy returns a *request identifier* in the `result` parameter. This identifier represents a promise by the viceroy to inform the application should the resource stray beyond the stated bounds.

If, on the other hand, the viceroy determines that the resource is outside the tolerance bounds, the request returns `ODY_ENOTINWINDOW`, and places the current availability of the resource in `result`. The application is expected to try again with a window encompassing the current availability.

In addition to resource bounds, requests specify an Odyssey object, either by by pathname or file descriptor. The object is used as the context in which to evaluate the availability of item-specific and type-specific resources. The pathname form is necessary because a process might not wish to go to the expense of opening an object in order to place requests on associated resources. Likewise, the file descriptor form is required for processes which inherit open file descriptors from parents without having the pathnames that correspond to them.

To safeguard against out-of-date clients, requests carry version information both for generic and type-specific interfaces. This is provided not only for the usual reasons of defensive programming, but also because the type-specific portions of Odyssey are expected to change over time.

In the event that an application no longer wishes to be notified for a previously granted request, it may `cancel` it. An application may also replace an old window of tolerance with a new one simply by placing a new request. The system allows only one registered window per resource on behalf of a single application, since multiple, overlapping windows can always be combined into the single tightest set of bounds on the current resource value. All requests are implicitly cancelled on process exit, to avoid the cost of sending a notification to a process that doesn't exist.

Unlike signal handlers, resource windows are not inherited across calls to `fork`. The resources available to the original application will be spread across both the parent and child of the fork, commonly invalidating the parent's window immediately. Therefore, the child is expected to place a new lower window. As with signal handlers, all registered

```
struct ody_vers {
    unsigned int32_t  ov_gs;
    unsigned int32_t  ov_type;
    unsigned int32_t  ov_tsvnum;
};

void (*ody_req_fn_t) (IN int32_t reqid,
                      IN u_int32_t rsrc,
                      IN int32_t val);

struct ody_req_des {
    u_int32_t        ord_resource;
    struct ody_vers  ord_version;
    int32_t          ord_low;
    int32_t          ord_high;
    ody_req_fn_t     ord_fp;
};

int ody_request (IN  char *path,
                 IN  struct ody_req_des *request,
                 OUT int32_t *result);

int ody_frequest (IN  int fd,
                  IN  struct ody_req_des *request,
                  OUT int32_t *result);

int ody_cancel  (IN int32_t reqid);
```

This figure specifies the API used by Odyssey applications to place and cancel resource requests. Resource requests are placed through `ody_request` and `ody_frequest`; the latter uses a file descriptor rather than a pathname to specify the Odyssey object in whose context the resource should be evaluated. Requests are cancelled by the `ody_cancel` call. The remainder of the figure specifies the types used in these three calls.

Figure 3.4: Resource Request Interface

windows are cleared on a call to `exec`; the body of code originally meant to handle the notification will not be part of the process after the call.

The decision to name only one window per request is an explicit one; doing so preserves the goal of simplicity. One could implement requests passing a vector of windows at once, defining an N-dimensional space, with one resource per dimension. A request of two resources defines a rectangle within the plane, and so on. Such a form would complicate the handling of the request call, and unnecessarily commingle resources that, in the current design, are completely separate. In return for this complexity, applications could amortize the cost of a single system call across many requests.

However, such a scheme does not fundamentally improve the functionality of the API. If one desired to, one could implement such a scheme by blocking the reception of upcalls, placing each of the individual requests in the vector in turn, and resuming the reception of upcalls. This duplicates the vector-based behavior, at the cost of additional system calls but at the savings of a simpler approach.

3.3.4 The Viceroy: Controlling Resources

To effectively manage resources in the presence of concurrent applications, the system must provide a single point of control for each resource. This single point for the generic resources is called the *viceroy*. The viceroy is responsible for monitoring availability of CPU, disk space, battery power, and money as well as the bandwidth and latency on connections to all servers.

When an application places a request on one of these generic resources, the viceroy checks the version stamps and resource window for validity, and then compares the resource's current availability with the window bounds. If the resource is not in bounds, an error code and the current value are returned. If the window is in bounds, it is remembered by the viceroy for comparison with future estimates of that resource's availability, and the request identifier is returned.

The viceroy is also the point at which any resource reservation or admission control decisions must occur. The current design of Odyssey provides only for resource estimation, not reservation. However, stronger forms of resource management are not ruled out, and could be incorporated into Odyssey.

In addition to the tasks of resource monitoring and control, the viceroy is responsible for all type-independent tasks on behalf of the Odyssey client. This includes object management, operation dispatch, and the provision of a uniform communications substrate. Each of these are described in the remainder of this section.

Object Management

Each object in the Odyssey store is named by a unique identifier called an `ofid`. The `ofid` is a four-part number, identifying the type, tome, *vnode number,* and *uniqifier*; it is illustrated in Figure 3.5. Each type is represented by a well-known number that is glob-

```
struct ofid {
    u_int32_t  ofid_ttype;
    u_int32_t  ofid_tome;
    u_int32_t  ofid_vnum;
    u_int32_t  ofid_uniq;
};
```

Figure 3.5: Structure of an Odyssey Identifier: `ofid`

ally assigned. Within each type, tome numbers must name a single, distinct tome. Tome numbers may never be recycled within a type, though they may be duplicated across types.

The third number, *vnode*, identifies a particular object within a tome. At any instant, the vnode numbers within a single tome should be unique, but they are allowed to be re-used by a single tome. To prevent any confusion arising from such re-use, each new object in any given tome must be assigned a strictly increasing uniqifier. This prevents clients from confusing old vnode numbers with new objects.

This two-part object identifier is not strictly necessary; the vnode number is superfluous. The use of two numbers for object identification within a tome was inherited from AFS and Coda. These systems used vnode numbers for efficient object access by re-using slots in persistent data structures rather than resort to dynamic management of a persistent heap [68]. These systems also encoded semantic information in vnode numbers, using the low-order bit to distinguish between directories and leaf nodes.

Operation Dispatch

All operation requests forwarded from the interceptor are fielded by the viceroy. Each request names the operation to perform, along with an `ofid` that names the object on which the operation should be performed.

As described in Section 3.3.6, every operation on an object of a given type is handled by a type-specific component within Odyssey. By examining the `ofid` on which the operation is to be performed, the Viceroy forwards the operation request to the correct code component. The general design of these type-specific components is described in Section 3.3.6, as are the interactions between these components and the viceroy.

Communications

In order to act as the point of network control, the viceroy must provide a uniform communications substrate to the rest of the system. This substrate is connection-based, and provides estimation of per-connection bandwidth and latency. It also handles connection creation, destruction, and automatic re-establishment of broken connections; it is called the *connection manager*.

Connections are named by `<host, service, number>`, allowing multiple connections to the same service if necessary. They are based on RPC2[65], which provides both traditional remote procedure call as well as a sliding-window, selective-acknowledgement protocol for bulk transfer. All communication on behalf of Odyssey is expected to make use of the connection manager; without doing so, there is no single point of network control.

The current focus of Odyssey is adaptation with respect to changes in network bandwidth; the connection manager's chief responsibility then is estimating available bandwidth, tracking variation in both the supply of and demand for network bandwidth. Because Odyssey may often be used in weakly-connected environments, we rely on purely passive observation; Odyssey adds no traffic whatsoever to that already present. This is in contrast to active approaches, such as Keshav's packet-pair protocol [38]. The details of estimation are presented in Chapter 4.

This estimation is built upon that performed in Coda, designed in collaboration with Mummert [49]. As a side effect, RPC2 records the sizes and times of requests and responses between client and server. These observations are recorded in two logs per connection, one for remote procedure call and one for bulk transfer. In Coda, each connection's logs were examined separately to estimate the bandwidth to each file server. Odyssey extends this by first collecting all logs into a master log, to accurately account for interference between competing connections.

3.3.5 Upcalls: Notifying Applications

When the viceroy detects that some resource has strayed beyond a requested window of tolerance, it must notify the requesting application. This notification, as shown in Figure 3.4, carries with it the request for which the notification is being sent, the resource whose availability has changed, and the new availability of that resource.

There are three design requirements of such a notification mechanism. First, such notification can happen at any time during an application's execution; thus, notifications should be asynchronous, requiring no application action. Second, notifications should be delivered reliably and in a timely fashion. Third, applications should not have to subscribe to a particular programming model beyond the common UNIX API in order to receive them.

This last requirement leads one directly to the `signal` facility; signals are asynchronous, delivered in reasonable time, and are familiar abstractions. Unfortunately, they carry two shortcomings rendering them unsuitable for resource notification. First, they are unreliably delivered; most UNIX implementations promise at-most-once delivery rather than exactly-once. Second, they do not provide any convenient way to pass arguments with them; such arguments are required for resource notification.

IPC mechanisms, such as UNIX-domain sockets or Mach's messages [18], provide exactly-once delivery and the passing of arguments. Unfortunately, they also require the application to either frequently `select` on some set of file descriptors, or to make use of a particular thread model. In keeping with the philosophy of minimalism, Odyssey avoids IPC mechanisms and their associated encumbrances.

Instead, Odyssey adds to the kernel an *upcall* facility. This facility provides for asynchronous notification in the manner of signals, but adds exactly-once semantics and the passing of arguments. This mechanism is meant to stand apart from Odyssey, and is used for tasks other than resource notification; for example, it plays an important role in data collection. The API for handling and invoking upcalls appears in Figure 3.6.

To simplify administration, the total set of upcalls supported by the system is partitioned into *upclasses*. Each upclass contains a set of related upcalls. For example, all upcalls relating to resource notifications belong to the same upclass. Thus, an upcall is named by an `(upclass, upcall)` pair. This partitioning allows different subsystems to autonomously manage their own upcalls.

An upclass provides one or more upcalls, each named by a unique identifier. Associated with each upcall is an argument type, and possibly a return value type. Arguments and results are passed in unstructured memory buffers. Callers and callees must agree on the interpretation of such buffers, as the system places no interpretation on them.

Of course, if the operating system itself supported threads, upcalls could be done away with. Since the application would already be party to the kernel's threading model, the Odyssey run-time library could spawn a thread meant only to handle resource notifications. However, it has been argued by Ousterhout [55] that simple event-handling mechanisms, such as upcalls, are preferable to threads when the full power of the latter is not necessary.

Handling Upcalls

To receive a particular upcall, a process must first declare its intention to handle it. This is done through the `upcall_reg` system call. The call specifies an upcall that the process will handle and a function to handle it. A system component responsible for the upclass in question will be notified of any registrations to allow for system bookkeeping.

Upcall registration is similar to the placement of signal handlers. A successful call to `upcall_reg` returns a pointer to the previous handler, or NULL if no such handler exists. Registering a NULL handler for a particular upcall declares the process's unwillingness to continue handling that upcall. As with signal handlers, upcall registrations are inherited across calls to `fork`. After an `exec`, all upcall registrations held are cancelled; the body of code registered to handle the upcall will no longer be part of the process after the call. Registration changes resulting from `fork` and `exec` are made known to upclasses as a side effect.

An upcall handler takes seven arguments. The first two specify the upcall for which the handler is being invoked, allowing a single function to handle multiple upcalls. The second two, `argsz` and `argbuf`, pass the arguments of the upcall to the process. The buffer containing the arguments is pointed to by the latter; the former gives the size of that buffer.

The next two arguments, `retszp` and `retbuf`, are used to pass return values back to a waiting caller. The latter is the actual buffer and is created before execution of the handler. The former is a pointer to an integer which, on entry, gives the maximum possible size of the return buffer. The handler, after filling the return buffer, is expected to reset it

```
typedef int32_t         upclass_t;
typedef int32_t         upcall_t;

/* Upcall-handling calls */

int (*uc_hdlr_t) (IN     upclass_t upclass,
                  IN     upcall_t  upcall,
                  IN     size_t    argsz,
                  IN     void      *argbuf,
                  IN/OUT size_t    *retszp,
                  OUT    void      *retbuf,
                  IN     int       flags));

void *upcall_reg (IN upclass_t upclass,
                  IN upcall_t  upcall,
                  IN uc_hdlr_t handler);


int upcall_block();
int upcall_unblock();

/* Upcall-invocation calls */

int upcall_sync  (IN     upclass_t upclass,
                  IN     upcall_t  upcall,
                  IN     pid_t     pid,
                  IN     size_t    argsz,
                  IN     void      *argbuf,
                  IN/OUT size_t    *retszp,
                  OUT    void      *retbuf);

int upcall_async (IN     upclass_t upclass,
                  IN     upcall_t  upcall,
                  IN     pid_t     pid,
                  IN     size_t    argsz,
                  IN     void      *argbuf)
```

This figure shows the interface for handling and posting upcalls, along with associated data types. Upcalls are partitioned into *upclasses*, each of which are managed independently. A process declares its intentions to handle a certain upcall with `upcall_reg`; when posted, that upcall will be delivered to the function named in the `handler` argument. Upcalls may be blocked with `upcall_block`, and unblocked with `upcall_unblock`
Upcalls may be posted by either `upcall_sync` or `upcall_async`. The former blocks until the upcall is handled, and may then accept a return value from the callee; it may post to only a single process. The latter continues without blocking, and cannot accept return values; however, it can post upcalls simultaneously to process groups, or all interested processes, rather than a single process.

Figure 3.6: Upcall Handling/Invocation Interface

to reflect the actual size of the result buffer. In addition to the result buffer, the handler is expected to return a result code much like a UNIX system call: zero for success, some other meaningful error code on failure.

The final argument, `flags`, is used to tell the handler whether it was invoked synchronously or asynchronously. In the former case, the callee is waiting for the result from the handler; in the latter it is not.

A process may block the reception of all upcalls by calling `upcall_block`, and resume reception of them via `upcall_unblock`. Each call to the former must be balanced by one to the latter; if `upcall_block` is called twice in succession without an intervening `upcall_unblock`, `upcall_unblock` must be called twice to resume reception.

Posting Upcalls

Upcalls may be posted either synchronously or asynchronously. Synchronous upcalls may be posted only to a single process, and may receive results from the callee. Asynchronous upcalls can be posted to a single process, all interested processes in a process group, or all interested processes on the host. They do not receive results from the callee; rather, the posting process continues immediately.

Synchronous upcalls are posted by the `upcall_sync` system call. It takes seven arguments. The first two name the upcall being posted. The third, `pid`, is a process identifier that names the process for which this upcall is intended; it must be a currently running process that has declared an interest in the posted upcall. The next two arguments, `argsz` and `argbuf` pass the size of the argument buffer and the buffer itself, respectively. The final two arguments, `retszp` and `retbuf`, are used to obtain return values from the callee. The caller is responsible for allocating the return buffer, and placing the size of that buffer in `retszp`. The callee will reset `retszp` to be the actual size of the argument structure, which will not be larger than the original size.

Asynchronous upcalls are posted similarly. However, since asynchronous callers do not wait for results, they need not provide result buffers. Furthermore, asynchronous upcalls may be sent to groups of processes rather than a single process; if `pid` is negative, it is sent to all processes in the process group whose identifier is equal to `-pid` and have declared a willingness to handle the upcall. If `pid` is zero, the upcall is posted to all processes on the host which have declared interest.

Posted upcalls are delivered to each process for which they are intended exactly-once and in-order. They are also delivered strictly serially; if a process is currently handling an upcall, any pending upcalls will not be delivered until the in-progress upcall is handled.

3.3.6 Wardens: Handling Typed Data

When applications are notified of resource changes, they must change the fidelity at which they access data. Since fidelity is a type-specific notion, changing fidelity must also be type-specific. To provide such fidelity-changing operations, Odyssey incorporates a collection of type-specific managers called *wardens*, one per type. Wardens are responsible for all operations on data items of their type, and communication between the client and servers storing those items.

Fidelity changes could, in principle, be handled entirely by applications directly rather than a system-level component. It would be possible to put no type-specific handling in Odyssey, and instead treat all data as an untyped byte stream. However, there are two compelling reasons to place some degree of type-awareness in Odyssey.

Just as there must be a single point of control for concurrent access to any given resource, there must also be a single point of control for concurrent access to any given object. For example, suppose two different applications request two different fidelities of the same object. A system component privy to both requests has an opportunity to merge them, improving service for one or both requesting applications.

More generally, the wide disparity in the physical and logical properties of various data types requires that some form of type-awareness be incorporated into the system for efficient resource usage. For example, the size distribution and consistency requirements of data from an NFS server differ substantially from those of relational database records. Image data may be highly compressible using one algorithm but not another. Video data can be efficiently shipped using a streaming protocol that drops rather than retransmits lost data; in contrast, only reliable transmissions are acceptable for file or database updates. It is impossible to optimize for such differences without some system-level knowledge of type.

As described in Section 3.3.4, the viceroy forwards operations on an object of a particular type to that type's warden; this dispatch is carried out by inspecting the type field of the operand's `ofid`, and forwarding the operation to the warden.

Wardens advertise their operations to the viceroy by means of a *warden table*, which lists each function supported by the warden. The functions in this table, summarized in Figure 3.7 are divided into three groups: administration, general operations, and Odyssey extensions. Administrative and general operations are outlined in the remainder of this section. The main Odyssey extension, the *type-specific operation*, is outlined in Section 3.3.7.

Administration functions are largely used during initialization or for debugging. Odyssey may be started with any number of arguments, some for the Viceroy, others for the wardens. Each warden is expected to provide both a function to parse arguments as well as a usage message describing the arguments it accepts. Each warden must also provide an initialization routine to be called by the Viceroy at startup time; the warden is guaranteed that no operation other than argument parsing will be invoked before initialization. Wardens may also provide a statistics collection function, which can report usage statistics to a named file descriptor.

The most important administration function, `getcid`, is used by the Viceroy to obtain the connection associated with a particular Odyssey object. Since the wardens are responsible for interactions with individual servers, they are the components that map between individual objects and the servers that store them. However, the Viceroy is responsible for fielding resource requests, including those for as network bandwidth. This function allows the viceroy to query the warden for the appropriate connection on which to actually place the request.

Administrative Functions	
`parseargs`	Given a partial list of arguments in `argv`, try to parse the next and subsequent arguments. Return the number of arguments consumed. Only function that may be called before `init`
`init`	Set up the warden's internal data structures, if any.
`pstats`	Print summary statistics to a particular file.
`getcid`	Given one of this warden's onodes, return the connection to the server storing that onode.
General Functions	
`lookup`	Given an onode and a pathname component, return the named onode.
`root`	Obtain the root object of a tome.
`access`	Given an onode, a credential, and an operation, return successfully if the user named by the credential is allowed to perform the operation on the onode.
`getattr`	Return the meta-data (attributes) of an onode).
`open`	Open an onode.
`close`	Close an onode.
`rdwr`	Read a buffer from or write a buffer to an onode. The warden is responsible for allocating the buffer which read operations are to fill, allowing it to satisfy reads from prefetched buffers.
`rdwrdisp`	Dispose a buffer previously returned by `rdwr`
`readdir`	Read a portion of a directory's contents.
`readdirdisp`	Dispose a buffer previously returned by `readdir`.
Odyssey Extensions	
`tsop`	Warden's function to handle type-specific operations. Responsible for allocating result buffer, if any.
`tsopdisp`	Warden's function to dispose result buffers returned by `tsop`.

Figure 3.7: Warden Table Summary

The general operations are those required of any VFS client. In order to support legacy applications, which expect the system to treat all file system objects as untyped byte streams, wardens are expected to provide at least `open`, `close`, and `rdwr`. Wardens must also support access control via the `access` operation, and provide meta-data through the `getattr` operation.

Wardens also provide name resolution via the `lookup` operation. When the viceroy receives a name resolution request, it is given a directory `ofid` from which to start resolution, and a pathname. That `ofid` and pathname are passed to the appropriate warden, which is expected to resolve the pathname as far as possible. If the warden encounters a mount point during resolution of the pathname, it passes control back to the viceroy, which then dispatches the remainder of the resolution to the warden of the mounted tome's type. Name resolution may be specialized to provide something other than the hierarchical UNIX file system. For example, a Web warden might provide naming based on URL's embedded in pathnames.

3.3.7 Type-Specific Operations: Changing Fidelity

There may be many different types supported by Odyssey, and each type may support many different fidelity-changing operations. Enumerating each such operation may well prove impossible; types and their supported fidelities will certainly evolve over time. Odyssey therefore provides a single, general-purpose mechanism, called *type-specific operation*, or *tsop*. It is used to provide both fidelity-changing operations as well as other useful type-specific extensions to the interface.

There are two forms of `tsop`, both following the same general form, taking seven arguments. The first argument is the object upon which the operation is to be performed. The second specifies the version of the Odyssey interface being used by the caller. This is used to detect version skew between applications and wardens, much as with the `request` operation described in Section 3.3.3.

The third argument specifies the operation to perform on the named object; the operation is named by an integer that is unique within a type, but may be reused across types. The fourth and fifth arguments together specify the arguments, if any, to be used in carrying out the named operation; the arguments themselves are passed in the unstructured buffer `arg`, and the size of that buffer is passed in `argsz`. The results from the operation are passed back in the sixth and seventh arguments. The results themselves are passed back in a buffer, `ret`, that is supplied by the caller. The original size of that buffer is passed in `retsz`, and the actual size of the result is placed there on return.

The two forms differ in how they name the object upon which the operation is to be performed. The first, `ody_tsop`, passes a file descriptor referring to some previously opened Odyssey object. The second, `ody_ftsop` passes a pathname to identify the object. As discussed in Section 3.3.3, both forms are necessary.

```
int ody_tsop (IN     char *path,
              IN     struct ody_vers *vers,
              IN     unsigned int op,
              IN     size_t argsz,
              IN     void *arg,
              IN/OUT size_t *retsz,
              OUT    void *ret);

int ody_tsop (IN     int fd,
              IN     struct ody_vers *vers,
              IN     unsigned int op,
              IN     size_t argsz,
              IN     void *arg,
              IN/OUT size_t *retsz,
              OUT    void *ret);
```

This figure specifies the API used by Odyssey applications to perform type-specific operations. There are two forms, `ody_tsop` and `ody_ftsop`. the former uses pathnames to specify the object upon which the operation is to be performed; the latter uses file descriptors. These functions are similar to the UNIX `ioctl` system call.

Figure 3.8: Type-Specific Operations

3.4 Summary

Odyssey is designed as a small set of extensions to the NetBSD operating system. These extensions provide facilities for application-aware adaptation on a mobile client. They were designed to be compatible with the UNIX idiom, and provide an efficient split of mechanism and policy, while respecting the end-to-end argument.

The design borrows heavily from two previous systems, Coda and AFS. Odyssey maintains a strict split between trusted servers and untrusted clients. It provides application-aware adaptation in the context of the file system, while maintaining ease of administration and development.

Odyssey introduces the idea of resources as the environment in which adaptive applications operate. The system monitors the availability of those resources, and applications use resource requests to tell the system which resource changes are of interest. An Odyssey client is divided into the viceroy, which manages resources on the client and is responsible for type-independent functionality, and a set of wardens that provide type-dependent operations, one warden per type. The viceroy and wardens together are implemented in user space.

The viceroy remembers resource requests, and compares them to the changing availability of resources. If a resource strays outside of a requested window of tolerance, the viceroy notifies the requesting application with an upcall. Upcalls are asynchronous, carry arguments and return results, and are delivered exactly-once and in order.

Application operations on Odyssey objects are forwarded to the viceroy by an in-kernel interceptor, and are then dispatched to the appropriate warden. Wardens provide both standard file system operations as well as type-specific operations when possible. Wardens provide fidelity-changing operations, and also extend the standard UNIX name resolution operation in type-specific ways.

Wardens are responsible for interactions between the client and servers, but use a common communications substrate to do so. All Odyssey traffic is expected to use this common substrate, providing a single point of control for the network. This single point of control allows Odyssey to manage the network intelligently in the face of concurrent applications.

Chapter 4

Implementation

This chapter presents the implementation of the Odyssey prototype. It begins by asking the questions that the implementation was to answer; these goals dictated the areas in which implementation effort was focused. The chapter then summarizes the two NetBSD features required for the discussion of the implementation — *signal delivery* and the *virtual file system*, or VFS.

The bulk of the chapter presents the Odyssey prototype itself. It begins with a description of the two mechanisms on which the remainder of the prototype is built: *upcalls*, which provide support for resource requests and notifications, and the *interceptor*, which glues the Odyssey name space into the client's file system. The interceptor forwards Odyssey requests to the *viceroy*, which dispatches them to the appropriate *warden*. The chapter describes the details of the viceroy's implementation, and the general structure of the wardens.

Finally, the chapter presents the API extensions provided to Odyssey applications. *Resource requests*, and the accompanying *notifications*, are implemented in terms of both upcalls and VFS extensions. Type-specific operations make use only of the latter.

4.1 Implementation Goals

The Odyssey prototype was built to answer a specific set of questions:

- Can the Odyssey API and architecture effectively support application-aware adaptation?
- Can the viceroy estimate resources with enough accuracy to enable applications to make good adaptive decisions? Can this be done even for resources that are not directly under the client's control, such as network bandwidth?
- How critical is a single point of resource control required to support concurrent applications?

The first, and most important, implementation goal was to confirm that the design could support application-aware adaptation. This focused implementation effort on the API extensions and the architecture supporting the viceroy and wardens.

The second goal was to understand the impact of estimating the availability of resources entirely at the client. This estimation is relatively simple for resources under direct client control, such as disk space. Tools for measuring resources of more recent interest, such as battery power, are being integrated with current hardware [32]. However, for resources not entirely the province of the client, such as network bandwidth, the case for purely local estimation is less clear. This observation, combined with the obvious need to adapt to changes in the bandwidth available to a mobile host, led to the focus on bandwidth as the first resource to which to adapt.

Finally, one can argue from first principles why applications must have some say in the adaptation process. However, the need for the system's involvement — specifically as the single point of resource control — is less clear. The prototype therefore ensures that all network traffic is visible to the viceroy, and the viceroy's support for communications is one of the main thrusts of its implementation.

The prototype also was influenced by what were explicitly not the goals of its construction. For example, while performance of the prototype is a concern, some performance was sacrificed in building a user-level viceroy and set of wardens to ease implementation and debugging. Furthermore, the prototype focuses on support for adaptive applications exclusively rather than include legacy applications, making a complete implementation of the VFS interface less important.

4.2 Implementation Background

There are two mechanisms in the NetBSD kernel upon which the prototype relies: *signal delivery* and the *virtual file system*, or VFS. This section provides a brief summary of these two, intended to give enough context the remainder of the chapter. For more details on these mechanisms, the interested reader is directed toward McKusick's text [46].

4.2.1 Signal Delivery

The UNIX signal mechanism provides a fashion of *software interrupt*. Signals can be used to notify a process, asynchronously, of some exceptional condition. By default, most signals terminate the process to which they are sent, though some pause the process and others are simply ignored. A process may choose instead to handle a signal with a particular function, or to ignore it entirely. Signal handling is expensive in comparison to local procedure calls, and are comparable in cost to most inter-process communication mechanisms.

At the heart of NetBSD's signal implementation is the `sigset_t`, which is a single 32-bit integer, one bit per signal. This data structure represents signal lists in many different instances. For example, the element of the process structure that lists all signals pending delivery to a process, `p_siglist`, is a `sigset_t`.

A process posts a signal to another process by setting the appropriate bit in the signalled process's `p_siglist`. Delivery of this signal can happen only at discrete times. Signals may be delivered when a process is in or enters an interruptible sleep. They also may be delivered upon returning from a trap or system call.

Since posted signals are represented by a bit, and signals are not delivered the instant they are posted, signals are delivered with at-most-once semantics. If the same signal is posted to a process twice before the first instance is delivered, the process will only see a single delivery of that signal.

Signal delivery itself is a very complicated, three-step process. In the first step, the kernel builds a *signal context*, and adds arguments for the signal handler to the signalled process's stack frame. As part of this modification of the stack frame, the kernel sets the current instruction pointer of the signalled process to point to the *signal trampoline* code, which is placed at the top of each process's stack at process creation time. The trampoline, which is the second step of signal delivery mechanism, will then run when next the process is pulled from the run queue. The trampoline then calls the actual signal handler as the third step.

4.2.2 VFS: Virtual File System

The BSD 4.4 file system, along with the file system of several other kernels, uses the Virtual File System, commonly known as VFS [40], to glue together several distinct file systems under a unified layer. The VFS layer presents a unified front to the file I/O system calls, as well as a framework within which each of the individual file system types can be implemented, as shown in Figure 4.1.

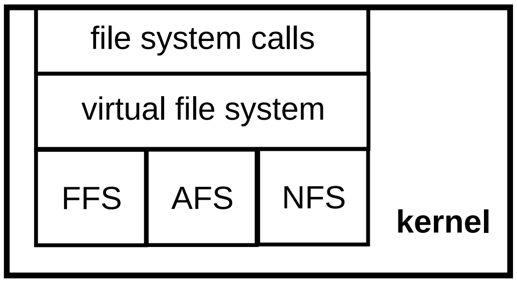


Figure 4.1: VFS: the Virtual File System

The central abstraction in VFS is the *vnode*, which represents a single object in one of the file systems under VFS. Each vnode has a type-independent layer common to all vnodes, and a type-specific layer which is managed entirely by the underlying file system. Included in the type-independent state of the vnode is the file system type to which it belongs, and whether the vnode is a *directory* or *regular file*; the former may be used as contexts for name resolution, but the latter may not.

The file I/O system calls call down to the VFS layer, which then redirects these calls through a function table called the *vnode operations table*. Each vnode has a pointer to the operations table for its file system type; different file systems implement the same operations differently. There is one entry per file system operation in the vnode table, and the operations are commonly implemented by all underlying file systems. This structure is illustrated in Figure 4.2.

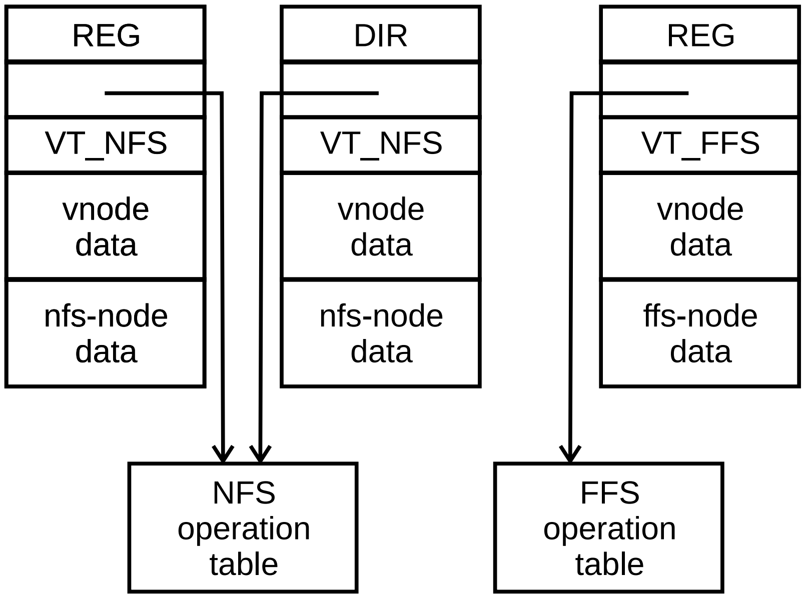


Figure 4.2: Structure of Vnodes

The original VFS implementation required each underlying file system type to implement each of the vnode operations. BSD 4.4 included an extension to VFS called *stackable layers* [27]. This extension allowed each file system to implement a different subset of all vnode operations; a file system's particular subset was determined at boot time from a descriptor provided by the file system. This allows individual file systems to add functionality to VFS without requiring changes to all other file systems in the kernel.

4.3 Component Overview

There are four main bodies of code that together provide support for application-aware adaptation: upcalls, the in-kernel interceptor, the viceroy, and wardens. They were introduced in Section 3.2, and are depicted in Figure 3.1.

The upcall mechanism is one of the basic building blocks with which Odyssey is constructed; upcalls are chiefly used for resource notifications. The bulk of this mechanism is within the kernel; it consists of approximately two thousand lines of code. Most of this in-kernel code is new, though small changes were made to the signal handling mechanism and the routines that handle `fork`, `exec`, and `exit`. There is also a small user-level library that applications using upcalls must include; it is approximately 500 lines of code.

The in-kernel interceptor forwards Odyssey requests to the viceroy, and also implements the extensions to the system call interface. It was ported from the Coda *Mini-*

Cache [76], which was twelve thousand lines of code. In this port, approximately four thousand lines were removed; other than changing the names of routines and structures, approximately one thousand lines of code were added or changed. There is also a very small user-level library supporting the API extensions; it is approximately 1.5 thousand lines of code.

The viceroy forms the bulk of the implementation effort. It makes use of pre-existing code to implement the user-level threading and communications packages, consisting of 36 thousand lines of code. The viceroy adds to this less than twelve thousand lines of code for object management, communications infrastructure, operation forwarding, and resource management.

The individual wardens — the type-dependent code components — are described in Chapter 5; this chapter focuses only on their common structure. However, a simple warden providing access to data streams is less than 1.5 thousand lines in size. This warden is used in support of a synthetic application used in experiments, described in Section 7.2.2, and represents a minimal warden.

4.4 Basic Components

There are two basic components on which the remainder of the Odyssey prototype is built: *upcalls*, which provide exactly-once, asynchronous notifications with arguments and return values; and the in-kernel *interceptor*, which forwards vnode operations on Odyssey objects to the viceroy. This section describes the implementation of each of these two, and lays the foundation for the description of the prototype itself.

4.4.1 Upcalls

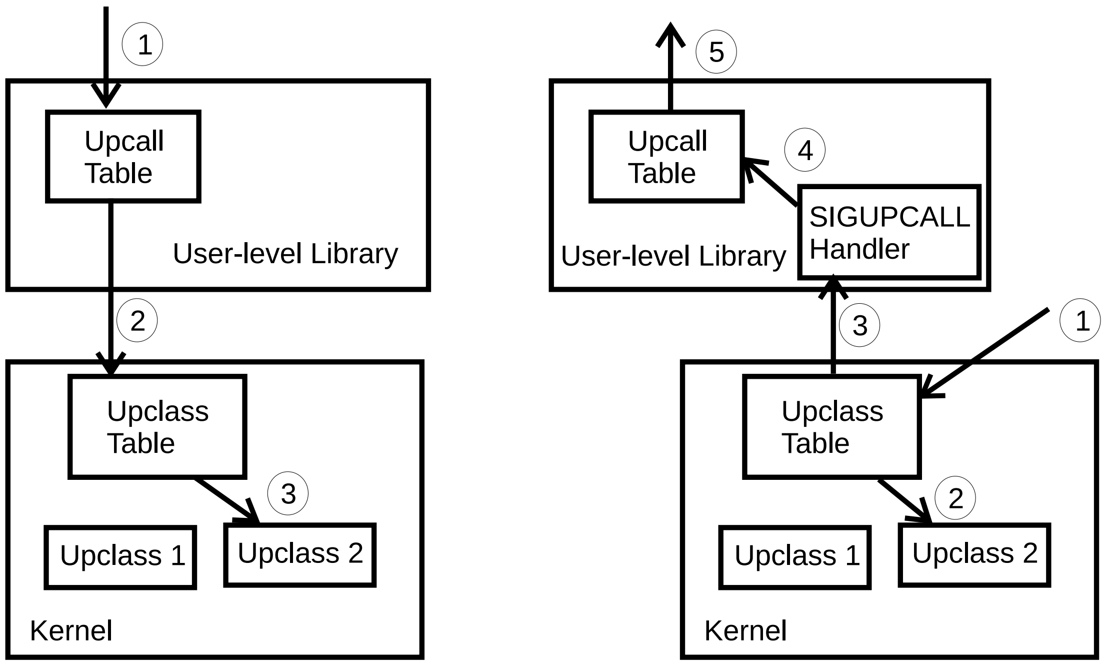
Upcalls, introduced in Section 3.3.5, are the foundation upon which resource requests are built. While they are more powerful than signals they are implemented using the signal facility. This implementation strategy makes use of the signal delivery process for invocation, but adds facilities for exactly-once, in-order delivery and the passing and returning of arguments. A clean separation between signals and upcalls is maintained; the former need know nothing about the latter.

A new signal, `SIGUPCALL` has been added to the set of signals that NetBSD supports, and ties together the upcall implementation. By default, this signal is ignored by applications. Using a new signal that is ignored by default minimizes the impact on programs which are not aware of upcalls.

The upcall implementation comprises four areas of functionality that support the design of Section 3.3.5. *Upclasses* divide the domain of upcalls into related groups and expose the details of upcall handling to individual subsystems. A process may declare upcall handlers through upcall *registration*. An upcall may be *posted* to a process that has declared a handler for that upcall. Finally, processes *handle* posted upcalls in the previously declared handler. This section gives a brief overview of these areas, and then describes each of them in turn.

An upclass is a collection of related upcalls, and a kernel subsystem which provides support for them. Each upclass is described through a function table, which describes the bookkeeping functions required of an upclass. The upcall infrastructure provides a set of default implementations for each of these functions; upclasses with no special needs can use them directly.

Upcall registrations are handled by a combination of kernel support and a user-level library. At initialization time, this library registers a signal handler for `SIGUPCALL`. When the process linked to the library requests an upcall registration, the library records mappings between the registered upcall and declared handler. The registration information is then forwarded to the kernel, which makes it available to individual upclasses for bookkeeping purposes. The upcall registration process is depicted in Figure 4.3(a). Regardless of the upclass's own bookkeeping, the upcall infrastructure, through `fork` and `exec`, maintains a list of upcalls registered for a process. This list is kept in the process structure.



(a) Registration (b) Invocation

Figure 4.3: Upcall Registration and Invocation

Upcalls may be posted by user-level processes via a system call, or through a set of routines internal to the kernel; both mechanisms use the same underlying code. Because signal-handling overhead can be significant, upcalls are posted only to processes willing to accept them. These posted upcalls are enqueued to each destination process; this queue, like the list of handled upcalls, is part of the process structure. The kernel then notifies these processes that an upcall is pending by posting a `SIGUPCALL` to them.

The `SIGUPCALL` handler in the user-level library catches the signal and enters a loop to handle all pending upcalls. For each such upcall, the library reads the upcall and arguments from the in-kernel queue, and forwards the request to the appropriate upcall handler. The library then passes the return code and result buffer, if any, back to the kernel, which returns the results to the caller. The invocation process is illustrated in Figure 4.3(b).

The remainder of this section describes the upcall implementation in detail. For reference, Figure 4.4 summarizes the upcall API, and, for each function, specifies whether the library or kernel implements it. This API is supported both by service routines in the library, and system calls intended to be used only by the library. These are summarized in Figures 4.5 and 4.6, respectively.

`upcall_reg`	Register an upcall handler. Library routine.
`upcall_block`	Block reception of upcalls. Library routine.
`upcall_unblock`	Unblock reception of upcalls. Library routine.
`upcall_sync`	Synchronously invoke an upcall. System call.
`upcall_async`	Asynchronously invoke an upcall. System call.

Figure 4.4: Upcall API Summary

`uclib_init`	Check for presence of upcall support in the kernel. Set up hash table that maps upcalls to handlers. Establish signal handler for `SIGUPCALL`
`catch_sigupcall`	Main routine for upcall handling. When notified by `SIGUPCALL` of pending upcalls, obtains them from the kernel, hands them to appropriate upcall handler, and returns results.

Figure 4.5: Upcall Library: Internal Routines

Upclasses

To preserve isolation between different kernel subsystems, and to allow individual subsystems access to the details of upcall handling, groups of related upcalls are collected into upclasses. Each upclass is represented in the kernel by an *upclass descriptor*, a table of functions that provides the basic bookkeeping actions required to register for and post upcalls. The key elements of this descriptor are presented in Figure 4.7.

At boot time, the kernel steps through each descriptor in the table, and calls the descriptor's `init` function. This function is responsible for setting up any upclass-specific

`_upcall_reg`	Called by `upcall_reg` in the library. Passes registration information from the library to the kernel
`_upcall_next`	Called by `handle_sigupcall`. Obtains the next upcall to be invoked, and the arguments if possible.
`_upcall_args`	Called by `handle_sigupcall`. Obtains arguments in cases where `_upcall_next` was unable to.
`_upcall_ret`	Called by `handle_sigupcall`. Tells kernel that library has finished this call. Passes return information back to the caller if necessary.

Figure 4.6: Private Upcall System Calls

`init`	Function to set up upclass's internal state. Called only at boot time. Result determines `inited`.
`inited`	True if initialized. Set only at boot time.
`notify`	Function to call when a process (de-) registers an upcall handler.
`rcpt`	Function to call to check if some process can receive a particular upcall.

Figure 4.7: Key Elements of an Upclass Descriptor

data structures; if it returns successfully, the kernel marks the descriptor as `inited`. If the function returns failure, or the descriptor is not filled in,[1] the descriptor is not marked `inited`. The remaining functions are described in the remainder of this section in the context of their use by other parts of the upcall implementation.

Upcall Registration

A request to register an upcall handler is handled first by the run-time library. If this is the first request by the application, the library calls `uclib_init`. This routine installs the `SIGUPCALL` handler, and initializes the hash table mapping upcalls to handlers.

From then on, when registrations are requested, the library records the address of the handler in the hash table, which is keyed by upcall number, and then forwards the registration to the kernel via `_upcall_reg`. This system call need carry only the upclass and upcall numbers, along with a flag specifying whether this is a registration or deregistration; the kernel does not record the address of upcall handlers.

Changes in upcall registrations are handled by the kernel routine `uc_doreg`; it is told the process for which the request is being made, the upclass and upcall numbers, and a flag

[1]Such situations typically arise when the subsystem implementing the upclass is not configured into the kernel.

specifying whether this is a registration or a deregistration. The upclass is notified of the request via its `notify` method, and is free to allow or disallow the registration change.

If the upclass allows the change in registration, it returns a *process entry* to be inserted into or removed from the process's list of registered-for upcalls. This list is stored in the process structure. It is used to provide upcall inheritance through `fork`, which walks the parent process's list of registered upcalls, registering an identical set for the child. A process's registrations are all removed on either `exec` or `exit`.

Upcall Posting

Upcalls may be posted either by user-level processes or the kernel. Synchronous upcalls may be posted only by processes, and not the kernel. This is because the poster of a synchronous upcall is put to sleep while the upcall is being processed. At invocation time, the kernel might be executing in a device driver or exception handler, with no corresponding process to suspend. Regardless of how they are invoked, they are handled similarly.

The posting routine first obtains the list of processes to which this upcall is to be posted by calling the upclass's `rcpt` routine; this list is called the *to-notify* list. When posting an asynchronous upcall, `rcpt` may return a list with many processes. In contrast, synchronous upcalls will obtain at most one process from `rcpt`; synchronous upcalls specify a particular pid, and there can be only one registered handler per upcall for a given process.

The upcall is then posted for each process in the to-notify list. This task is handled by `uc_post`, which places the upcall onto the tail of the process's pending upcall list, and then posts the `SIGUPCALL` signal to that process. Each element on this queue carries with it the arguments to be passed to the upcall handler, and, for synchronous upcalls, a buffer to receive the return values. The storage for arguments to upcalls that are posted to more than one process is shared amongst all instances of the posted upcall. The upcall subsystem manages this storage by reference count, and deallocates it when it is no longer needed.

Upcall Handling

When a `SIGUPCALL` is posted to a process accepting upcalls, the handler `catch_sigupcall` catches it. This handler loops, processing all outstanding upcalls pending for that process. This loop makes use of the last three system calls in Figure 4.6; these system calls are intended to be used only by the `catch_sigupcall` routine.

The first of these private system calls is `_upcall_next`. This system call returns the upclass and upcall numbers of the next upcall, as well as the sizes of the argument and result buffers. As an optimization, the library names an argument buffer which `_upcall_next` fills if the pending arguments are not too big to fit. If they are too big, the library routine allocates a buffer big enough to hold the pending arguments and then calls `_upcall_args` to obtain them. The library allocates a buffer large enough to hold any returned results.

Once the arguments are obtained, the library looks in the hash table for the function handling the in-progress upcall. The library calls the upcall handler with the argument and result buffers. The return value from the upcall handler, along with the result buffer, are passed back to the kernel via the third private system call, `_upcall_ret`. This last system call checks to see if this is the last invocation of this upcall, and deallocates the argument buffer if it was. It also checks to see if the upcall was invoked synchronously. If so, it wakes up the waiting caller so that it can retrieve the results.

The library also exports routines for blocking and unblocking the reception of upcalls. These routines are used internally during upcall registration. They may also be used by an application to provide critical regions during which no upcalls will be handled. These routines ensure that, if `upcall_block` is called more than once, an equal number of calls to `upcall_unblock` are required to resume upcall reception.

4.4.2 Interceptor

The interceptor acts as Odyssey's VFS component, gluing the Odyssey name space into the client's file system. The interceptor receives operations from VFS and forwards them to the viceroy. The interceptor can also satisfy simple requests on Odyssey objects without contacting the viceroy. The interceptor is based on Coda's MiniCache, which played the same role in Coda.

The communications channel between the interceptor and the viceroy is a two-way pipe implemented as a pseudo-device. As requests come in to the interceptor to be forwarded to the viceroy, they are written to the pseudo-device. The viceroy reads these requests, performs the operations, and writes the results back to the pseudo-device. The interceptor matches responses from the viceroy with their requests, and passes the results to the appropriate caller.

Upon initialization, the viceroy opens this pseudo-device; the device driver interprets this as a `mount` request, and mounts the interceptor as a VFS file system mounted at the root of the client's file system. From then on, any file system operations in the Odyssey portion of the name space are redirected by VFS to the interceptor. These redirected requests name the operation requested, the vnode upon which the operation should be performed, and the arguments to the operation, if any.

The per-vnode data maintained by odyssey is called an *onode*, An onode contains the object's `ofid`, a structure introduced in Section 3.3.4. There are four operations that can be satisfied entirely by the interceptor: `lookup`, `getattr`, `readlink`, and `rdwr`. To do so, the interceptor keeps a small, fixed-size cache of onodes. Onodes can be retrieved from the cache directly by `ofid`, or by their parent onode and the pathname component that names the child from the parent. When a `lookup` request arrives at the interceptor, it checks, for each pathname component, if the onode is already in the cache. If the eventual target onode is found, the cache can satisfy the request. If it is not found, the interceptor forwards the request to the viceroy, and enters the newly-discovered onode into the cache. Since UNIX processes show a high degree of pathname locality, the cache can eliminate most of the name translation overhead [76].

Once an onode is in the cache, attribute and symbolic link information also can be filled in. On receiving either a `getattr` or `readlink` operation, the interceptor first checks the cached onode for the information. If it is present, it is returned; if not, the operation is forwarded to the viceroy, and the results are cached for later use.[2]

Reads and writes can also be handled entirely by the interceptor for certain classes of ondoes; they are marked as *cacheable,* and are stored by the viceroy in a container file on the client's local disk. When such onodes are opened, the viceroy returns the on-disk device and inode numbers identifying the local container file holding that onode's contents. Later `rdwr` requests for that onode can be redirected to the local file system code, which performs the request directly on the container file. Wardens decide which onodes can be marked cacheable; for example, video streams are too large to be cached in their entirety, and thus `rdwr` requests must be forwarded to the viceroy.

The interceptor provides four vnode operations that are implemented only in the Odyssey portion of the file system. Three of these operations support the implementation of resource requests, the fourth supports type-specific operations; they are all described in Section 4.7. Because the BSD 4.4 implementation of VFS includes the stackable layers extension, Odyssey may implement these four in isolation, without requiring changes to the other VFS file system types.

4.5 Viceroy

The viceroy, introduced in Section 3.3.4, is the central component of the Odyssey prototype. It acts as the single point of resource control, and is responsible for type-independent functionality. It is uses a non-preemptive, user-level threads package. There are two tasks for with the viceroy is responsible: *operation dispatch*, and *bandwidth estimation*.

When an operation is enqueued to the pseudo-device by the interceptor, a *worker thread* dequeues it for processing. It examines the operand, and then forwards the request to that operand's warden, which performs the operation. This process of operation dispatch is described in Section 4.5.1

The focus of the prototype is adaptation with respect to *connection bandwidth*. The connection manager, described in Section 4.5.2 is responsible for providing network services to the wardens. The viceroy has a background thread, the *estimator*, that periodically estimates the bandwidth available to each of the connections in the connection manager.

4.5.1 Operation Dispatch

During initialization, the viceroy spawns a pool of worker threads, opens the pseudo-device, and calls `mount` to set up communications between the interceptor and the viceroy. After initialization, the main thread of the viceroy enters the *kernel loop*.

[2]Presently, the viceroy does not implement symbolic links, but the functionality to handle them in the interceptor exists.

In this loop, the main thread repeatedly calls `select`[3] on the pseudo-device. As requests come in to the interceptor, they are enqueued to the pseudo-device. The interceptor then marks the pseudo-device available for reading, which causes the main thread to return from its `select`. This thread pulls the request from the pseudo-device, and places it on the work queue.

The worker pool threads dequeue requests from the work queue as they arrive. When a worker thread receives a request, it examines the opcode of the request, and then decodes the arguments to the operation based on the opcode. The operands for the operation — the onodes — are reference-counted and locked, and then the operation is passed to the appropriate function in the warden's function table. Most of these functions form a subset of the vnode operations, and are described in Section 4.6. The remainder support Odyssey's API extensions, and are described in Section 4.7.

4.5.2 Communication Substrate

While the wardens are responsible for the contents of the communications between client and server, the mechanism over which this communication travels is the domain of the viceroy, and is provided by the the *connection manager*. The reason for this is twofold. First, the common tasks of setting up and maintaining connections are shared throughout the client, rather than requiring that each warden reimplement them. Second, and most importantly, this common substrate provides the mechanism by which the viceroy can serve as the point of network control. This section first outlines the services provided by the connection manager to wardens, and then describes the process of bandwidth estimation in detail.

Connection Management

As described in Section 3.3.4, the connection manager's key abstraction is the *connection*. Connections are named by their endpoints: `(host, service, number)`. Each connection may have only one outstanding request at a time; for parallelism, there may be more than one connection between a client and a particular service. Connections use RPC2 as their underlying transport mechanism. It provides both traditional remote procedure call as well as a sliding-window, selective-acknowledgement protocol for bulk transfer called SFTP. A summary of the connection manager's operations appear in Figure 4.8

Wardens can `create` and `destroy` connections. They must `get` an existing creation in order to use it, and `put` the connection when they are through with it. The connection manager uses reference counting to prevent the destruction of connections that are in use.

Once created, connections are long-lived. That is, the underlying RPC2 link between client and server may break and a new one later be established, but the connection persists across these changes, and appears to be a single connection to the wardens that use it.

[3]For concurrency concerns in the presence of our user-level threads package, this select is actually a veneer on a routine in the threads package which only calls the `select` system call in the event that no threads are able to run.

`create`	Create a new connection.
`destroy`	Destroy a connection.
`get`	Increment a connection’s reference count.
`put`	Decrement a connection’s reference count.
`isvalid`	Check to see if a connection is currently usable. Will not block.
`validate`	Attempt to ensure that a connection is currently usable. May block if connection must be reestablished.
`markdown`	Mark a connection as unusable.
`register`	Register a function to be called when a connection changes state. For example, a connection that was marked down was validated.
`unregister`	Remove a previously registered function.

Figure 4.8: Connection Manager Functions Exported to Wardens

Because connections are long-lived, they may be be unusable at times — situations where the underlying RPC2 link is broken. Thus, there is state associated with connections; a connection may be either *alive* or *dead*. There are two operations for wardens to examine or change this state. The first is `isvalid`, which returns immediately with the current state of the connection, but makes no attempt to resurrect dead connections. The second, `validate`, will try to resurrect a connection if it is dead, but will block while doing so; it should not be used in time-critical sections of a warden.

In the course of using a connection, a warden may discover it is dead. It uses the `markdown` function to make this known. If a warden wishes to be made aware of changes to a connection’s state, it may `register` a callback function with the connection manager. For example, a warden may need to re-validate any items cached from a previously inaccessible server when the connection to that server is reestablished. Whenever a connection changes state from alive to dead or vice versa, all registered functions will be called in the order in which they were registered. Wardens may also `unregister` previously registered functions.

Bandwidth Estimation

As wardens make use of connections to communicate with servers, RPC2 logs the sizes and elapsed times of both short exchanges and bulk transfers. The short, small remote procedure calls give an approximation of *round trip times*, while the long, large bulk transfers give an approximation of *throughput*. The viceroy periodically examines the recent transmission logs, and determines the instantaneous bandwidth available to the entire machine. It then estimates how much of that bandwidth is likely to be available to each connection in the coming period.

It is important to note that this is estimation, and not reservation or admission control. The viceroy makes no attempt to guarantee that its estimates will be met by the system. Rather, it assumes that over the short term, network usage and quality are unlikely to change; the definition of "short term" is the period of estimation. If, for concerns of agility, this period is too long, it can be shortened to take account of more fine-grained variations. The period used in the current prototype is 500 milliseconds.

Estimation is also purely passive; it depends only on observing traffic already present on the client. This is in contrast to approaches such as Keshav's packet-pair protocol [38], which add traffic to the network for the purposes of estimation. There are pros and cons to each approach. Passive estimation does not add traffic to what may be an already overtaxed network, but depends on the regular generation of traffic in the normal course of events; if the client infrequently generates traffic, the granularity of measurement will be correspondingly coarse. In contrast, active measurement does not rely on the presence of other traffic, but can reduce the overall utilization of the network.

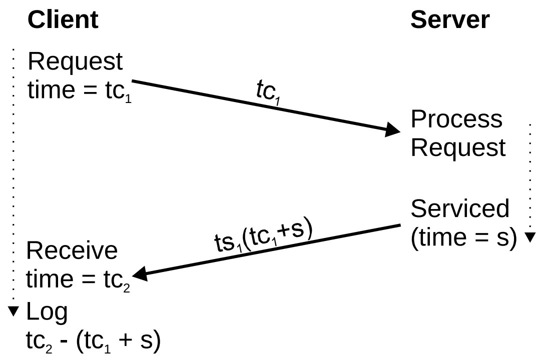
RPC2 logs As wardens perform remote procedure calls and bulk transfers, RPC2 observes the requests and responses to adjust its own retransmission timers. Requests are timestamped and responses echo timestamps, allowing RPC2 to estimate both round trip time, or RTT, as well as throughput; the RTT values influence the shortest retransmission interval. In addition to using this information internally, RPC2 logs its RTT and throughput information, and makes the contents of these logs available to higher levels of the system; in Odyssey's case, the viceroy. This section presents a brief overview of these logs. The details of how RPC2 calculates RTT and correspondingly adjusts retransmission timers has been described by Mummert [49].

On a client, each unique `(host, service)` pair maintains two separate logs: one for remote procedure call traffic, and one for bulk transfer. Multiple connections to the same `(host, service)` pair share logs. Each log contains *observations* in a fixed-size circular buffer. Each observation is of the form:

At time t, hosts C and S exchanged D bytes in e seconds.

where C is the client, S is the server, D is the size of the exchange, and t is the time to send the data and receive the acknowledgement that it arrived. These observations are made by timestamping request packets sent either by a host, and echoing those timestamps in response packets. Hosts only use internally generated or echoed timestamps; clocks on distinct hosts need not be synchronized.

Log entries mean different things for each of the two protocols. For remote procedure calls, shown in Figure 4.9, the logs represent the time for a single, short exchange, minus any server-side delays. The client sends a request to the server of size d_q at time tc_1, and the request packet is stamped with that time. The server, on receipt of that request, saves the timestamp and processes the request. The time taken for this processing, s, is measured by the server. The server sends the response, which is of size d_r, and echoes the timestamp $tc_1 + s$. The client receives this response at time tc_2, and logs the observation that $D = d_q + d_r$ bytes were exchanged in time $tc_2 - (tc_1 + s)$.



This figure illustrates the contents of RPC2's round-trip time logs. The client is on the left, the server on the right; time progresses form the top of the figure to the bottom. Packets are exchanged on arrows; timestamps are shown, and where timestamps are echoed, they are shown in parentheses. At time tc_1, the client sends a request to the server. The server takes s time to respond to that server, and send the response at time ts_1, echoing the timestamp $tc_1 + s$.

Figure 4.9: Remote Procedure Call Logs

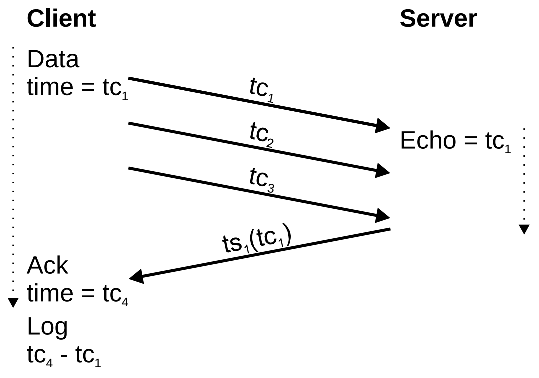
Note that the time recorded in the observation excludes the processing time at the server, s. Ignoring this time provides isolation from computational load at the server; the recorded duration is only the time required to exchange packets. Since it is a measure of elapsed time, the term s may be used on the client in the absence of synchronized clocks.

Note also that the number of bytes in the observation is the sum of the request and response packets, not including headers. This total is expected to be small, and is limited to twice the maximum RPC2 packet size; in our implementation, this is 4500 bytes.

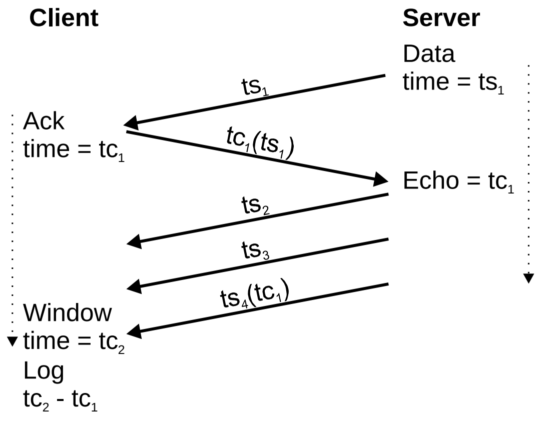
The client's bulk-transfer logs record the time it takes to send a window's worth of data from sink to source, plus either an acknowledgement of or request for data. Exactly which of these is recorded depends on whether the client is acting as the source or the sink. These two situations are shown in Figure 4.10.

Figure 4.10(a) depicts the client as the source of the bulk transfer. In the top of the figure the client is ready to send a new window's worth of data, D. Suppose it takes three packets to transfer all D bytes. The first portion of that window, d_1 bytes, is sent at time tc_1; the packet is stamped with that time, and marked. When the server receives this marked packet, it remembers the timestamp the packet carries for later echo. Meanwhile, the client sends the remaining packets in the window, d_2 bytes at tc_2, and d_3 bytes at tc_3. On receipt of the last packet in the window, the server sends an acknowledgement for the window, and echoes the remembered timestamp tc_1. This acknowledgement is received by the client at time tc_4, which logs the observation that $D = d_1 + d_2 + d_3$ bytes of data were sent and acknowledged received in time $tc_4 - tc_1$.

When the client is the data sink, the situation is only slightly different. Upon receiving the last packet in a window, the client sends an acknowledgement of that window at tc_1. When the server receives this acknowledgement, it remembers the timestamp, and begins



(a) Client as Source



(b) Client as Sink

Each of these figures illustrate the contents of RPC2's throughput logs. In each the client is on the left, the server on the right; time progresses form the top of the figure to the bottom. Packets are exchanged on arrows; timestamps are shown, and where timestamps are echoed, they are shown in parentheses.

Figure 4.10: Bulk Transfer Logs

the next window. When the server sends the last packet in the window, it echoes the remembered timestamp tc_1. Upon receipt of this last packet at time tc_2, the client logs the observation that D bytes were requested and received in time $tc_2 - tc_1$.

In the client logs, there is no way to differentiate between sink and source transfers. However, assuming the presence of a symmetric network, no such distinction need be made. Each case, sink or source, measures the transfer of a full window's worth of data, and one half of a round trip time; the order of these events is immaterial.

The two kinds of observation logs — remote procedure call and bulk transfer — estimate round trip time and throughput respectively. From these, one can determine bandwidth in the following way, provided one is willing to assume symmetric network performance. Consider the case with only a single remote procedure call entry, and a single bulk transfer window from client to server. The latter measures the time to transfer data in one direction, plus the time to send an acknowledgement. Subtracting the time for the acknowledgement — one half of a round trip time — yields the time to transfer the data only; the size of the transfer divided by the time to make that transfer gives the bandwidth. So, given a round trip time, R, and a throughput observation, D bytes in t seconds, the bandwidth was:

$$\frac{D}{t - R/2} \tag{4.1}$$

The remainder of this section describes how this simple idea is used in estimating the bandwidth available both to the client as a whole as well as individual connections on the client.

The viceroy periodically collects logs for all active connections on the client.[4] The first step is to estimate the round trip time to each connection's server; these will be used to convert each connection's throughput observations to bandwidth estimations. The second step is to build a *master log* that captures the immediate past history of the client as a whole, correctly accounting for parallelism amongst connections. The third step uses the observations from this master log to estimate the total bandwidth available to the client. Finally, the fourth step estimates the portion of this total bandwidth that will be available to each individual connection.

Each remote procedure call is small, and, unlike bulk transfers, they are assumed to not interfere with one another while in transit. They are also assumed to be small enough to be significantly delayed by only latency considerations, not bandwidth.[5] Since service time is excluded from the observations in the remote procedure call logs, each observation in those logs represents the instantaneous round trip time for that connection.

Rather than take the most recent remote procedure call observation for each connection as its round trip time, the viceroy applies two forms of smoothing. In the first filter, the log is scanned oldest to youngest, accumulating an estimate of the round trip time with the following recurrence relation:

$$R_i = \begin{cases} O_1 & \text{if } i = 1, \\ \alpha O_i + (1-\alpha) R_{i-1} & \text{if } i > 1. \end{cases} \tag{4.2}$$

where R_i is the round trip time calculated at stage i, and O_i is the ith most recent observation in the log. The value of α is 0.75, which results in a fairly aggressive smoothing function. This value was chosen to give as agile an estimator as possible.

Our user-level package can occasionally produce anomalously high RPC times. In order to discount this in the presence of the aggressive estimator of Equation 4.2, the viceroy caps the maximum rise in the RTT to at most 1/2 the previous RTT value; this provides the second filter. Thus, Equation 4.2 can be rewritten as:

$$R_i = \begin{cases} O_1 & \text{if } i = 1, \\ \alpha O_i + (1-\alpha) R_{i-1} & \text{if } i > 1 \text{ and } (\alpha O_i + (1-\alpha) R_{i-1}) < 1.5 R_{i-1}, \\ 1.5 R_{i-1} & \text{otherwise.} \end{cases} \tag{4.3}$$

The need for this second filter is unfortunate; two alternatives suggest themselves. First, the source of these unusually long request-response times could be found and corrected. Second, the smoothing function might be made slightly less aggressive, retaining agility while providing improved stability. However, the main effect of this filter is to occasionally produce overly conservative bandwidth estimates; therefore, the viceroy retains it.

[4]Because some connections may be to the same remote `(host, service)` pair, some of the logs may be identical; as described later in this section, these identical logs are filtered as a side effect of the estimation algorithm.

[5]While not strictly true, this simplifying assumption turns out to yield good results in practice.

Because remote procedure calls are small, they are assumed to not interfere with one another. Therefore, the two filters described by Equation 4.3 are applied to each connection independently. This associates a single round trip time with each connection.

```
build_master(all_logs)
    while (!empty(master_log))
        record = get_youngest_entry(all_logs)
        if (!pending_record)
            pending_record = record
        else
            if (equal(last_record, record))
                continue
            endif
            if (overlap(pending_record, record))
                pending_record = merge(pending_record,
                                       record)
            else
                enter(pending_record, master_log)
                pending_record = record
            endif
            last_record = record
        endif
    endwhile
```

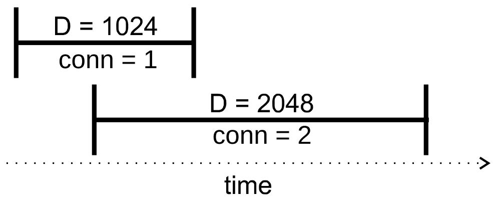
This figure shows pseudo-code for building the master log from the bulk transfer logs from each individual connection.

Figure 4.11: Master Log Generation

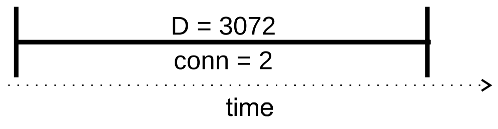
The next step is to create a single bulk transfer log that correctly describes the recent bulk transfers performed by the client as a whole. Figure 4.11 gives the pseudo-code for building this master log. At each step in building the log, the viceroy extracts the most recent entry from the set of bulk transfer logs. If this record is identical to the previously extracted record, it is discarded; this can arise if two or more connections are to the same `(host, service)` endpoint, and hence have the same log contents.

If this record is not identical to the previous one, it is checked against the *pending record* for overlap; the pending record was built in the previous iteration through the algorithm. If the current record and the pending record do not overlap, the pending record is placed on the tail of the log, and the current record becomes the pending record.

However, if the current record and pending record overlap, they are merged, and the merged record becomes the new pending record. The record resulting from the merge of the pending and current records begins when the pending record did, and ends when the current record did, has a size that is the sum of the two merged records, and is marked as belonging to the later records connection. This merge process is depicted in Figure 4.12.



(a) Before Merge



(b) After Merge

Figure 4.12: Merging Log Records

Once the master log has been constructed, the viceroy estimates the total bandwidth as shown in that log; to do so, it applies a filter similar to that in Equation 4.2. Each log record in the master log gives throughput; to convert to bandwidth, we must subtract the time for the acknowledgement or request packet in Figures 4.10(a) and 4.10(b), one half of the round trip time. Thus, each bandwidth observation, B, is:

$$B = \frac{D}{t - R/2} \tag{4.4}$$

where D is the number of bytes in the log record, t is the time recorded in the log record, and R is the round trip time of the connection marked as owning this record. It is here that an overly-optimistic round trip time manifests itself as an overly-pessimistic bandwidth estimate; the denominator of Equation 4.4 is too large. Since applications make fidelity decisions on this estimate, unwarranted pessimism is preferable to unwarranted optimism.

The bandwidth observations in the master log are smoothed according to a recurrence relation similar to that in Equation 4.2, but with an α value of 0.875. This is an even more aggressive smoothing function than Equation 4.2, biasing heavily in favor of recent observation to be as aggressive as possible. The resulting bandwidth is considered to be the bandwidth available to the entire machine.

Per-Connection Estimation After producing an estimate of total bandwidth, the viceroy must estimate how much of this total bandwidth will be available to each connection. The actual bandwidth along each connection is unknown; recent observation of each connection's performance is the only clue to actual bandwidth. Further, the sum of the bandwidths available to each connection cannot be more than the total estimated bandwidth to the machine.

To divide bandwidth, the viceroy bases its estimate of the bandwidth available to each connection on recent use; this is determined by examining each connection's individual log, deriving its apparent bandwidth as is done for the master log. Connections along constrained paths will have lower per-connection bandwidth than connections with along faster paths.

However, the viceroy cannot rely on recent use alone. Consider an application that, by choice, has not used the network recently. The connection supporting this application may well be capable of carrying significant network traffic. However, if recent use were the only factor, the application would receive an estimate of zero bandwidth from the viceroy. Thus, some fraction of the bandwidth must be allocated fairly among all connections.

The current implementation uses 80% as the amount to divide proportionally, with the remaining 20% divided fairly. When the bandwidth accounted for in the individual logs is less than 80% of the total, the excess is divided fairly as well. In no event is more than 80% divided based on recent use.

This rule is a simple heuristic, and seems to work well in practice. The key shortcoming is the handling of connections that have been idle for some time. There is also some question of whether this strategy unduly rewards greedy connections. While these are both fair criticisms, heuristics such as the 80/20 rule are the best available approach without allowing some form of active probing of the network.

One could imagine allocating excess bandwidth according to need rather than past performance. For example, the upper bounds of any tolerance windows could be used to decide which connections could benefit from such an excess. However, such strategies reduce to the question, “How should one arbitrate between competing applications?” In the context of mobile computing, where devices are carried by individual users, this arbitration must involve the end user. Such a scheme is beyond the scope of this work; it is posed as an area of future research in Section 9.2.4.

4.6 Wardens

The wardens provide type-specific functionality for an Odyssey client, one warden per type. Because the types are very different from one another, the details of each wardens implementation will also differ. Despite this, all wardens share a common structure that is described in this section.

The central element of each warden is its *warden table*; this table is exported by each warden to the viceroy, and lists the functions that the warden provides to implement administrative, vnode, and odyssey-specific operations, or *methods*. Each of these methods is gathered together in an array of such tables, `wardens`. This section describes the administrative and vnode operations; discussion of the odyssey-specific operations is deferred to Section 4.7, where Odyssey’s API extensions are presented in their entirety.

4.6.1 Administrative Operations

The four administrative functions, originally presented in Figure 3.7, are reproduced for reference in Figure 4.13. Two of these are used exclusively at client start-up. The third is for debugging and measurement, while the fourth is the only one of the administrative functions used during the course of normal Odyssey operations.

`parseargs`	Given a partial list of arguments in `argv`, try to parse the next and subsequent arguments. Return the number of arguments consumed. Only function that may be called before `init`
`init`	Set up the warden's internal data structures, if any.
`pstats`	Print summary statistics to a particular file.
`getcid`	Given one of this warden's onodes, return the connection to the server storing that onode.

Figure 4.13: Warden Table: Administrative Operations

The Odyssey client — the viceroy and wardens — may be passed arguments when it is started. Some of these arguments are meant for the viceroy. For example the period to use for bandwidth estimation. Others are meant for individual wardens. The viceroy, when it encounters an argument it does not understand, passes the unprocessed arguments — a suffix of the full argument list — to each of the wardens in turn through the `parseargs` method.

When a warden receives this partial argument list, it looks at a prefix of the passed list. If the warden recognizes some prefix of the list, it returns to the viceroy the size s of that prefix list. The viceroy then skips those s arguments and resumes processing with the remainder. If the warden does not recognize the first element of the passed list, it returns zero, and the next warden in the `wardens` table is given the arguments. If the warden does recognize some prefix, but finds that they are malformed in some way, it returns negative one, and the viceroy exits; printing its own usage message as well as those provided by each warden in `wardens`.

After all arguments have been recognized, the viceroy calls the `init` method of each warden in turn. This method sets up any internal state the warden will need in order to satisfy requests. Initialization may include preallocating large buffers, contacting servers, or the like. If the warden's `init` method returns successfully, the warden will be marked ready, and further methods may be invoked on that warden; otherwise, the warden will not be so marked, and objects of that warden's type will be inaccessible. The viceroy guarantees that no operation other than `parseargs` will be invoked before `init` returns successfully.

For purposes of debugging or data collection, the viceroy may periodically need to record data for later examination; it is useful to have the wardens do so as well. At such times, the `pstats` method is called by the viceroy, which passes in a file as an argument. The warden then prints diagnostic information to that file, and returns.

The final administrative operation, `getcid`, is passed an onode; the warden is to return the connection used to communicate with the server storing that onode. The `getcid` operation is used by the viceroy in handling resource requests on network bandwidth, a process described in Section 4.7.1.

4.6.2 Vnode Operations

Odyssey's wardens implement a subset of the vnode operations defined by the BSD 4.4 implementation of VFS. Because the focus of the prototype was on adaptation, Odyssey provides only the operations necessary to explore adaptive applications. This subset, originally presented in Figure 3.7, is reproduced in Figure 4.14.

`lookup`	Given an onode and a pathname component, return the named onode.
`root`	Obtain the root object of a tome.
`access`	Given an onode, a credential, and an operation, return successfully if the user named by the credential is allowed to perform the operation on the onode.
`getattr`	Return the meta-data (attributes) of an onode).
`open`	Open an onode.
`close`	Close an onode.
`rdwr`	Read a buffer from or write a buffer to an onode. The warden is responsible for allocating the buffer which read operations are to fill, allowing it to satisfy reads from prefetched buffers.
`rdwrdisp`	Dispose a buffer previously returned by `rdwr`
`readdir`	Read a portion of a directory's contents.
`readdirdisp`	Dispose a buffer previously returned by `readdir`.

Figure 4.14: Warden Table: Vnode Operations

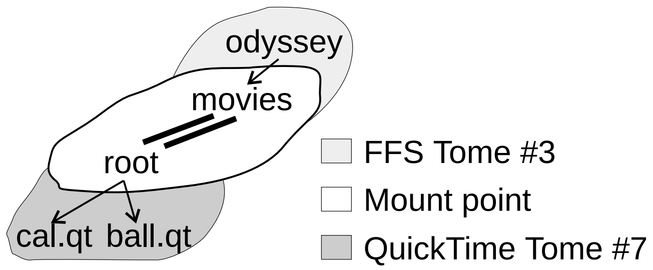
The first operation, `lookup`, is the most complex of the vnode operations supported by Odyssey wardens. It is passed a *directory* onode, and a pathname component; it is to look up that pathname component in the context of the directory onode, and return the named onode. If no such onode exists, the warden is expected to return a suitable error code.

The warden, when yielding an onode in lookup, must identify whether it is a directory or a *regular file*. Designating an onode as a directory means only that it may be used as the context for a later `lookup` operation; regular files cannot be used as lookup contexts. Since such operations are handled by the warden itself, the onode's contents need not correspond to any particular notion of a directory; it need not even use hierarchical naming schemes.

The component-by-component nature of name translation does impose some restrictions on pathnames used by individual wardens; in particular, the character `'/'` is used to separate components, and cannot be part of a component name. The implications of this restriction on the Web warden are discussed in Section 5.3.

While individual name translations are handled by wardens, crossing tome mount points is the province of the viceroy. Each warden that allows other's tomes to be mounted in its own is free to represent mount points as it chooses. Conceptually, a mount point is an onode in the mounted-upon tome that is *covered* by the root of the mounted tome. Upon returning a covered onode, the warden marks it as such, and includes the type and tome identifier of the mounted tome in the meta-data of the onode.

On return from the warden's `lookup` method, the viceroy checks to see if the returned onode is covered. If so, it invokes the `root` method of the warden responsible for the mounted tome's type, and passes the mounted tome's identifier. The warden returns the root onode for that tome, a pointer to which is cached in the covered onode. This is used to speed later crossings of this tome boundary.



This figure shows a simple name space used to illustrate the process of tome boundary crossing. The space contains two tomes. The first is tome number 3 of the Fast File System type, the second is tome number 7 of the QuickTime type. The QuickTime tome is rooted by the unnamed onode `root`, and is mounted on the onode named by `movies`

Figure 4.15: Tome Boundary Crossing

Figure 4.15 depicts a simple example of a tome mounted within another. Suppose the Fast File System, or FFS, warden were asked to resolve the name `movies` in the context of the onode `odyssey`. It would obtain the onode `movies`, which is part of the tome rooted at `odyssey`; however, that onode is covered by tome 7 of the QuickTime type. The FFS warden returns `movies` marked as covered, with the both the QuickTime type identifier and the tome number 7 in its meta-data. The viceroy, on discovering that the `movies` onode is marked as covered, asks the QuickTime warden, via its `root` method, for the root onode of tome 7.

The `access` and `getattr` methods are both simple. A warden's `access` method is passed an onode, a user credential, and one of three possible operations: read, write, or execute.[6] The warden is to return successfully if the named user is allowed to perform the operation on the onode.

The `getattr` operation is passed an onode, and is expected to fill in an *attribute* structure for it. This function is supported to enable standard UNIX applications which use the `stat` system call, such as `ls`, to work within Odyssey. Wardens are expected to fill in as much of the attribute structure as is meaningful for their data.

The `open` and `close` operations work together as a pair. Wardens are notified of every `open` and `close` on objects in their tomes so that they can perform whatever internal

[6]The "execute" operation is interpreted differently for files and directories; for the former, it means "execute the file," for the latter, it means "perform a lookup in this directory."

bookkeeping might be necessary for caching or other activities. The viceroy manages the reference counting of onodes themselves, to prevent the flushing of an active onode.

At `open` time, the warden can aggressively fetch the entire contents of the opened onode, and store the contents in a *container file* on the client's local disk. It marks the onode as cacheable, as mentioned in Section 4.4.2, and returns in the meta-data of the onode the number of the device on which the container file is kept, and the inode number of the container file on that device. This information is returned to the interceptor by the viceroy; the interceptor can then satisfy future reads and writes by forwarding them directly to the VFS file system responsible for the container file. On the `close` of a cacheable onode that was open for writing, the warden is expected to flush the dirty container file back to the server storing that onode; since the warden is not aware of individual writes, it cannot forward them as they happen.

There are situations where an entire object cannot be stored on disk at `open` time; such onodes are marked uncacheable. Because there is no container file for them, reads and writes on uncacheable onodes cannot be satisfied by the interceptor. Instead, they must be forwarded to the warden responsible for that onode. For example, objects on servers accessible only by a very slow link should be streamed to the application using them. Likewise, very large objects should always be streamed, and never stored in their entirety.

Reads from and writes to uncacheable onodes are satisfied the appropriate warden's `rdwr` method. For writes, the warden is passed a byte offset into the object, a buffer of new data, and the length of that buffer; it applies the update to the object. Since the warden sees individual updates to uncacheable onodes, it may send those updates to the server at any time, not just at `close`.

For reads, the warden is asked to return some number of bytes starting at a particular offset. The warden is responsible for allocating and returning the buffer holding the result. This allows a warden to return data that has been prefetched without forcing an extra copy of that data. The `rdwrdisp` operation is used by the viceroy to dispose of this read buffer after the results have been returned to the interceptor.

The `readdir` and `readdirdisp`[7] are analogous to `rdwr` and `rdwrdisp`, but are used to satisfy reads from directories rather than regular files. If a warden that implements a directory structure unlike that standard UNIX is to return valid data for `readdir`, it must coerce its own format into that expected by the NetBSD kernel.

4.7 API Extensions

Odyssey provides two API extensions to the standard UNIX interface: *resource requests*, and *type-specific operations*. These two involve each of the preceding components described in this chapter, and are used by Odyssey applications to gain awareness of their

[7] As of this writing, `readdir` does not supply its own buffer, but rather uses one created by the viceroy. The benchmarks used to evaluate Odyssey make no use of `readdir`, however, and thus are not effected by this slight discrepancy.

environment, and react to significant changes to it. Each extension is described from the point of view of its use by an application.

4.7.1 Resource Requests

The resource request API, introduced in Figure 3.4, is implemented by a combination of a run-time library, kernel support, and the viceroy, and includes extensions to the vnode interface that are implemented solely by the Odyssey VFS driver. The implementation also makes use of the upcall subsystem for notification delivery.

Only the application and viceroy need to agree on resource requests; the kernel itself need know nothing about the details of individual requests. Furthermore, the sizes and types of notification arguments, shown in Figure 3.4, are always the same. Therefore, all notifications, and hence requests, are multiplexed onto a single upcall, `ODY_NOTIFICATION`, just as all upcalls are multiplexed onto one signal. The library maintains a hash table mapping resource requests to registered handlers.

This simplifies the implementation of resource requests, and minimizes the interactions between the kernel, application, and viceroy during request placement and notification. The runtime library need only register for a single upcall during initialization; all remaining work for request handling is between the application and the viceroy. Resource requests are forwarded to the viceroy, which records them in structures called *resource request lists*. When, in the course of monitoring resource availability, the viceroy discovers that granted requests are violated, it uses the upcall subsystem directly to notify the requesting application.

The remainder of this section describes the implementation details of resource requests, by examining request placement, cancellation, and notification in turn. Each of these activities involves many different Odyssey components. For reference, Figure 4.16 summarizes the public and private library routines supporting upcalls; reception of upcalls is blocked during all of these routines. Figure 4.17 lists the system calls supporting the library's API implementation, one system call per function in the API. Finally, Figure 4.18 lists the vnode operations added by Odyssey in support of requests.

`ody_request`	Place a resource request; reference item named by pathname.
`ody_frequest`	Place a resource request; reference item named by file descriptor.
`ody_cancel`	Cancel a request.
`init`	Set up hash table, register upcall handler. Private.
`uchdlr`	Routine which handles notification upcalls; dispatches them to the appropriate handler. Private.

Figure 4.16: Request Library Routines

`_ody_req`	Tell kernel about a resource request; reference item named by pathname. Called by `ody_request`.
`_ody_freq`	Tell kernel about a resource request; reference item named by file descriptor. Called by `ody_frequest`.
`_ody_cancel`	Tell kernel/viceroy about a cancelled request. Called by `ody_cancel`.

Figure 4.17: Request System Calls

`vop_req`	Forward a request to the viceroy. Called by `_ody_req` and `_ody_freq`.
`vop_cancel`	Forward a cancel to the viceroy. Called by `_ody_cancel`.
`vop_cancpid`	Tell viceroy to drop all requests for a particular process. Called by the kernel's internal `exit` and `exec` routines.

Figure 4.18: Request VFS Operations in the Interceptor

Request Placement

When an application makes a resource request, that request is first fielded by the library. If this is the first request ever placed by this application, the library first calls `init`. This function sets up the hash table that maps notifications to handlers, and registers `uchdlr` to handle the single upcall supported by the Odyssey upclass.

Once the library has initialized itself, the request is passed to the kernel by either `_ody_req` or `_ody_freq`. The `_ody_req` routine calls `namei` to translate the pathname to a vnode. This will make use of the interceptor's cached name translations if present, or will call the viceroy and appropriate wardens if they are not. The `_ody_freq` routine looks in the process's open file table for the vnode backing the named file descriptor.

Once a vnode is in hand, these routines call `vop_req`. This operation is only implemented for Odyssey vnodes; other file system types will return an error code. The `vop_req` routine is part of the interceptor; it packages the request for the viceroy and forwards it.

The viceroy obtains the request, and decodes it to see which resource is requested. Recall from Section 3.3.2 that network bandwidth — the focus of the prototype — is an item-specific resource; that is, it must be evaluated in the context of a particular onode. This is because different onodes might be stored on different servers, and the bandwidth available might differ from server to server. Thus, when a new bandwidth request arrives at the viceroy, it must first determine the connection to which the request applies. It does so by invoking the onode's warden, via the `getcid` operation in Figure 4.13. This operation, given an onode, returns the connection used by the warden to communicate with that onode's server.

Each distinct resource in Odyssey keeps a list of previously granted requests called *resource request lists*, or RRLs. In addition to previously placed requests, an RRL has as state the current value of the resource. The key operations provided for RRLs are summarized in Figure 4.19.

`insert`	Given a resource request list, and a request, attempt to place the request on the list. If the current value of the resource is within the window, grant the request and return a request identifier; otherwise, deny the request, and return the resource's current value.
`set_value`	Change the resource's current value. If this new value violates any previously granted requests, return them so that they can be notified.
`cancel`	Remove a specific request, named by its identifier, from the resource request list it is on.
`purge`	Remove all requests placed by a particular process.

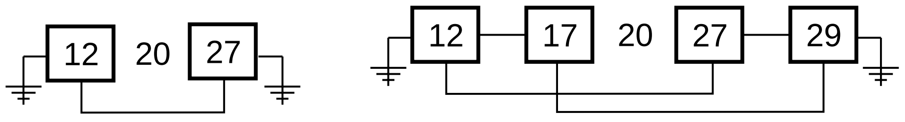
Figure 4.19: Vnode Operations in an Upclass Descriptor

Once the viceroy has the correct RRL in hand, it tries to `insert` the new request into the RRL. The new request's bounds are checked against the RRL's current resource value. If the value is out of bounds, the viceroy returns an error code along with this current value; these are propagated back through the call chain to the requesting applications. Otherwise, the request is placed in the RRL.

The viceroy adjusts its estimates for all resources frequently, but applications place resource requests relatively rarely. This means that updates to the resource value in an RRL, via `set_value`, will be a much more common event than the placement of requests on that list. Therefore, RRLs are optimized for `set_value` rather than `insert`. RRLs are implemented as two sorted lists of request bounds, one descending and one ascending. Each previously granted request that is still valid has its lower bound on the descending list and its upper bound on the ascending one; each pair of bounds is linked together.

All requests in all RRLs are also indexed by unique identifier, as well as by requesting process. The data structures providing these access paths are implicitly managed by the routines in Figure 4.19. This facilitates cancellation of previously held requests, as discussed in the next section.

Figure 4.20 illustrates an example of insertion into a resource request list. The viceroy would like to insert a request with bounds (17, 29) into the list in Figure 4.20(a). The `request` operation first checks to see that the current value of the list is within the requested bounds. It is, so the lower bound is insertion-sorted into the descending list, and likewise for the upper bound in the ascending list. The result is shown in Figure 4.20(b). Since valid requests enclose the resource's value in their bounds, the largest value descending list is less than the resource, while the smallest value in the ascending one is greater. In the common case of updating the resource value without invalidating a request, only two



(a) Before Insertion (b) After Insertion

This figure illustrates a simple resource request list. In Figure 4.20(a), only a single request with bounds (12, 27) is in the list, whose current value is 20. A later request of (17, 29) can be inserted, since the list's current value is within the request bounds. The list with the new request inserted is shown in Figure 4.20(b). Notice that entries in the two sorted lists that are from the same request are linked with one another.

Figure 4.20: Illustration of Resource Request Lists

comparisons are required.

When a request is successfully inserted, `insert` returns a new request identifier. These are guaranteed to be unique across all resources. This identifier is passed back through the call chain to the Odyssey library, which records the identifier along with the handler to be invoked if the application is notified that the granted request is violated.

Request Cancellation

A previously granted request can be removed in one of two ways. First, an application may explicitly cancel a previously granted request through `ody_cancel`. Second, all of an application's outstanding requests are implicitly cancelled on either `exit` or `exec`.

Individual requests are cancelled through the library's `ody_cancel` routine, which immediately forwards the request to the kernel using the system call `_ody_cancel`. This routine invokes `vop_cancel` on a special file called the *control file*, forwarding along the identifier of the request to cancel. The control file is a hidden, nameless file in the root of the Odyssey name space that is used for two purposes. First, system calls that, for whatever reason, do not have a reference to an Odyssey vnode may use it to forward general Odyssey operations to the interceptor, and thence the viceroy. Second, the interceptor itself may need to send messages to the viceroy that are not associated with any particular Odyssey object.

When the viceroy receives the cancel request forwarded from the interceptor, it uses the `cancel` operation to remove the cancelled request from the RRL it resides on. Note that `cancel` operates on all RRLs in the system, rather than just one. This is because the application requesting cancellation cannot know which RRL the request had been kept on.

After cancellation, a successful return code is propagated back through the call chain to the library. The library then removes its record of this identifier/handler pair, and returns success.

Process-wide cancellation is handled similarly. A process that, at any time, attempted to place a request will have an upcall handler registered for `ODY_UPCALL`. When that process exits or calls `exec`, its handler for that upcall is deregistered by the upcall subsystem. When this happens, the Odyssey upclass descriptor's `notify` routine is invoked; this routine must cancel that process's outstanding requests.[8]

It does so by invoking the `vop_cancpid` operation on the control file. This operation passes the process identifier for which the deregistration is being handled up to the viceroy. The viceroy then calls `purge`, which removes any requests placed by that process. Just as with `cancel`, `purge` must look at all RRLs, since a process may have requests on more than one resource.

Notifications

When the viceroy updates an RRL's current resource value, `set_value` will return a list of previous requests that are no longer met, called the *notify list*. This list of requests is passed to `do_notify`, a private routine in the viceroy; its job is to post the required notifications to each process with a request in the notify list.

For each request in the notify list, the viceroy extracts the process identifier from the request, and posts an `ODY_NOTIFICATION` upcall to that process. This upcall is posted asynchronously so that it won't unnecessarily delay the notifications to later processes in the list.

```
struct ody_ucargs_t {
    int          oua_reqid;
    unsigned int oua_rsrc;
    int          oua_rval;
};
```

Figure 4.21: Arguments for `ODY_NOTIFICATION`

When a notified process next is able to receive signals, that process's `catch_upcall` routine will field the posted upcall. The arguments to this upcall appear in Figure 4.21; there are no return values. The `catch_upcall` routine extracts the arguments, and removes the record of this identifier and associated handler from its hash table. It then calls the handler, passing the request identifier, resource identifier, and resource value.

4.7.2 Type-Specific Operations

The implementation of type-specific operations, the API for which was introduced in Figure 3.8, is straightforward. Both functions in the API are implemented as system calls. Figure 4.22 summarizes these system calls, the single vnode operation added by Odyssey to support them, and the two functions in the warden table which together implement them.

[8]Of course, the `notify` routine also is called when the handler is initially registered, but it doesn't perform any special handling at registration time. It is also possible that no such outstanding requests are currently in place; however, since the kernel does not keep track of resource requests, there is no way to know whether or not that is the case. Therefore, the deregistration is merely a hint that requests need to be cancelled, and the kernel conservatively informs the viceroy.

`ody_tsop`	Perform a type-specific operation on some Odyssey object; this object is named by pathname.
`ody_ftsop`	Perform a type-specific operation on some Odyssey object; this object is named by file descriptor.
`vop_tsop`	Vnode operation called by both `ody_tsop` and `ody_ftsop`. Packages arguments and forwards them to the viceroy.
`tsop`	Warden's function to handle type-specific operations. Responsible for allocating result buffer, if any.
`tsopdisp`	Warden's function to dispose result buffers returned by `tsop`.

This figure lists the functions that together implement type-specific operations. There are three levels of the system providing these functions, and the figure is divided by level. The top portion of the figure lists the two system calls applications can use to request a type-specific operation be performed. The middle portion lists the single VFS operation Odyssey adds in support of these two system calls. The bottom portion lists the two functions in the warden table used to satisfy these operations.

Figure 4.22: Vnode Operations in an Upclass Descriptor

To invoke a type-specific operation, an application may use one of two calls; the only difference between them is the way in which an application specifies the object upon which to operate. The first call, `ody_tsop`, names the object by pathname, the second, `ody_ftsop`, by open file descriptor. Upon entry into the kernel, these translate pathname or file descriptor into a vnode in the same manner as `_ody_req` and `_ody_freq`, described in Section 4.7.1.

These two then invoke `vop_tsop` on the resulting vnode, passing along arguments and a result buffer. As with the three VFS operations in Figure 4.18, `vop_tsop` is implemented only for Odyssey's vnodes. This interceptor routine packages the arguments and delivers the request to the viceroy.

The viceroy checks the arguments for validity, and then looks at the type field of the operand's `ofid`. It looks up the warden table for that type, and passes the type-specific operation to that warden's `tsop` method.

This method then carries out the requested operation. If there are any results to be returned, the `tsop` method is obligated to supply the result buffer and to fill it; as with `rdwr` or `readdir`, this allows wardens to fulfill type-specific operations from cached data without extra copying. After the viceroy returns the results to the interceptor, it calls the warden's `tsopdisp` method, to dispose of the result buffer supplied by `tsop`. The results are then propagated back through the call chain to the application.

4.8 Summary

The Odyssey prototype was built to answer three questions. First, can the Odyssey API and architecture support application-aware adaptation? Second, can the viceroy estimate resources, such as network bandwidth, not directly under the client's control? Third, is the presence of a single point of resource control required to support concurrent applications? The areas of the prototype upon which implementation effort was focused were chosen with these three questions in mind.

The prototype itself makes use of two important NetBSD mechanisms: signal delivery and the virtual file system. Signal delivery is used heavily in the implementation of upcall: an asynchronous notification mechanism that has exactly-once, in-order semantics that can pass arguments and return results. The virtual file system, or VFS, is the mechanism by which the Odyssey name space is integrated into the client's file system; the stackable layers extension is used to implement Odyssey-specific extensions to the VFS operations.

The Odyssey portions of a client's file system are managed in-kernel by the interceptor. This code component forwards file system requests to the viceroy, when then dispatches them to the appropriate warden. Wardens implement all operations on objects of their type; they are free to customize these operations, such as lookup, to their liking.

In addition to dispatching these requests to wardens, the viceroy also encompasses the connection manager. This provides the connection abstraction to wardens, and provides estimation of available bandwidth. This estimation is purely passive, and is the viceroy's best guess as to the bandwidth each connection will receive in the immediate future.

The two API extensions, resource requests and type-specific operations, make use of all of the components of the prototype. Requests depend on upcalls for notification, and resource request lists in the viceroy for efficient checking of previously granted requests.

Chapter 5

Applications

To exercise the Odyssey prototype, a small set of applications were chosen and modified to take advantage of application-aware adaptation. The applications were chosen with a number of goals in mind:

- They should provide experience in porting applications to Odyssey's adaptive model. They therefore must be real applications that one might wish to use in a mobile setting, rather than simple, toy applications.
- They should span a broad a range, using a variety of different data types, supporting a variety of different fidelity levels.
- To explore the needs of binary-only, or *shrink-wrapped* applications, at least one application for which source code is not available should be made to use Odyssey in some way.

The three applications chosen were: XAnim, a video player; Netscape, a web browser for which source was not available; and Janus, a speech recognition system. Each application uses a different data type, and has its own warden and server, giving a broad range of experience. They are all real applications that enjoy significant use, and each presents unique challenges in integration with Odyssey.

These applications, wardens, and servers were all written by other students at Carnegie Mellon; this shed some light on the difficulty of integrating new data types with Odyssey as well as writing adaptive applications. However, the author did provide significant direction in the nature and structure of these components.

5.1 Application Metrics

The central addition to each application was some *adaptive policy*; the strategy that application would use in trading fidelity for performance. In order to express such a strategy, the application must first have metrics for fidelity and performance.

The data accessed by each application has a number of different fidelity levels, each of which is assigned a fidelity value, f. The fidelity metric is such that $\forall f : 0 \leq f \leq 1$.

Fidelities are assigned based on some measure of data quality; larger fidelities imply better quality. A fidelity of one corresponds to reference-copy quality. A fidelity of zero would mean that the data delivered has no useful information; obviously, no data item would actually have assigned to it a value of zero. Though fidelities are quantitative, the only restriction on their assignment is that they have a full relative order. Thus, they may only be compared directly; differences between fidelities are not comparable. Such indirect comparisons would require a fixed, defined scale within the range.

Each application has associated with it a performance metric, chosen with the application's data access needs in mind. This metric, together with fidelity, allows an expression of an adaptive policy. These policies are not meant to produce the best possible adaptive application. Selecting such policies would require spending significant effort in understanding user preference; such an undertaking is beyond the scope of this work. The question is not whether these applications have the best fidelity policies, but rather: Once an application selects some reasonable adaptive policy, how well can that policy be supported by the Odyssey prototype?

Sections 5.2–5.4 describe each of the three applications in turn. Each section describes the application, warden, and server, as well as how these components are integrated into the Odyssey prototype. It then defines the fidelity and performance metrics, and the fidelity policy of the application.

5.2 Video Player: XAnim

The first application to be added to Odyssey was *XAnim*, a video player whose source code is publicly available [57]. In its original form, XAnim reads a movie file from a local disk and plays it back to the screen, skipping late frames to maintain pace through the file; it was approximately 57 thousand lines of code in total. The main data type used by this player in the context of Odyssey is QuickTime, a standard video format defined by Apple Computer [4]; this format has an explicit time base in which the video stream is encoded, and provides facilities for many different representations.

5.2.1 Integration with Odyssey

Dushyanth Narayanan split this single application into a client, warden, and server. Adding these components to Odyssey was straightforward; they are illustrated in Figure 5.1. The server is a relatively simple piece of code consisting of five thousand lines, of which one thousand were added to or modified in the original player. It stores each movie as a number of pre-computed versions called *tracks*; each at a different fidelity. One could as easily use an on-line scheme, that degraded video images on the fly [24]; at present, our QuickTime server does not do so.

The number and sizes of tracks available for each movie are part of that movie's meta-data. The meta-data also specifies, for each track, the sizes and offsets of each frame in that track. The warden, which handles QuickTime data at the system level, can obtain this

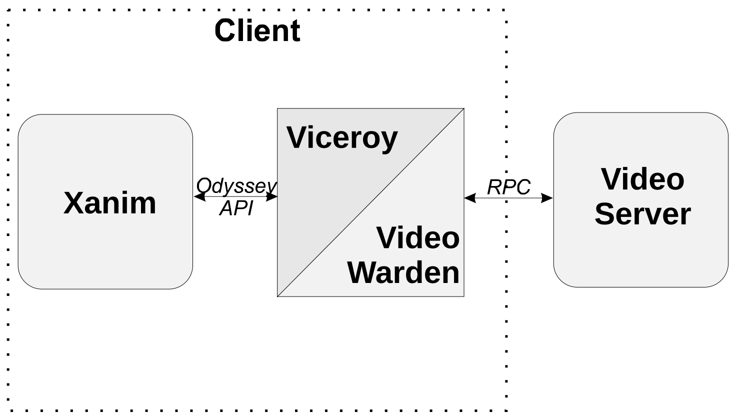


Figure 5.1: Integration of XAnim with Odyssey

`qt_open`	Open a QuickTime movie, return a vector of track descriptions, one per track.
`qt_getframe`	Get frame number N from track T
`qt_getframe_at`	Get the first frame to be displayed at or after time t from track T.

Figure 5.2: QuickTime Access Methods

meta-data from the server, and fetch a range of bytes from a particular track. The warden is responsible for mapping a track's frames to byte ranges within that track; the server provides no such mapping. The warden also prefetches data from the server, anticipating that the most common client behavior is sequential access within a single track. Like the server, the warden is a relatively small piece of code at 2.5 thousand lines.

The client obtains movie data entirely through type-specific access methods, rather than the more cumbersome `read` interface. This enables a simplification of the client, removing 7.5 thousand lines. These operations are summarized in Figure 5.2, and described below.

On `qt_open` the warden fetches the movie meta-data and builds two frame maps. The first translates time offsets to frame numbers for each track of the movie. The second translates frame numbers to byte ranges. Once these are constructed, the warden returns the track summary to the application that called `qt_open`.

On `qt_getframe`, the warden translates the frame number to a byte range. If the requested frame is in the warden's prefetch buffer, it is returned. If it is not present, the warden returns the frame from a better quality track if it is present. If either of these are not present, the warden fetches it from the server on demand. Less than six hundred lines of code needed to be added to the client to use this new interface.

A background thread belonging to the warden prefetches frames from the last track requested by `qt_getframe`; this prefetching is controlled by two parameters, `highwater` and `lowwater`. Whenever there are fewer than `lowwater` bytes in the prefetch buffer, the warden prefetches frames until the prefetch buffer contains at least `highwater` bytes. If higher-quality frames are present when an application requests a lower-fidelity track, the better frames are retained. Conversely, lower-quality frames are discarded and the requested higher-quality ones are fetched on demand.

5.2.2 Metrics and Policy

In the experiments reported in Chapter 7, each movie has three tracks. These three versions are: JPEG-compressed [83] color at quality 99, JPEG-compressed color at quality 50, and black-and-white. Each track is encoded at ten frames per second. The overhead for this extra storage is modest: typically 60% more space than storing only the highest-fidelity track.

Individual frame fidelities are assigned as 1.0, 0.50, and 0.01 to JPEG(99), JPEG(50), and black-and-white frames, respectively. Over a single execution of the player, the achieved fidelity metric is the average of the fidelities of the displayed frames; higher average fidelity means that displayed frames were, on average, of a better quality. Thus a movie with half of its frames displayed from each of the two best tracks would have a fidelity of 0.75.

The client's performance metric is the number of frames it is forced to skip due either to frames arriving late, or the player having been delayed in decoding. If a frame arrives after its deadline, it will be dropped rather than shown. If a frame is more than one frame-time late, then the client will skip past frames that should have been shown while the late frame was being obtained.

Of course, user perception of "quality" of playback includes many other factors. For instance, dropped frames certainly influence the perception of quality. Some number of frames dropped consecutively will be perceived as worse than the same number dropped intermittently. However, precisely capturing such notions is beyond the scope of this work. Rather, these metrics are meant to capture two factors which the video player balances in its adaptive decisions.

The client's fidelity policy is to play the best quality track possible without dropping any frames. When the client opens a movie with `qt_open`, it calculates the bandwidth required to play each track in the movie. From these calculations, the client derives a set of bandwidth ranges appropriate to each fidelity. These ranges are defined with some overlap, and select for fidelities aggressively. The lower bound in a track's range is set to 95% of the bandwidth nominally required to support it; the upper bound is the minimum nominally required for the next higher track.

After opening a movie, the client places a resource request on the bandwidth for the highest quality track, and begins playing frames from that track. Whenever it is notified that the bandwidth has strayed outside of the bounds for the current track, it changes the track from which it is requesting frames, and places a resource request appropriate to the new track's bandwidth requirements.

QuickTime data, using these three encodings, lends itself particularly well to an adaptive policy switching between tracks. This is because each frame can be rendered in isolation, without need for some reference frame. However, one could easily extend this notion to a format with inter-frame compression, such as MPEG [35], by restriction track changes to points in the track with stand-alone frames.

5.3 Web Browser: Netscape

The second application to be modified to make use of application-aware adaptation was *Netscape Navigator*, or more simply Netscape. Netscape allows the retrieval and display of HTML [62], a hypertext mark-up language. At the time of its adoption, the source for Netscape was not publicly available. It was chosen expressly as an example of an application for which source is not available; it provides a simple example of how such applications might take advantage of Odyssey's API extensions.

5.3.1 Integration with Odyssey

To cope with the lack of source code, Odyssey makes use of Netscape's *proxy* facility. Through it, Netscape can route all of its HTTP [22] requests for data through a designated process. This process is commonly on a remote host; such a remote process might act as a gateway that is exempt from firewall restrictions, or a caching proxy for a group of machines [43]. Instead, Odyssey places the proxy, called the *cellophane*, between Netscape and Odyssey, redirecting Netscape's requests through the file system to Odyssey. The cellophane is quite small, at three thousand lines of code. It is this re-routing that ensures that Netscape's network traffic will be visible to the viceroy.

Any application with a proxy mechanism could make use of Odyssey in this way. However, those without some sort of explicitly mechanism would require a more sophisticated approach. For example, one could intercept file and network I/O system calls to redirect them to the viceroy. Binary rewriting could also be used with more difficulty.

The Web is integrated into the Odyssey name space as a single tome. The root of this tome is an onode marked as a context onode, and lookup operations on that onode will attempt to resolve the name component as a URL in the Web. Since URLs use the `'/'` character, which is also used as the UNIX component separator, the cellophane must convert all instances of `'/'` appearing in a URL to backslashes.

The HTML warden handles all of the cellophane's requests. It is less than five thousand lines of code. It forwards all such requests to a remote *distillation server*, which is presumed to be well-connected to the rest of the Web.

The distillation server, at five thousand lines of code, fetches HTML pages and associated images in response to requests, and forwards them to the HTML warden. It is capable of degrading images on the fly using JPEG compression to shorten their transmission time from the distillation server to the client. These components — cellophane, warden, and distillation server — were implemented by Eric Tilton, and are depicted in Figure 5.3.

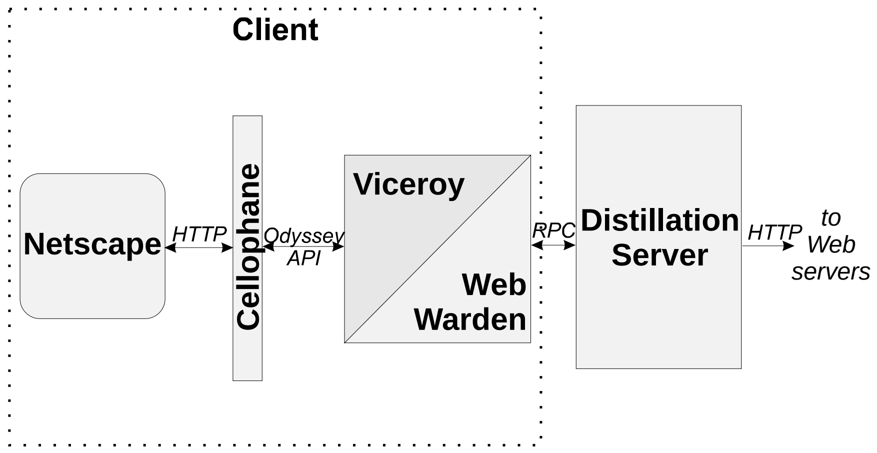


Figure 5.3: Integration of Netscape with Odyssey

`webw_setqual`	Set desired quality for images
`webw_getqual`	Obtain desired quality for images
`webw_sethdr`	Pass along request headers for the next requests
`webw_stat`	Obtain the headers for a particular page

Figure 5.4: HTML Type-Specific Operations

Odyssey focuses on images for two reasons. First, they make up a majority of all bytes served on the Web [48]. Second, there exists a natural degradation method — JPEG compression — that gives good size reductions while yielding tolerable quality. No such obvious degradation exists for text in HTML pages.[1]

The Web warden exports four type-specific operations; two have to do with the fidelity of images, and two are used to help integrate the Web into the client's file system. They are summarized in Figure 5.4.

The `webw_setqual` operation is used by the cellophane to set the desired level of compression for images fetched. It is set based on current bandwidth as described in Section 5.3.2. The `webw_getqual` operation is used by the cellophane to query the current quality setting.

The other two operations handle meta-data for HTTP requests and responses. A request for a Web object carries with it *request headers*, which can affect the content of the fetched objects. To pass these request headers to the Web warden, an application uses the `webw_sethdr` operation. Each page can have associated with it a separate set of meta-

[1]Presently, the distillation server distills only GIF images, not JPEG images. However, as reported in [48], more than half of all image bytes served are from GIF images. Furthermore, JPEG to JPEG distillation has been fruitfully pursued by Fox [24]; extending Odyssey's distillation engine should therefore be straightforward.

data, called simply *headers*. An application obtains these by the `webw_stat` call.

5.3.2 Metrics and Policy

The distillation server, in addition to passing images unchanged, has three distinct levels of degradation available to it, for a total of four levels of fidelity. These levels of degradation consist of JPEG compression at quality levels 50, 25, or 5; lower numbers produce smaller but lower-quality images. These degraded qualities are assigned fidelity levels of 0.5, 0.25, and 0.05, respectively; the original image is assigned a fidelity level of 1.0. The distillation server degrades only those images for which it is expected to provide a benefit — images 2 KBytes or larger. For smaller images, the effort to distill them takes longer than simply forwarding them on all but the very slowest of networks. The f

The performance metric for Netscape is the time to load and display a particular HTML object; in the case of our benchmarks, all objects are single images. Netscape's fidelity policy is to load the best quality image possible within twice the expected time to load the reference quality image at Ethernet speeds. This heuristic is based on the following intuition: there is little utility in loading an image faster, since users typically are willing to wait roughly as long as an Ethernet might take. However, much longer waits, albeit for better quality images, are not likely to be tolerated. As with XAnim's policy, this policy may not reflect the actual desires of users; rather, it was chosen to provide a reasonable policy to support in the Odyssey prototype. To correctly calibrate to user's desires would require human factors experiments.

For each of the four fidelity levels available, the cellophane selects a bandwidth range appropriate to that fidelity level. These ranges are currently hard-coded in the cellophane, and were based on a small set of experiments measuring times to perform JPEG compression and resulting reduction in size. As the bandwidth between the warden and distillation server changes, the cellophane adjusts the distillation level of images served.

5.4 Speech Recognizer: Janus

The final application modified to take advantage of Odyssey is *Janus* [82], a speech recognition system. Janus takes as input a raw, sampled speech utterance collected from a microphone, and returns the words it recognizes in the utterance. This process is very expensive in both CPU cycles and virtual memory. Since the mobile host is relatively underpowered compared to a similarly-priced desktop workstation, it would be useful to offload this computation whenever possible.

The recognition process has two interesting phases. The first is *vector quantization* [61], a signal processing step that transforms the raw speech utterance into a much more compact representation. This phase is relatively inexpensive to compute. The second phase consists of the remainder of recognition, and comprises the bulk of the processing required in recognition.

5.4.1 Integration with Odyssey

This application is quite different from the others; speech data is not something that is merely accessed, but rather it is generated at the client and computed upon. When it is profitable to do so, this computation should take place on some remote, static host.

Despite the fact that Janus is not strictly a data access application, it presents both considerable potential as well as challenge for mobile systems. It is especially useful when mobile since it leaves the user's eyes and hands free for other activities [73]. However, the resource requirements for high-accuracy speech recognition are substantial, especially when mobile, since background noise is often high. Adding higher-level semantic processing, such as language translation, leads to even greater demands on computing resources, which are already in short supply on mobile machines as compared to their static counterparts.

To integrate speech within the client's file system, there is a single, distinguished speech tome in they system, with a single object at its root. Writing an utterance to this object begins recognition on that utterance. A subsequent read on that object will return the recognized text when it is available. A simple front end, consisting of just over six hundred lines of code, loops collecting the raw speech utterance, writing it to the speech object, and reading the result.

This utterance is forwarded to the speech warden, which is approximately two thousand lines of code. The warden can forward the utterance on to one of two different recognition servers, each a full copy of Janus with one thousand additional lines to handle communication. These components are illustrated in Figure 5.5.

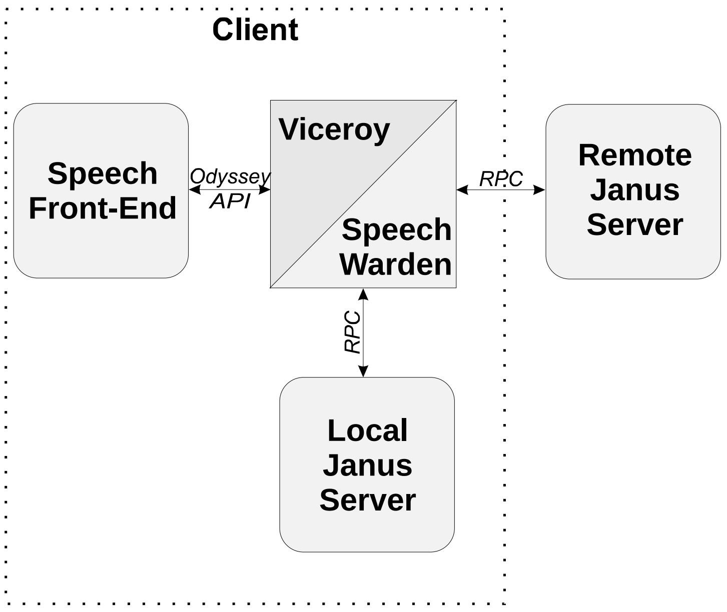


Figure 5.5: Integration of Janus with Odyssey

The warden, when passed a speech utterance, can do one of three things. First, it can pass the raw, large utterance to the remote Janus server for full recognition; this is called *remote* recognition. Second, it can pass the utterance to the local Janus server for *local* recognition. Third, it can pass the utterance to the local server for just the vector quantization step, and pass the much smaller result to the remote server for the second phase of recognition; this is called *hybrid* recognition. The warden provides a single type-specific operation, `speech_setstrat`; the front end can use this to select one of the three strategies.

5.4.2 Metrics and Policy

Currently, there is no notion of fidelity for Janus. Both the local and remote servers use the same acoustical model, vocabulary, and grammar in performing recognition. Thus, there is no fidelity metric.

The performance metric is latency: the time it takes to recognize a speech utterance. The front end has a hard-coded model for the relative computational power of the local and remote hosts, and a model for the costs of local, remote, and hybrid recognition. Together with the current bandwidth, the front end decides whether local or hybrid recognition will result in the fastest response. If network bandwidth is sufficiently high, remote recognition is best; otherwise, hybrid recognition is best. Because of the severe computational demands of the second phase of recognition, the front end does not use local recognition unless the connection to the remote host is down.

While Janus has not incorporated fidelity levels in the course of this dissertation, Jason Flinn, the student responsible for integrating Janus, has been exploring altering vocabulary sizes. Smaller vocabularies result in less computationally intensive recognitions, but at the cost of user expressiveness. To cope with this changing expressiveness, this new speech system provides feedback to the user to keep him informed when increases or decreases in vocabulary occur.

5.5 Summary

Three applications were modified to exercise the Odyssey prototype. The main goals in doing so were threefold. First, this process should shed light on the difficulty in porting applications to take advantage of application-aware adaptation. Second, the applications should provide experience in designing fidelity levels and adaptive strategies. Third, some light should be shed on the problem of using shrink-wrapped programs in the context of application-aware adaptation.

For each of the applications at hand, modifying them to take advantage of application-aware adaptation was relatively simple, requiring only a few hundred to a few thousand lines of code. In each case, the logic required for the adaptive decision loop was isolated from the main body of the application. This was possible because, even in degraded form, the application already knew how to deal with whatever data could be produced. To the extent this is possible with new applications, modifying them should be similarly simple; the complexity of the new code should directly correspond to the complexity of the adaptive strategy.

The applications are all used broadly in their original form outside this project, each uses substantially different data types. At present, the fidelity levels provided for each data type are discrete, the mechanism is in place to broaden them. Adding the on-line construction of fidelity levels to the video player and increasing the number of fidelity levels for Web images would both be straightforward. There has been substantial progress in adding levels of fidelity to Janus.

Finally, the task of integrating Netscape into Odyssey's framework is an important special case of the general problem of working with shrink-wrapped programs. Any such application that provided a proxy mechanism could make use of a technique similar to the cellophane. More generally, techniques such as binary rewriting and system call interception hold some promise in integrating these applications.

Chapter 6

Evaluation Methodology

Recall that there are two key issues in evaluating and comparing adaptive systems. First, how long does it take each system to make an adaptation decision when one is required? Second, when that adaptation decision is finally made, is it the right one? To answer these questions in the context of Odyssey, the client must be subjected to changes in available bandwidth, and its reactions observed and analyzed. This chapter describes the experimental methodology used to subject an Odyssey client to bandwidth variation.

The chapter begins in Section 6.1 describes two key features of wireless networks that render them unsuitable for comparative experimentation: their behavior is complex, and not repeatable. To cope with these challenges, the evaluation instead relies on the technique of *transient response analysis*, more commonly used in *control systems*, [3]; this technique is introduced in Section 6.2.

The chapter then describes a technique called *trace modulation*. In trace modulation, a small layer capable of delaying or dropping all packets to or from a host is inserted in that host's networking stack. These delays are calculated according to a simple network model. The modulation process, as well as the method by which the parameters to the model are changed over time, is described in Section 6.3.

In addition to being used for transient response analysis, this system can also be used to repeatably produce performance in an isolated, wired network that is faithful to that observed in some live, wireless network. This technique of *empirical modulation* — the original motivation for the trace modulation work — is described in Section 6.4. Section 6.5 demonstrates that empirical traces do in fact re-create the performance of the original network for a range of benchmarks.

6.1 Challenges in Wireless Experimentation

There are numerous difficulties in using live wireless networks in the evaluation of mobile systems. Unlike that of wired networks, the medium over which wireless messages travel is difficult to isolate. When mobile nodes that are not part of the evaluation are nearby, they may perturb the results by injecting packets into the wireless spectrum. It is very difficult to physically prevent such packets from interfering with those of the system under test.

Instead of physical control, one could imagine that, if all devices operating in the relevant frequency ranges cooperated, these devices could be logically restrained from interfering. However, many different kinds of wireless devices make use of the same unlicensed frequency ranges. For example, the WaveLAN wireless devices [6] operate in the 900 MHz range; many cordless phones use this range as well. These phones have no concept of media access protocol, and cannot be party to cooperative scheme to avoid interference.

Even if the relevant region's wireless spectrum could be isolated from interference from other devices, obtaining predictable, reproducible results remains a challenge. Wireless propagation is affected by environmental factors that are both spatially [51] and temporally [79] dependent; small changes in the path taken through an area of wireless coverage can have large impacts on performance. For example, multipath effects are extremely sensitive to small changes in position. Likewise, as different obstructions pass through the path between base station and mobile client, signal propagation can change dramatically.

These difficulties present two challenges to experiments carried out over wireless networks. First, the highly variable performance of wireless networks is difficult to describe and understand [21]. Analyzing the behavior of an adaptive system on top of such chaotic performance would be a daunting task. Good evaluation of Odyssey requires a testbed that provides performance that is more easily analyzed.

Second, the inability to isolate wireless media combined with the unpredictable, varying performance makes results extremely difficult to reproduce. Reproducibility of the networking environment is important for three reasons. First, it is essential for sound performance evaluation of a given system. Second, it is necessary for comparative performance evaluation of alternative mobile system designs. Third, it is valuable in debugging mobile systems because it enables re-creation of conditions that trigger bugs.

As understanding of mobile networks improves, the complexity problem may be ameliorated. Unfortunately, such improvement has not yet come to pass; the evaluation of Odyssey must rely on other means. In contrast, the lack of reproducibility is inherent in the very nature of wireless networks. The lack of experimental control and high variance combine to ensure the difficulty of carrying out live wireless experiments.

6.2 Control Systems

At some level, one can view an adaptive client in terms of *control systems theory*. In such a treatment, the system reacts to some set of *inputs*, and produces some set of *outputs*. The production of these outputs is influenced by an *adaptive control system*, which bases its decisions for the next set of outputs based on the current inputs and outputs.

In taking this view, the various resources available to the client, and the demands on them, are the inputs to the system. The adaptive decisions made by the applications running on that client are the resultant outputs. The adaptive control system is the composition of the viceroy, applications and wardens in *discovering* when resource availability changes, *deciding* how to adjust fidelity in response, and *effecting* that decision quickly.

One of the basic techniques to analyze and evaluate an adaptive system is to subject it to a set of *reference waveforms* in the input signals, and study its response to those waveforms. Reference waveforms are typically *transients* — idealized, severe changes in the inputs.

Four simple transients are shown in Figure 6.1. These and similar waveforms are simple and easy to analyze, but produce changes in bandwidth that are idealizations of that observable in the real world. For example, a step function might represent a client that switches from one network technology to another, while an impulse might represent a client that switches back and forth between two different networks. They are all characterized similarly: a sharp change in bandwidth followed by a long, steady-state period; they also restrict themselves to two discrete bandwidth levels throughout the waveform.

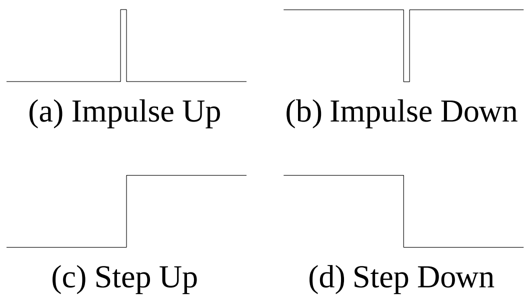


Figure 6.1: Simple Reference Waveforms

These simple waveforms are especially useful in evaluating bandwidth estimation. The client is started with a synthetic application, consuming data as fast as possible. This client is then subjected to the waveforms in Figure 6.1, and the viceroy's estimations from the transient forward are compared to the nominal value. Ideally, the estimation should converge on the nominal value soon after the transient. The figures of merit in this evaluation include *settling time* and *overshoot*. Settling time is the time required after the transient for the estimated value to reach and remain at the nominal value. Overshoot describes the maximum amount by which the estimated value departs from the nominal value after it is reached for the first post-transient time.

These simple waveforms are also used in evaluating applications in isolation. Each application is executed, subject to the waveforms of Figure 6.1. The applications' fidelity and performance can then be examined and compared to non-adaptive strategies.

When the applications are run concurrently, a longer trace with more transitions is used. This longer trace is shown in Figure 6.2. It is fifteen minutes in duration, and alternates between the same two bandwidths used in the shorter waveforms. It is meant to approximate what someone might see walking with a mobile device through a city with two different networks — one fast but short-range, the other long range but slow — that together implement an overlay network. As the visitor walks through the city, his device switches between the two technologies every few minutes. Again, the applications' fidelity and performance can be analyzed in the context of this simple but stressful waveform.

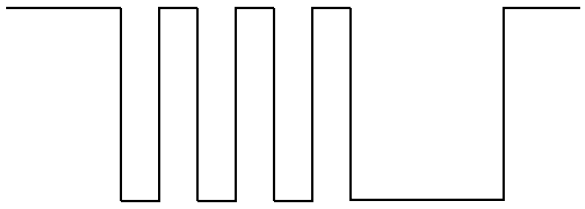


Figure 6.2: Long Reference Waveform

The treatment of Odyssey and its applications as a control system is necessarily informal. Breaking the components that comprise an Odyssey client into easily analyzed units that behave in accordance with simply expressed rules would be challenging at best. Further, Odyssey as a system is certainly not linear; while impulse responses shed light on the behavior of Odyssey, they do not completely characterize the system. That said, the impulse response technique is an extremely valuable tool in evaluating adaptive systems such as Odyssey.

6.3 Trace Modulation

In addition to the inability to provide for precise experimental control, no wireless network is capable of reproducing the waveforms in Figures 6.1 and 6.2. Instead, they are realized through a technique called *trace modulation*. This technique provides application-transparent emulation of some target network on a LAN by modifying NetBSD's network layer; all other components of the host above the network layer of the protocol stack are exactly as they would be in a live system. The modified layer drops or delays packets in accordance with a concise, time-varying list, or *trace*, of performance parameters for a simple network model. Unlike most trace-based systems, modulation influences the environment in which a system operates rather than generating the workload for that system. In other words, trace modulation creates a synthetic environment in which to execute a real workload, rather than create a synthetic workload in a real environment.

Section 6.3.1 describes the simple, linear, network model that underpins trace modulation. The process of trace modulation, the actual delaying or dropping of packets, is described in Section 6.3.2. Finally, Section 6.3.3 describes how traces of model parameters are passed to and used by the replay layer. This, combined with simple tools to generate synthetic traces, can be used to create the waveforms in Figures 6.1 and 6.2.

6.3.1 Network Model

The modulation layer requires some simple model in order calculate the delay of and probability of dropping packets in a test host. The particular model selected drives everything from the implementation of the layer itself to the technique of observing and reproducing the performance of live wireless networks. Simplicity rather than sophistication is the keystone of the network model. It must be able to capture wide changes in network performance within a few hundred milliseconds. It must be possible to derive parameters to the model based on relatively simple observations from a single host; because fine-grained, well-synchronized clocks are not yet available for mobile machines, collecting observations from two hosts is not tenable.

Much of the complexity of wireless networks arises from temporal variation caused by changes in location, physical environment, or cross traffic. Trace modulation copes with this complexity while preserving simplicity by decomposing time-varying behavior into a sequence of short intervals of invariant behavior, much as a complex curve can be approximated by many short line segments. The model itself is most easily described by first treating the case of single packets, and then turning to multiple, queued packets.

Single Packet

Consider two hosts, H_1 and H_2, with H_1 sending packets to H_2 over some network N. This network may also be carrying cross traffic, defined as any traffic through N that is not between H_1 and H_2. Such cross traffic may change over time, and will affect the delays and losses experienced by any traffic between H_1 and H_2.

Consider the end-to-end path from H_1 to H_2 to be a series of m service queues, $q_1, q_2, \ldots q_m$. Each queue's instantaneous service time is modeled deterministically, as

$$t_k = f_k + sv_k \tag{6.1}$$

where t_k is the total delay imposed by queue q_k, f_k is a *fixed*, per-packet cost, s is the size of the packet in bytes, and v_k is a *variable*, per-byte cost. In physical terms, v_k is the inverse of the instantaneous bandwidth of the network element k; f_k is the current transmission latency of that element, and is the sum of queueing, per-packet processing, and propagation delays. For a single packet traversing this network, the total delay experienced from H_1 to H_2 is:

$$\begin{aligned} \Delta &= \sum f_k + s \sum v_k && (6.2) \\ &= F + sV && (6.3) \end{aligned}$$

It is important to note that each queue k services cross traffic as well as direct traffic between H_1 and H_2. This means that the delays and losses experienced by traffic between H_1 and H_2, and hence the values of f_k and v_k, change over time as cross-traffic load changes. The quantities f_k and v_k are thus intended to capture delays as experienced by traffic from host H_1 to H_2, rather than some static properties of the queue.

Multiple Packets

The approach to modeling queueing delay from H_1 to H_2 through N is also simple. As above, an individual queue's service time is broken up into f_k, the transmission latency over that portion of the network, and v_k, the per-byte cost induced by bandwidth constraints. The model holds that the maximum throughput at any individual queue is dependent only on the v_k term and the sizes of packets. Single single-byte latency, f_k, is overlapped and causes no queuing delay.

Since the connection between hosts H_1 and H_2 is a serial string of such queues, the overall throughput is determined by the largest per-byte cost, $\max(v_k)$ at some particular q_k. We call this *bottleneck* queue q_b, its per-byte costs, V_b, and the residual per-byte costs, V_r. By definition, $V = V_b + V_r$, allowing Equation 6.3 to be rewritten as

$$\Delta = F + s(V_b + V_r) \tag{6.4}$$

For a given time segment of duration d, there are a single set of *delay parameters*: $F, V_b,$ and V_r; these are combined into a *delay tuple* of the form $\langle d, F, V_b, V_r \rangle$. By composing a set of these delay tuples into a list, $\mathcal{D}$, one can model arbitrary changes in delay as perceived by traffic flowing from H_1 to H_2 over N.

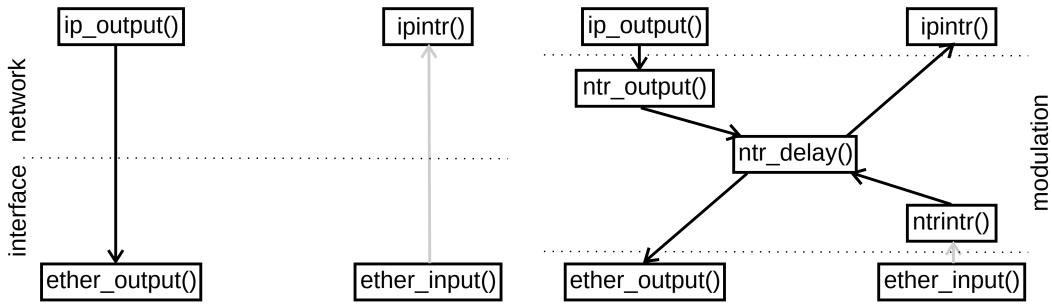
However, delay is only one aspect of network performance; the other is packet loss. Losses are modeled as a probability L of dropping a given packet during the interval d; each packet in d has the same chance of being dropped. The model, which is above the datalink layer, assumes that corrupt packets are coerced to lost ones. Instantaneous loss and delay behavior is combined into a sequence of five-element *network quality tuples*, $\mathcal{S}$, of the form $\langle d, F, V_b, V_r, L \rangle$. Such a sequence, the replay trace, describes network quality over time.

6.3.2 Modulation Layer

The modulation layer, placed between the network and interface layers of the protocol stack, delays or drops packets according to a particular network quality tuple, $s \in \mathcal{S}$; these tuples can be created synthetically or by empirical observation of some live wireless network. The modulation layer acts on both inbound packets — those headed up the protocol stack — as well as outbound packets headed down the stack. Both inbound and outbound packets are passed through the same *delay queue*. The delay queue ensures that packets flow through it at a rate no faster than allowed by V_b, and that each packet is further delayed by V_r and F. The delay queue also drops packets according to the loss parameter, L.

The modulation layer is placed between the IP and Ethernet layers of the protocol stack, as shown in Figure 6.3. This modified protocol stack is used by the host on which we wish to run experiments. The two routines `ntr_output` and `ntrintr` enqueue packets at the delay queue. The `ntr_delay` routine schedules packets in this queue, one at a time, and forwards them or drops them after they have been delayed; it uses an on-line scheduling algorithm to calculate delays and drops according to the parameters found in a replay trace.

Since modulation uses a real, physical network to transfer packets, there are limits to the class of networks that can be modulated. Stated simply, the modulated network must be sufficiently slower than the real network. Exactly how much slower the modulated network must be is unclear. However trace modulation has successfully recreated a 2 Mb/s wireless network over an isolated, 10 Mb/s Ethernet; therefore, this restriction is not particularly onerous.



(a) Standard Protocol Stack (b) Stack with Modulation Layer

This figure shows the insertion of the modulation layer in the NetBSD protocol stack. Figure 6.3(a) shows the original network and interface layers, while Figure 6.3(b) depicts the modified stack. The grey arrows signify software interrupts, while the black arrows are function calls.

Figure 6.3: Modulation Layer

Scheduling Packets

When each packet arrives at the delay queue, it is stamped with its enqueue time. Packets are forwarded in enqueue order;[1] therefore, the delay queue need only consider the packet in the queue's head at any given time. There are three times of interest in scheduling a packet: t_e, the time the packet was enqueued; t_h, the time at which the packet arrived at the head of the delay queue; and t_f, the time the delay queue will next become *free* after sending the preceding packet.

This last quantity, t_f, is maintained as a property of the delay queue. Intuitively, it is intended to capture the cascading effect of delays induced by bandwidth, or V_b, the per-byte bottleneck costs. Suppose that at time t_1, the delay queue begins processing packet p_1. This packet is s_1 bytes long, and takes a delay of $s_1 V_b$ seconds to pass through the emulated bottleneck. According to the model, packet p_2 cannot begin to pass through the emulated bottleneck before $t_1 + s_1 V_b$. This last quantity will be t_f when packet p_2 is at the head of the queue.

The algorithm for delaying and dropping packets is shown in Figure 6.4. It first obtains the next packet to schedule from the queue's head; it was enqueued at `arrived`. If the packet was enqueued before the emulated interface will be `free`, the delay imposed by the interface cannot begin until the `start` time, otherwise it can begin on arrival. The loop then calculates when the current packet will next leave the interface free, and the `send` time, at which the current packet should be sent.

In addition to calculating a packet's delay, the loop generates a random number to decide whether or not to drop it. The random number generator is a copy of that used by

[1]This does not allow the modeling of re-ordered packets, but this has not proven to be a problem in practice.

```
while (1)
    packet = dequeue(queue) /* Block until not empty */
    if (packet.arrived > free)
        start = packet.arrived
    else
        start = free
    endif
    free = start + V_b * packet.size
    send = free + V_r * packet.size + F
    if (random() < L)
        continue          /* Drop packet */
    else
        sleep(until(send))
        forward(packet)
    endif
endwhile
```

This figure shows pseudo-code for scheduling and dropping packets in `ntr_delay`. The quantities V_b, V_r, F, and L are all from a network quality tuple, $s \in \mathcal{S}$.

Figure 6.4: Delay Calculation

NetBSD, and is defined by the following recurrence relation [56]:

$$x_{n+1} = (7^5 x_n) \mod (2^{31} - 1) \tag{6.5}$$

All packets contribute to the calculation of `free`, whether or not they are dropped. If the packet is to be dropped, the loop schedules the next available packet, else it sleeps until `send`, and then forwards the packet.

Scheduling Granularity

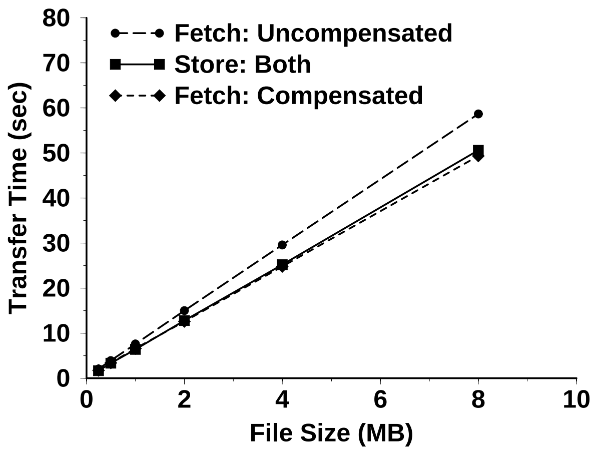
In our current implementation, the delay queue makes use of the clock-based interrupt in the NetBSD kernel. The resolution of that interrupt is only 10 milliseconds. To cope with this limited resolution, modulation makes the simplifying assumption that packet arrivals, and hence departures, are uniformly distributed between clock ticks. Hence, if each packet is scheduled on the closest clock tick, the long term average error should tend to zero.

Because packets to be delayed less than half a clock tick are sent immediately, sparse traffic modeled over relatively high-performance links will not be sufficiently delayed. This simplifying assumption could be avoided in one of two ways. One approach would be to use a custom hardware clock, but this would preclude the ability to run modulation on stock machines. The other approach, which was rejected in the interests of minimal system perturbation, would be to raise the frequency of clock interrupts as described by Ahn et al [2].

Delay Compensation

Modulation attempts to provide symmetric delay of inbound and outbound traffic; that is, for a fixed set of modulation parameters, inbound traffic should perform exactly the same as outbound traffic. However, because the unified delay queue is placed at the endpoint of the path, inbound and outbound delays are slightly asymmetric, as shown in Figure 6.5. In this figure, a synthetic trace roughly equivalent to that of a WaveLAN is used to modulate FTP transfers of varying sizes, both inbound and outbound. The uppermost, dashed curve in this figure shows the fetch FTP performance with this implementation; the solid line shows the store FTP performance. Without modifications, inbound traffic has significantly lower throughput than outbound, an artifact of the asymmetric placement of the delay queue. An accurate realization of the network model would delay these two streams identically.

To correct for this, modulation compensates for the additional delays on inbound traffic. To determine the amount of compensation, one must measure the physical network over which modulation will take place, using the tools described in Section 6.4. This measurement need occur only once; it is independent of the network to be emulated. The experimenter then must take the long-term average of the modulating network's bottleneck per-byte costs, V_b. This term is subtracted from the replay trace's bottleneck per-byte costs for modulation of inbound packets.



This figure depicts replay of a synthetic trace whose performance is close to that of a WaveLAN device, both with and without inbound traffic compensation. A perfect realization of the network model would result in identical performance for Fetch and Store.

Figure 6.5: Effect of Delay Compensation

The effectiveness of compensation is shown in Figure 6.5. Store throughput does not change. However, fetch throughput with compensation, shown by the dotted curve, is much closer to store, confirming the importance of compensation.

6.3.3 Trace Replay

The modulation layer provides facilities to delay packets according to a single network quality tuple. However, to reproduce the reference waveforms of Figures 6.1 and 6.2, one must use a list of such tuples, $\mathcal{S}$, that varies the behavior of the network over time. Such lists are called *replay traces*, and they are fed to the modulation layer by the *replay tool*.

The replay tool consists of a user-level daemon and an in-kernel, circular buffer. The daemon feeds the tuples from a replay trace to the kernel, which supplies them to the modulation layer, each in turn. When there are no records left to replay, the kernel forwards packets without modulation. The daemon may also abort a replay trace, even if there are unconsumed tuples in the kernel; these tuples are flushed, and packets are forwarded without delay from then on.

The interface used by the daemon is guarded by a pseudo-device that acts as a semaphore. Before using the replay interface, the daemon must open this device. The kernel ensures that only one such device can exist, and that only one process may hold the device open at any give time. When the daemon opens the pseudo-device, any old entries that might be present in the replay buffer are removed, and compensation parameters are set to zero.

`compensate`	Establish compensation parameters. Can be used to compensate either the inbound or outbound side of any network quality parameter. Establishes fixed offsets that are added to or subtracted from each applicable delay or loss calculation.
`seed`	Seed the random number generator used by the modulation layer.
`replay`	Pass a list of network quality tuples to the in-kernel buffer; they will be appended to the buffer as space permits. Will block until there is enough space.
`stop`	Forcibly abort a replay, clearing all pending tuples.

Figure 6.6: Trace Replay Interface

After successfully opening the pseudo-device, the daemon may then make use of the replay interface, summarized in Figure 6.6. The daemon first performs the two initialization steps; it calls `compensate` to apply any compensation parameters specified on invocation, and calls `seed` to set the seed for the random number generator if one was requested.

Once setup is complete, the daemon then reads tuples from a file and passes them to the kernel via `replay`. If the kernel has room for the records in its internal buffer, they are copied immediately. If the passed records would not fit in even an otherwise empty kernel buffer, the kernel returns `E2BIG`. In the intermediate case, the call to `replay` will block until there is room for all of the passed tuples to be copied. The daemon can feed the file to the kernel a single time, or it may loop over the file indefinitely.

If the daemon is interrupted, it calls `stop` to abort the replay that is in progress, and terminates. Alternatively, another program can also call `stop`. This causes the daemon's next call to `replay` to return with an error code, and thence to quit.

6.4 Empirical Generation

While synthetic traces are sufficient for carrying out impulse response studies, they do not produce results that are representative of real wireless networks. Unfortunately, in addition to the complexity of performance, such networks do not provide the repeatable results necessary for evaluation of or comparison between mobile systems. Thus, to improve the utility of trace modulation, it was extended with facilities to repeatably recreate the performance of a real wireless network.

To provide this facility, an experimenter collects *empirical traces* of some path through an existing wireless infrastructure. When replayed through the modulation layer, this trace yields performance faithful to that observed during collection, but is much more reproducible. The generation of empirical traces takes place in two phases. In the first phase, *collection*, the experimenter traverses a real wireless infrastructure with an instrumented laptop. This laptop generates a known workload, and records the performance of that workload. In the second phase, *distillation*, the experimenter transforms the raw collected trace into a replay trace. Sections 6.4.1 and 6.4.2 describe collection and distillation, respectively.

6.4.1 Collection

During trace collection, an experimenter carries a portable computer, typically a laptop, along some path of interest through a wireless infrastructure. During this traversal, the laptop generates a particular, known workload. The performance of that workload is observed and recorded by the *collection tool*, which is a combination of an in-kernel trace collection buffer and a user-level program which extracts this collected trace.

Trace Collection and Extraction

A laptop used for trace collection has an instrumented protocol stack that records information about each incoming and outgoing packet; the recorded information is gathered into a fixed-sized, circular buffer called the *trace buffer*. This in-kernel trace collection is similar to other network data collection platforms [33, 45], in that it provides accurate timing of network events with modest overheads.

Hooks are placed in the input and output routines of traced devices to allow the tracing software access to packets. If tracing is enabled, the packet tracing routine examines the media header and encapsulated packet to ensure that the packet is one of the traced types. It then copies relevant information from the packet into the trace buffer. Periodically, the kernel examines the device performance parameters and places that information into the buffer as well. Since the kernel buffer is limited in size, it may be overrun. In that case, the collection tool is careful to keep track of the number and type of lost records.

Packet, device, and lost record information is collected in a trace format defined elsewhere [52]; the details are beyond the scope of this document. The format is designed for flexibility and extensibility, but is fully self-descriptive.

Collection is controlled by a user-level daemon, which interacts with the kernel buffer through a pseudo-device, for which the kernel supports `open`, `close`, and `read` operations. When the daemon opens the pseudo-device, it enables tracing as a side effect. As with the replay pseudo-device, only one such collection device can exist, and only one process may have that device open at any time. The daemon then reads records from the kernel's buffer; a call to `read` will block until there are records available. When the daemon is finished collecting data, it calls `close` for the pseudo-device, which turns off trace collection and flushes any uncollected records.

Known Workload

In order to obtain observations that can be distilled to a list of model parameters, the collection host must generate and record the performance of some known workload. Because clock drift on laptops can be significant relative to the time scale of network quality variation, the workload is constrained to use a strategy that depends only on timestamps taken from a single host. This implies that the workload must consist of round-trips, and that empirical trace collection must assume that network delays are symmetric. These assumptions could be removed if trace collection hosts were equipped with high-resolution, low-drift, synchronized clocks.

The known workload is a modified version of the `ping` utility, consisting of ICMP ECHO and ECHOREPLY packets. The modified `ping` sends out a group of three packets each second, in two stages. In the first stage, `ping` sends an ECHO request with a small data payload of size s_1 to a target host. When the corresponding ECHOREPLY is received, `ping` begins the second stage by sending two larger ECHO requests of size s_2, back-to-back, to the same target host. If the first response is never received, the second and third ECHO packets are never sent; instead, the cycle begins anew.

The trace format for ECHO packets records the sequence number, the `id` field — the process identifier of the process that generated the ECHO — as well as the time at which the packet was generated. For ECHOREPLY packets, the kernel again collects the `id` field. It also records the round-trip time, obtained by subtracting the time the ECHOREPLY was received from the time stored in the packet's payload. Since all timestamps are provided by a single host, synchronized clocks are not needed.

The trace format supports measurements from a variety of network devices. This allows post-processing tools to correlate device and network performance. However, only the AT&T WaveLAN packet radio device was available to the author at the time of this writing. This device operates in the 900MHz region, and offers a nominal bandwidth of 2 Mb/s. The static infrastructure for our WaveLAN network, Wireless Andrew [10], consists of a collection of base stations called WavePoints that serve as bridges to an Ethernet. A roaming protocol triggers handoffs between WavePoints as a WaveLAN host moves. The WaveLAN device reports signal characteristics such as signal level, signal quality and silence level, which we record in the collected trace along with packet traffic.

6.4.2 Distillation

Once the raw observations of the modified `ping` workload have been collected, they are *distilled* by an off-line algorithm that converts these observations to a replay trace. The central idea is to take each group, called a *triple*, of three ECHO and ECHOREPLY pairs, and from those three observations derive the three delay parameters, F, V_b, and V_r. A sliding window passes over the raw observations and generates network quality tuples by averaging the delay observations from each triple in the window, and counting the percentage of lost packets within the window.

Delay The production of the estimate of instantaneous network delay from one triple requires two steps. The first step is determining the end-to-end latency, F, and the total per-byte costs, V. The second step involves discovering the relative proportions of V_b and V_r.

The first packet takes some time t_1 for its round-trip, the second some longer time t_2. Since their round-trips were entirely non-overlapping, we know the second packet incurred no queueing delay due to the first packet. For these packets, each taking a round trip, the network model says that:

$$t_1 = 2(F + s_1 V) \tag{6.6}$$
$$t_2 = 2(F + s_2 V) \tag{6.7}$$

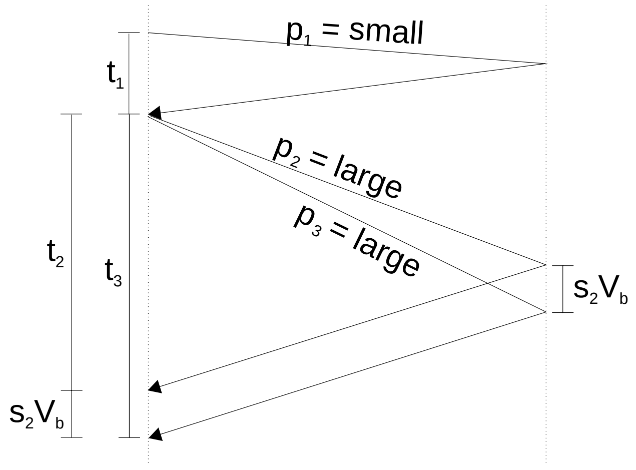
From these equations we can determine F and V.

The second and third packets each have size s_2, but the second takes time t_2, and the third t_3, where $t_3 > t_2$. Since they were sent back-to-back, the third packet is subject to queueing delay behind the second. Hence the model says that:

$$t_2 = 2(F + s_2(V_b + V_r)) \tag{6.8}$$
$$t_3 = 2(F + s_2(V_b + V_r)) \quad + \quad s_2 V_b \tag{6.9}$$

That is, that the third packet is delayed behind the second at the bottleneck queue, and that delay is exactly $s_2 V_b$. Note that the bottleneck cost is paid only on the outbound leg; on the inbound leg, the third packet is already delayed far enough behind the second to get through the bottleneck queue without extra delay. These two equations, together with the first two, yield one set of F, V_b, V_r estimates. The packets of a triple and their associated elapsed times are shown in Figure 6.7

Occasionally, solving Equations 6.6–6.9 for a single group of packets results in a negative value for one or more of the parameters F, V_b, V_r. Such values arise when some packets in the group experienced substantially different networking conditions from the



This figure depicts one triple as sent by the modified `ping` workload. The instrumented host is the dotted line on the left, the target host the dotted line on the right; time moves forward down the figure. There are three ECHO requests and ECHOREPLY responses. Each one of the former immediately elicits one of the latter from the target host. The first and second round trips are completely non-interfering, and thus are independent. The third round trip, however, experiences queuing delay behind the first; in the networking model, this queueing delay is equal to $s_2 V_b$.

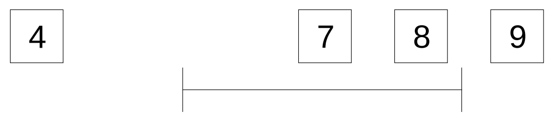
Figure 6.7: One Workload Triple

others. In such situations, the distillation algorithm plugs in the immediately preceding observed parameters, and takes the difference between the expected and observed times. This difference is applied to F, reusing the previous V_b and V_r; the reasoning for this is that short-term performance variation is most likely due to media access delay. The algorithm is careful not to let this corrective factor cascade through successive observations.

A sliding-window algorithm converts these estimates into components of an element of $\mathcal{D}$, a delay tuple. Each step produces an average of the n observations of each parameter F, V_b, and V_r in the current window. The choice of window width, five seconds, balances the desire to discount outlying estimates with the need to be reactive to true change in network conditions.

Loss To estimate the loss rate, L, the algorithm examines sequence numbers of the ECHOREPLY packets in and immediately surrounding the current window. This reveals, for the time from the last packet before the window to the first packet after the window, how many ECHOREPLY packets were expected but failed to arrive.

A very simple example of this is illustrated in Figure 6.8. The last packet that arrived before the window is packet number 4, the first packet that arrived after the window is packet number 9. There are four packets between 4 and 9, only two arrived; therefore two were lost. The algorithm assumes that a run of lost packets would have arrived evenly distributed through time. So, in our example, only one of the two lost packets, number 6, would have arrived within the window. So, for the current window, we received two ECHOREPLY packets, but expected to receive three.



This figure illustrates an example loss calculation. Each box contains an ECHOREPLY with a label equal to the sequence number of the packet. In this example, two packets were lost between packets 4 and 7 — packets 5 and 6 — but only one of them, packet number 6, was expected within the window. Therefore, for this window, three ECHOREPLY packets were expected but only two were seen.

Figure 6.8: Fraction of Packets Lost in a Window

In general, for each window it is known that the instrumented host received b ECHOREPLY packets, but expected to receive a. Let $P = 1 - L$ be the unknown probability that a packet sent arrives without being dropped. Thus, a ECHO packets were sent, of which Pa arrived at the target host. For each of those Pa packets, the target host responds with an ECHOREPLY, of which P^2a arrive back at the original sender. But, since the sender received b ECHOREPLY packets:

$$b = P^2a \tag{6.10}$$
$$L = 1 - \sqrt{b/a} \tag{6.11}$$

Combining L and $\mathcal{D}$, the distillation algorithm obtains a network quality tuple, which is an element of $\mathcal{S}$. The algorithm to produce the entire list runs in the order of the length of the trace, and comprises a single pass.

As an aside, no clocks are used in calculating L, so in fact one could dispense with the symmetry assumption for loss information. However, to simplify the implementation, empirical trace collection preserves the more restrictive assumption of symmetry already needed for calculating delay parameters. While the WaveLAN exhibits slightly asymmetric loss rates, the assumption of symmetry does not significantly affect the accuracy with which empirical traces capture the performance of real wireless networks.

6.5 Validation

The goal of empirical trace modulation is to subject a system to a networking environment indistinguishable from the one on which the trace was collected. To gauge its success, a set of three diverse benchmarks, described in Section 6.5.1 were run over four live *wireless scenarios*, described in Section 6.5.2. Then, empirical traces were collected for each scenario, and the benchmarks were run over a network modulated by those traces. The benchmark performance over live and modulated networks is compared in Section 6.5.3 to see how faithfully the latter reproduced the former.

6.5.1 Benchmarks

The first benchmark involves a World Wide-Web browsing workload [74]. In this benchmark, Web reference traces of five users performing search tasks are replayed as fast as possible on a modified Mosaic v2.6 browser. To ensure good experimental control, all objects referenced in these traces are first copied to a private Web server, and all URLs in the Web traces are then changed to refer to this server. This benchmark generates a moderate amount of traffic, and uses TCP as its underlying transport mechanism. The objects transferred by the benchmark are typically small.

The second benchmark is FTP. This benchmark transfers a single 10MB file disk-to-disk, both to and from a laptop. It makes heavy use of the wireless network, and also uses TCP as its underlying transport protocol. It is designed to highlight any potential asymmetry in network performance, especially important given the assumption of network symmetry forced by the lack of synchronized clocks during trace collection, as discussed in Section 6.4.1.

The third benchmark is the Andrew Benchmark [28] run on NFS [64], a commonly-used network file system. Since NFS was not designed for a mobile environment, it makes no special attempt to defer or eliminate traffic on networks of low quality. The NFS cache is flushed before each trial of the experiment.

The input to the Andrew Benchmark is a tree of about 70 source files occupying about 200KB. There are five distinct phases in the benchmark: MakeDir, Copy, ScanDir, ReadAll, and Make. The benchmark is run over files stored in NFS. Roughly, there are two classes of NFS operations: status checks and data exchanges. The former are typically very small messages, while the latter are larger. For most NFS clients, the ScanDir and ReadAll phases operate on warm caches, and transmit only status-check messages. All NFS messages are sent over UDP.

6.5.2 Mobile Scenarios

The four scenarios we have chosen for evaluation are all from Carnegie Mellon University, and were chosen to cover a wide range of user behavior and network quality. Figures 6.9–6.12 present key network characteristics of these scenarios.

All scenarios use the WaveLAN wireless network exclusively. This is an especially stressful test case for empirical modulation because WaveLAN is a fast medium and packet delays are short. The accuracy of emulation, therefore, is likely to be particularly sensitive to limitations of the model and shortcomings in the implementation.

The top left component of each figure depicts the observed signal level in WaveLAN-specific units. Higher levels indicate stronger signals; levels below 5 are assumed to be background noise by the WaveLAN driver. The other three components of each figure depict quantities derived from the distilled traces: latency in milliseconds, bandwidth in Kbits per second, and loss rate in percent. To account for temporal variation, four trials were obtained of each scenario. Each graph combines the observations from all four trials.

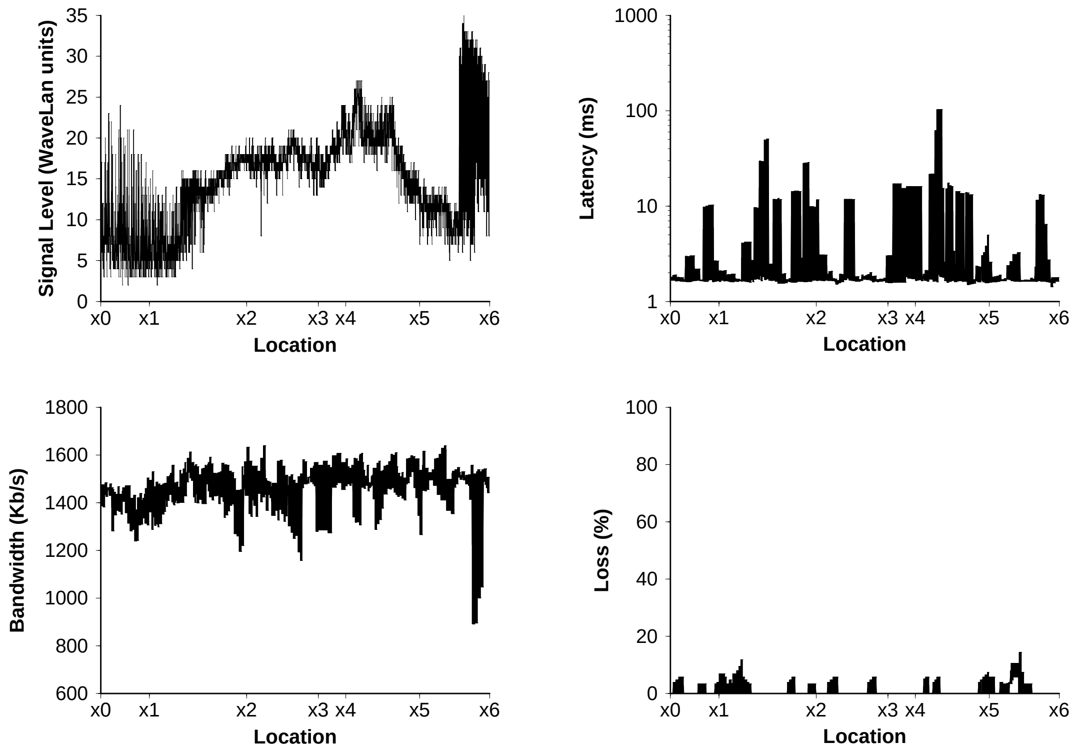
In Figures 6.9 through 6.11, the X axis represents location, with labels indicating checkpoints along the path followed during collection. Although every effort was made to keep physical speed identical across trials of a scenario, perfect consistency is impossible. To account for this, inter-checkpoint intervals are normalized to be of the same length across different trials of a given scenario; these lengths roughly correspond to the proportion of time that interval took with respect to the entire trace. At each X value, the vertical line represents the range of observed parameter values at that location across different trials. For example, at location x4 in the first graph of Figure 6.9, the minimum observed signal level was 17 and the maximum was 22.

Since the fourth scenario does not involve motion, it is meaningless to attempt to correlate parameter values with locations. Hence Figure 6.12 depicts the occurrence of observed values as histograms.

Porter: Inter-Building Travel

The *Porter* trace, depicted in Figure 6.9, begins in the main lobby of Wean Hall (location x0 in the graphs), then traverses an outdoor patio to Porter Hall (x1-x3), and finally enters and traverses Porter Hall (x4-x6).

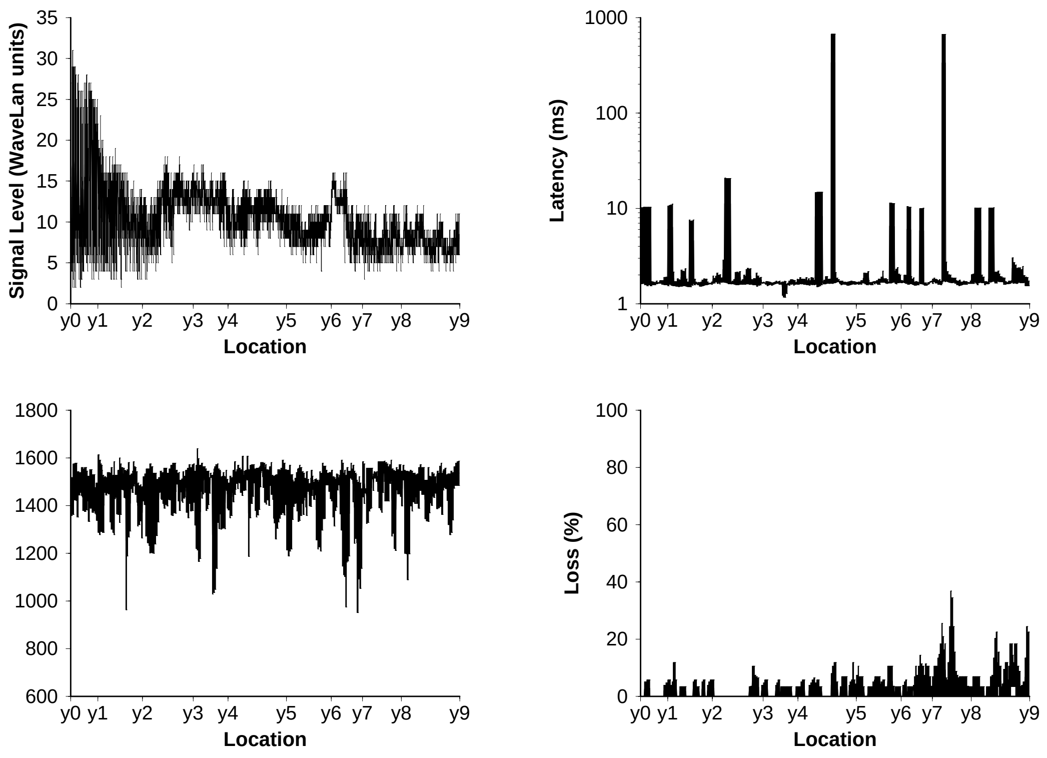
Signal level is highly variable initially, but steadily improves as the Wean-Porter patio is crossed. It falls off again as Porter Hall is traversed. Close to location x5 in Porter Hall, signal level again becomes highly variable. The latency graph indicates several spikes as high as 100 milliseconds, but typically hovers between 1.5 and 10 milliseconds. The bandwidth graph shows typical rates between 1.4Mb/s and 1.6Mb/s, but also indicates spikes as low as 900Kb/s. Loss rates are typically below 10%, the worst cases being the early portion of the Wean-Porter patio, and the end of Porter Hall.



This figure shows observed signal quality and derived model parameters for the Porter scenario. At each X value, the vertical line gives the range of observations at that location across trials. Note the log scale for latency.

Figure 6.9: Porter Scenario

Flagstaff: Outdoor Travel



This figure shows observed signal quality and derived model parameters for the Flagstaff scenario. At each X value, the vertical line gives the range of observations at that location across trials. Note the log scale for latency.

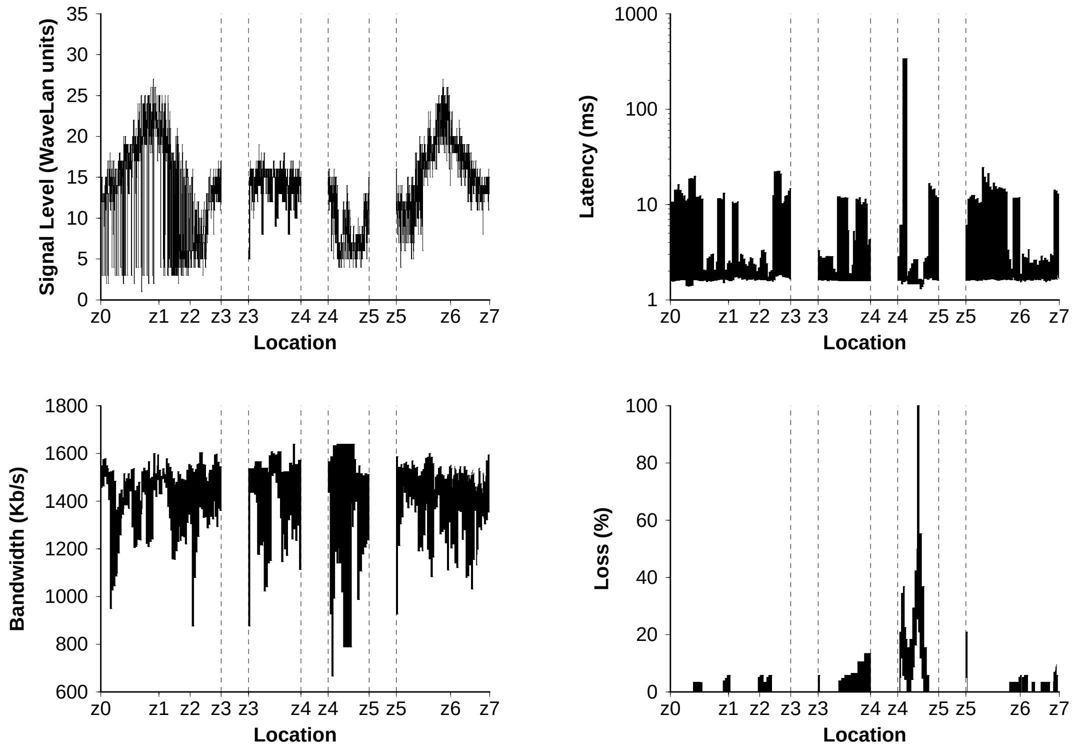
Figure 6.10: Flagstaff Scenario

Flagstaff, the next scenario, is depicted in Figure 6.10. The path for this scenario leaves Porter Hall (y0-y1) to walk along the back edge of the campus in Schenley Park (y1-y5), then around Flagstaff Hill (y5-y9). The entire trace takes place outdoors, but at all times remains in the line of sight of buildings housing WavePoint base stations.

Overall, signal quality during the Flagstaff traces is somewhat below that of the Porter traces. It starts off highly variable, then falls off sharply as soon as Schenley Park is entered, and stays roughly constant at a low level thereafter. On the whole, latency is much better in Flagstaff than in Porter. Average bandwidth is somewhat better in the Flagstaff traces than Porter. Where the Flagstaff traces are significantly worse than the Porter traces is in loss rate, particularly later in the traversal.

Wean: Traveling to a Classroom

The next scenario is traveling from a graduate student office to a classroom, all within Wean Hall. This trace, depicted in Figure 6.11, is called the *Wean* trace. The trace begins in an office with known poor connectivity (z0), then traverses a hallway to the building's elevator (z0-z3). After a wait for the elevator (z3-z4), the traversal enters and rides it three floors (z4-z5). It then exits the elevator and walks to the classroom (z5-z7). Since this scenario involves discontinuous motion, the graphs in this figure are broken into four regions: the walk to the elevator, the wait for the elevator, riding the elevator, and the walk to the classroom.

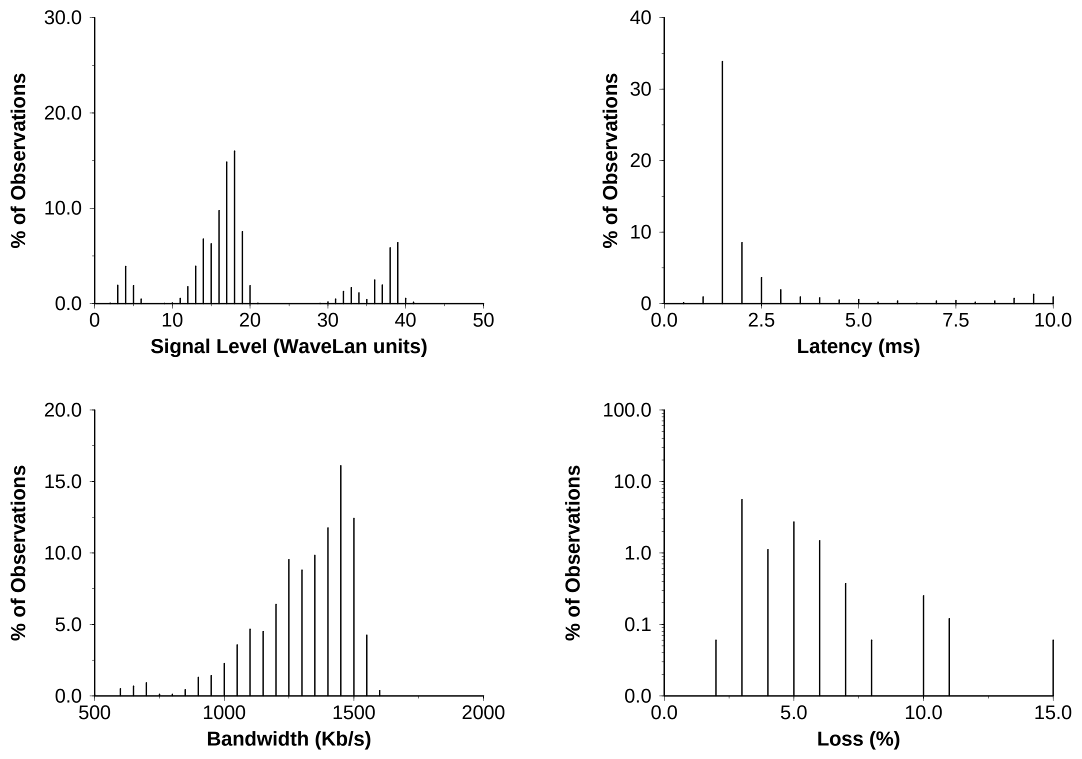


This figure shows observed signal quality and derived model parameters for the Wean scenario. At each X value, the vertical line gives the range of observations at that location across trials. Note the log scale for latency.

Figure 6.11: Wean Scenario

Signal level is variable, but acceptable for the entire walk to the elevator. While waiting, signal level is quite good, but on the elevator ride it drops precipitously. On exiting the elevator, signal level is again good during the walk to the classroom. Latency is good except during the elevator ride, peaking at 350 milliseconds. Bandwidth is somewhat lower than that found in the Porter traces. Loss rates are low except for the duration of the elevator ride, where they are atrocious.

Chatterbox: Busy Conference Room



This figure shows observed signal quality and derived model parameters for the Chatterbox scenario. Unlike previous scenarios, there is no physical movement. Thus, all graphs depict distributions of observed values. Note the log scale for loss rate.

Figure 6.12: Chatterbox Scenario

The final scenario is intended to capture the effect of interfering wireless traffic rather than physical motion. The trace collection host is placed in a room with five other laptops also using WaveLAN. Each of the other laptops continuously executes a workload produced by SynRGen [20], a synthetic file reference generator. The synthetic workload models a user in a edit-debug cycle on files stored on a remote NFS file server.

This scenario, called *Chatterbox*, is depicted in Figure 6.12. This figure differs from the depictions of the previous three scenarios because there is no physical motion. It uses simple histograms rather than a plot of parameter values along a sequence of checkpoint locations. The difference in depiction limits one to coarse comparisons to the previous scenarios.

Figure 6.12 shows that signal level is consistently high, typically around 18. In spite of high signal level, the presence of interfering traffic results in poorer latency and bandwidth relative to previous scenarios. Loss rates are reasonable.

6.5.3 Results

Trace collection and benchmarking were performed on an IBM ThinkPad 701c laptop with an Intel 80486 75MHz DX4 processor and 24MB of memory. The laptop used a WaveLAN radio modem, and relied on an infrastructure of several dozen WavePoint base stations to provide service to the mobile host through a single IP router. The laptop communicated with an Intel Pentium 90MHz workstation with 32MB of memory connected to the campus network via Ethernet. For modulation experiments, these same machines were connected with an isolated Ethernet. Both machines ran NetBSD version 1.2, customized for trace collection and modulation. In the experiments, only the ThinkPad performed collection or modulation.

Each benchmark, on each scenario, was measured over four live trials. Concurrently, four traces of each scenario were collected, interleaving trials with trace collection. These collected traces were then distilled for use in modulation; one trial of the benchmark was run over each of these distilled traces. Modulation is considered to have faithfully reproduced the live scenario if the differences of the means of modulated and live benchmarks are within the sums of their standard deviations.

As one might expect, two trials of the same benchmark over the same distilled trace show little variance. When the same benchmark is run over distinct distilled traces intended to duplicate the same path, however, the results can show significant variance.

World Wide Web Benchmark

Figure 6.13 presents the results from the World Wide Web benchmark. In all scenarios, the difference between the means of real and modulated elapsed times is less than the sum of their standard deviations. This indicates that trace modulation is accurate within the bounds of experimental error for these scenarios.

Scenario	Real (s)		Modulated (s)	
Wean	161.47	(7.82)	160.04	(2.60)
Porter	159.83	(5.07)	150.65	(5.83)
Flagstaff	157.82	(6.58)	148.64	(9.61)
Chatterbox	169.07	(17.63)	157.62	(10.18)
Ethernet	140.30	(3.07)	—	—

This table gives the mean elapsed time in seconds of four trials of the World Wide Web benchmark for each mobile scenario. For reference, the last row gives the performance of the benchmark over the Ethernet used for modulation. Figures in parentheses are standard deviations. Note that due to a problem with our experimental setup, the real Porter numbers come from only three trials rather than four.

Figure 6.13: Elapsed Times for World Wide Web Benchmark

FTP Benchmark

The FTP benchmark is important for two reasons. First, it is network-limited, and therefore most sensitive to network performance. Second, since send and receive performance are largely independent, it allows exploration of the impact of the network symmetry assumption forced by the lack of synchronized clocks during trace collection.

Figure 6.14 presents the results for FTP. In the Wean and Chatterbox scenarios, real and modulated performance are comparable: the difference between the means is less than the sum of their standard deviations. While this is not true of the Flagstaff scenario, the real send and receive performance differ by more than 20 seconds. Both modulated send and receive performance are very close to the mean of real send and receive, and hence an accurate recapturing of the traced environment given the symmetry limitation. The Porter scenario is the only troubling one, not sufficiently delaying either send or receive traffic. Modulated send performance is off by 1.05 times the sum of the standard deviations; receive is off by 1.56 times.

The substantial difference between send and receive performance over the real WaveLAN in these scenarios indicates that network performance is in fact asymmetric.[2] This contradicts the modeling assumption of symmetry stated in Section 6.4.1, and further emphasizes the need for synchronized clocks during trace collection; such clocks would allow us to trace one-way performance. Also of note is the large standard deviation in the Chatterbox scenario. This high variance is shown in almost all of the real and modulated results over that scenario.

Andrew Benchmark on NFS

Figure 6.15 presents the elapsed times for each phase of the Andrew Benchmark under real and modulated network conditions, as well as the total times for the benchmark. In three of the four scenarios — Wean, Porter, and Chatterbox — the difference between the means of real and modulated total times is within the sum of their standard deviations. In two of those three, Porter and Chatterbox, this is also true of all individual phases of the benchmark.

In the Wean trace, the ScanDir and ReadAll phases are both under-delayed in modulation. This likely is due to the inability to schedule modulation delays at granularities shorter than 10 milliseconds. Many of the short, infrequent messages exchanged during those two phases do not have calculated delays large enough to be acted upon.

The Flagstaff scenario, where real and modulated performance diverge the most, most likely also suffers from this phenomenon. As shown in Figures 6.9 through 6.12, Flagstaff latency and bandwidth tend to be comparable to or better than those of the other three

[2]This asymmetry arises from a difference between the client's implementation of the WaveLAN media access protocol from that of the cell server. In the WaveLAN, a node that is to send a pakcet must first listen for another transmitter. If some other transmitter is currently active, the node must back off and try again. The client's back-off counter has fewer bits than the cell's, but the total back-off times of the two kinds of nodes are comparable. Thus, the client backs off in fewer, larger steps, favoring the cell in transmission.

Scenario		Real (s)	Modulated (s)
Wean	send	79.88 (10.88)	72.65 (3.33)
	recv	64.93 (0.93)	67.83 (2.34)
Porter	send	86.38 (4.94)	76.65 (4.29)
	recv	82.23 (1.92)	72.95 (4.01)
Flagstaff	send	88.15 (1.60)	74.88 (2.97)
	recv	61.85 (1.12)	70.80 (3.36)
Chatterbox	send	116.83 (30.49)	92.13 (20.13)
	recv	96.83 (42.15)	87.28 (17.18)
Ethernet	send	20.50 (0.08)	—
	recv	18.83 (0.17)	

This table gives the mean elapsed times in seconds of four trials of the FTP benchmark. Send and receive performance are reported separately. For reference, the final row gives benchmark performance over the Ethernet used for modulation. Numbers in parentheses are standard deviations.

Figure 6.14: Elapsed Times for FTP Benchmark

scenarios. Thus, many of the shorter NFS messages that are present in the benchmark will have delays below the threshold.

Discussion

The three experiments have a broad range of traffic: medium to heave loads, using both short and long messages over both unreliable and reliable transport protocols. While empirical modulation is not perfect in reproducing wireless behavior for these applications, the results are quite good; it is inaccurate in only two of the twelve possible combinations of scenarios and benchmarks. This despite a very simple model capable of expressing only symmetric performance; something that isn't true in the case of the WaveLAN. Overall, empirical modulation provides a solid base upon which to evaluate and compare systems.

6.6 Summary

Wireless networks are too complex to carry out clear analyses of complex systems using them. The inability to easily understand such a system forces one to use techniques other than fully live experiments to evaluate systems such as Odyssey.

The key technique used in evaluating Odyssey, impulse response analysis, is borrowed from the field of control systems theory. In this technique, the system is subjected to simple, sharp changes in network bandwidth, called reference waveforms; its reaction to these waveforms can then be analyzed. The simplicity of the waveforms makes this analysis tractable; since quickly adapting to change is the focus of Odyssey, the sharp variations lead to an attractive test case.

Scenario		MakeDir (s)	Copy (s)	ScanDir (s)	ReadAll (s)	Make (s)	Total (s)
Wean	Real	3.00 (0.00)	18.00 (0.82)	13.50 (0.58)	23.00 (0.82)	105.50 (3.87)	163.00 (4.40)
	Mod.	2.50 (1.00)	19.25 (3.69)	10.00 (1.83)	19.00 (0.82)	112.00 (4.97)	162.75 (4.86)
Porter	Real	3.00 (0.00)	20.00 (1.41)	18.50 (4.65)	23.50 (1.29)	104.50 (1.29)	169.50 (5.45)
	Mod.	2.50 (1.00)	16.75 (3.86)	12.00 (4.24)	20.25 (2.87)	99.50 (4.20)	151.00 (14.09)
Flagstaff	Real	2.75 (0.50)	19.25 (0.50)	15.25 (1.26)	28.00 (1.83)	111.75 (2.99)	177.00 (4.69)
	Mod.	2.75 (0.50)	15.00 (2.71)	10.75 (3.59)	20.00 (2.83)	97.25 (4.92)	145.75 (5.91)
Chatterbox	Real	3.50 (1.00)	22.50 (5.69)	18.25 (3.96)	27.25 (6.55)	109.25 (11.18)	180.75 (27.61)
	Mod.	4.00 (0.82)	30.75 (19.43)	21.00 (18.92)	30.00 (12.44)	117.00 (20.61)	202.75 (50.79)
Ethernet	Real	2.25 (0.50)	12.50 (0.58)	7.75 (0.50)	17.50 (0.58)	84.00 (1.41)	124.00 (1.63)

This table gives the per-phase mean elapsed times in seconds, and the total benchmark time, of four trials of the Andrew Benchmark under real and modulated network conditions. For reference, the final row gives benchmark performance over the Ethernet used for modulation. Standard deviations are given in parentheses.

Figure 6.15: Elapsed Times for Andrew Benchmark Phases

In order to reproduce these simple, sharp changes in bandwidth with good experimental control, the evaluation makes use of trace modulation. Trace modulation emulates some desired network performance in an otherwise entirely live system; this emulation is driven by a trace, a list of parameters to a simple network model. Traces may be generated entirely synthetically to match any desired network performance; the only limit is the quality of the underlying network. This technique is used to generate the reference waveforms to which Odyssey and its applications will be subjected.

While Odyssey's evaluation focuses on impulse response analysis, one might wish to evaluate a system in the context of more realistic network performance. Unfortunately, wireless networks, in addition to having performance that is complex to describe, do not provide reproducible performance. Such reproducibility is critical in evaluating and comparing systems.

To address this, one can generate modulation traces empirically. An experimenter traverses some area of wireless network coverage, carrying an instrumented laptop that collects observations of the performance of a particular, known network workload. The observations are then distilled into a form suitable for trace modulation. A set of benchmarks shows that the resulting trace faithfully reproduces the original, measured environment. This despite the fact that the benchmarks exhibit a wide range of network traffic — all quite different than the workload from which model parameters are derived.

Chapter 7

Evaluation

The evaluation of the Odyssey prototype and applications was driven by three central questions:

- What are the limits on agility imposed by the system-level components of Odyssey?
- Are adaptive strategies effective in reducing the impact of large variations in bandwidth.
- Does centralized resource management allow multiple, concurrent applications to make better adaptive decisions than per-application approaches?

These questions were chosen to understand the architectural limitations of Odyssey, and the nature of application-aware adaptation. This chapter begins with an overview of these questions and the strategies used to answer them. The experimental conditions are given in Section 7.1, and the experiments are presented in Sections 7.2–7.4.

Agility is an end-to-end property, measuring the time required for a mobile client to notice and react to some salient change in its environment. Both the application and the system place limits on the agility of a client; a sluggish component anywhere in the adaptive chain renders the whole system unresponsive. The system-limited components of this chain are of particular interest, as they place the upper bound on agility for all applications. There are four system-limited components to agility: placing a resource request with the system, detecting a change in that resource, notifying the application of that change, and requesting a type-specific operation to change fidelity. These costs, described in turn in Section 7.2, are dominated by the detection of bandwidth changes; this takes on the order of a few seconds.

Next, Section 7.3 asks how effective adaptive strategies are in coping with changes in bandwidth. To do so, each application is extended to support a number of static strategies in addition to its adaptive one. Each strategy is subjected to the reference waveforms in Figure 6.1, and the resulting performance and fidelity is compared with that of the other strategies. When the waveform is one in which an adaptive strategy could benefit, it does; otherwise, the adaptive strategy mimics the correct static one.

Finally, Section 7.4 explores the importance of the single point of resource control in support of concurrent applications. To do so, Odyssey's centralized bandwidth estimation is compared to two alternative forms of estimation. Several experiments are run with concurrent applications making use of these estimation mechanisms over a variety of environments. In each of these, comparing the resulting performance and fidelity demonstrates clearly that centralized resource management is critical in providing applications with the information they need to make good adaptive decisions.

7.1 Experimental Conditions

All experiments used the same hardware and software configuration, shown in Figure 7.1: a single 90 MHz Pentium client with 32 MB of memory, and a collection of 200 MHz Pentium Pro servers with 64 MB of memory. The client ran a NetBSD 1.2 kernel customized to include Odyssey and trace modulation extensions. For simplicity, this software base was used on the servers as well, though trace modulation was performed only on the client.

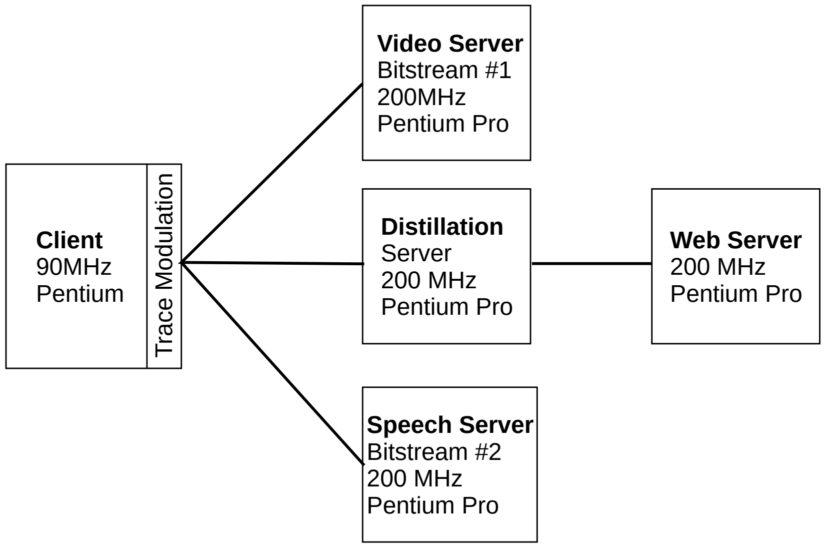


Figure 7.1: Experimental Set Up

For context, Figure 7.2 gives elapsed times for various micro-benchmarks on both client and server machines. Times for null a procedure call and `gettimeofday` were obtained by timing a million iterations of each, and dividing. Signal delivery time was obtained by running two programs that used signals to pass control to one another. The elapsed time of two hundred thousand such control exchanges was measured. This elapsed time includes one extra system call — the call to `sigpause` to await the next exchange of control — but the other costs, such as the time for a context switch, are incurred during the delivery of any signal. Intra-machine RPC time was obtained by measuring ten thousand calls between a client and server on the same machine; inter-machine time was measured from the client to one of the servers. The relatively long RPC time is a result of our untuned, user-level mechanism. The inter-machine RPC time is slightly lower than that for intra-machine RPC because the server CPU is substantially more powerful than that of the client, and this more than compensates for additional network delays.

Benchmark	Client	Server
Null Procedure Call	125 ns	42 ns
`gettimeofday`	12.8 μs	7.01 μs
Signal Delivery	61.1 μs	25.6 μs
Null RPC, intra-machine	1.26 ms	492 μs
Null RPC, inter-machine	1.20 ms	—

Figure 7.2: Measured Times of Basic Primitives

The client can run any of the sample applications, as well as a synthetic application called *bitstream*. Each server plays only a single role in any given experiment. These roles are labeled in Figure 7.1. The connections in Figure 7.1 are logical; all hosts in the testbed are connected to the same Ethernet segment.

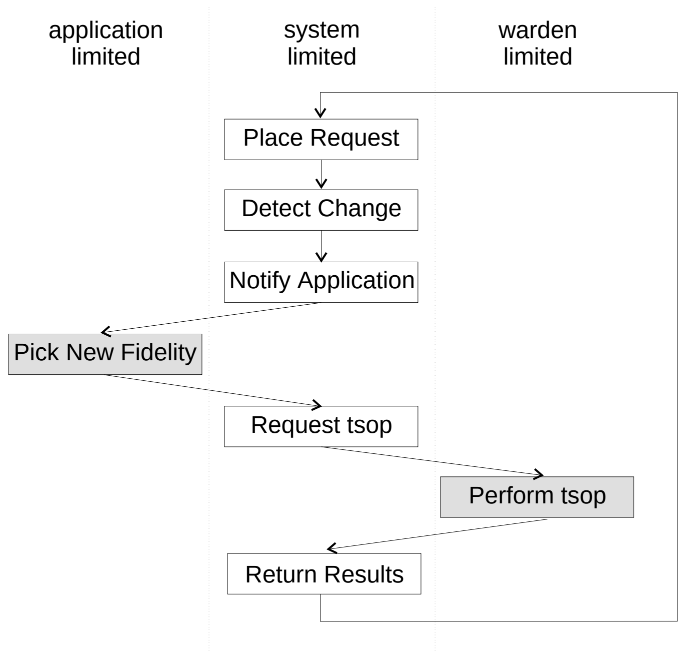
All experiments use either the reference waveforms of Figures 6.1 and 6.2, or — in the case of variations in network demand — a constant supply. The bandwidth levels used by the reference waveforms were chosen with two constraints in mind. First, they must be reasonably achieved on current wireless hardware. Second, they must provide for interesting tradeoffs when running the sample applications. The traces use 120 KB/s (kilobytes per second) and 40 KB/s for the high and low bandwidth levels. The protocol round trip time measured on the setup in Figure 7.1 was 21 ms for both bandwidths. The short waveforms of Figure 6.1 are one minute long, with the step occurring at the midpoint, and the impulses centered at the midpoint with a width of two seconds. The long waveform, in Figure 6.2 is fifteen minutes long.

Many experiments involve measurements relative to events in a reference waveform; it is important to synchronize points in a waveform trace with measurements taken in various processes. For example, when running an experiment with the Step-Up waveform of Figure 6.1(c), one might need to classify events in the viceroy, wardens, or applications as being either before or after the transition.

To cope with this difficulty, trace modulation was extended to use the upcall facility described in Section 3.3.5. Individual records in a modulation trace can be *marked*. There are three types of marks; *start*, *timestamp*, and *end*. When the modulation layer consumes such marked records, it generates an upcall in the trace modulation upclass as a side effect. Start and timestamp records are considered to be consumed when they begin; end records are considered to be consumed when they complete. Each of these upcalls carries with it the time at which the record was consumed, and the model parameters contained in that record. The viceroy registers for these upcalls, and when it receives one collects summary data from each warden.

7.2 Agility

The adaptive decision loop for an application, shown in Figure 7.3, determines the agility of that application; the amount of time taken to pass through that loop defines the most turbulent environment in which an application can operate. There are four components of this decision loop limited by Odyssey itself: the cost of placing a resource request, the agility in estimating resource availability, the cost of notifying an application of changes in that availability, and the time to invoke a type-specific operation and return its results.



This figure illustrates the decision loop of an adaptive application; the time it takes to traverse this loop defines the agility of any given application. The two shaded boxes represent steps limited by the agility of the application or the warden; the rest are system-limited steps, where the system includes the runtime library, kernel, and viceroy.

Figure 7.3: Adaptive Decision Loop

These components together place an upper bound on the agility of the system as a whole. An application that can immediately pick an appropriate fidelity level, using data from a warden that can effect fidelity changes without delay, is still limited by the remainder of the system. This section measures the agility of each of these four system-limited components.

7.2.1 Resource Requests

The cost of placing a request is a small component of the overall decision loop. The user-level library must record the address of the request handler in a data structure, and forward the request through the kernel to the viceroy, which places it on a resource request list.

To measure this cost, the viceroy was extended with one hundred test resources; this enables measurement overhead to be amortized across many requests without having them interfere with one another. A synthetic application then measures the total time it takes to place a request on each of these resources; division yields the time per request. The same was done to measure the cost of request cancellation. Taken over five trials, the time required to place one request was, on average, 768 μs, with a standard deviation of 30 μs. The time to cancel a request is slightly shorter: 671 μs, with a standard deviation of 24 μs. These times compare favorably with the intra-machine, null RPC time of 1.26 ms.

7.2.2 Bandwidth Estimation

In order to allow applications to make intelligent trade-offs between performance and quality, the viceroy must accurately track changes in both the supply of and demand for network bandwidth. This estimation, described in Section 4.5.2, is purely passive, and relies on observations of application traffic. From these observations, the viceroy estimates how much bandwidth is available to the machine in total — the supply of bandwidth — and then estimates the portion of that total that will be available to each active connection — the demand for bandwidth. Since the mechanisms to detect supply and demand are different, they must be measured independently.

The agility of bandwidth estimation is dependent upon the network demands of running applications; this is because estimation is purely passive. An application making heavy use of the network will have more accurate estimates than one that rarely uses the network. Since the goal of this section is to estimate the limits imposed by Odyssey, these experiments are carried out with a synthetic application, warden, and server, called *bitstream*.

The bitstream application reads bytes from some object at some rate from the bitstream server; this server generates an infinite stream of data for reading. The bitstream warden prefetches data on behalf of the application in chunks of 64 KB. There are two modes in which the bitstream application can operate. In the first, it fetches data as fast as it possibly can; since it is constantly fetching data over the network, there are always current observations in the RPC2 logs on which the viceroy can base its estimations. In the second, the application can rate-control its consumption, lowering the load it offers to the network.

Varying Supply

To measure the viceroy's agility in detecting variations in supply, a single bitstream was run at full-rate, subject to each of the reference waveforms in Figure 6.1. There were five trials taken for each waveform. To ensure that the system was in steady state at the beginning of each waveform, each trial was primed for thirty seconds prior to observation.

The bandwidth estimated by Odyssey for the Step-Up and Step-Down waveforms is shown in Figure 7.4, that for Impulse-Up and Impulse-Down is shown in Figure 7.5. Each graph depicts the waveform, as well as all estimates made by the viceroy. The waveform itself is shown as two curves. The uppermost, dashed curve shows the nominal bandwidth of the waveform. The lower, dotted curve shows the long-term throughput achieved by SFTP over a modulated network of that bandwidth. A perfectly agile estimator would produce observations entirely within these two curves.

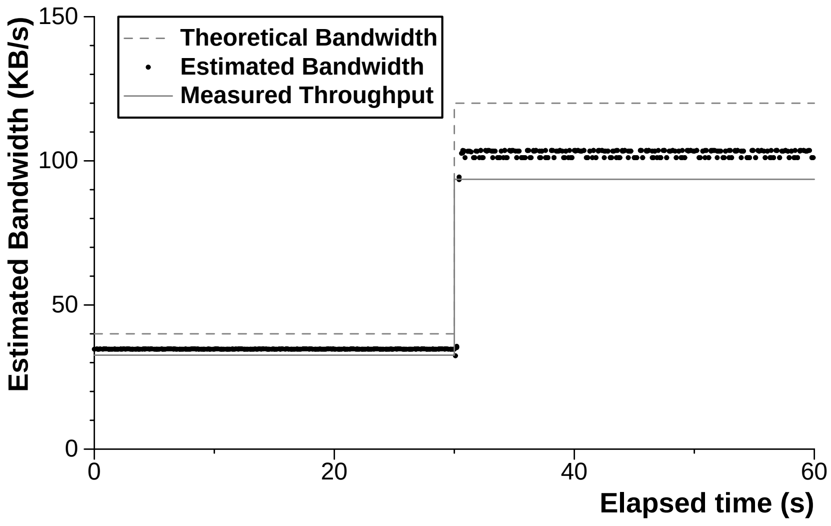
Figure 7.4(a) shows that Odyssey demonstrates excellent agility on the Step-Up waveform by detecting the increase in bandwidth almost instantaneously. There is no overshoot, and the settling time — the time required to reach and stay within the nominal bandwidth range — is at worst 0.5 seconds. The second graph, Figure 7.4(b), shows that agility on the Step-Down waveform is not quite as good as on Step-Up. The settling time for this waveform is as much as 2.0 seconds. However, there is no overshoot for Step-Down. The slower downward transition is caused by the fact that a throughput estimate is generated only at the end of a window of SFTP data, as shown in Figure 4.10. If bandwidth falls abruptly while a large window of data is being transmitted, the drop is not recorded until the last packet of the window arrives. Further, this window will be *elongated* in time; that is, the end of the window will arrive even later than it would have had the bandwidth not dropped, further delaying the observation.

Figures 7.5(a) and 7.5(b) show agility for the Impulse-Up and Impulse-Down waveforms. Neither waveform exhibits overshoot. For Impulse-Up, settling time in the worst case is 1.0 seconds. In both cases, the viceroy tracks the impulse.

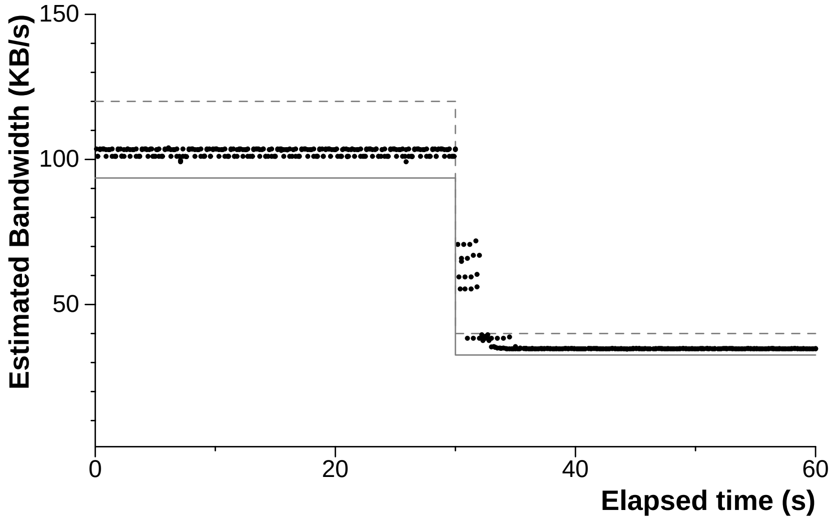
Varying Demand To measure agility with respect to bandwidth demand, the supply of bandwidth was held constant, but the demand for bandwidth was varied. At the beginning of the experiment, a single bitstream application is running; halfway through, a second, identical one is started. The viceroy's estimation of the total bandwidth available to the machine, as well as that available to the second, added stream, was measured. Ideally, the viceroy will yield a constant estimate of the total bandwidth, since the supply of bandwidth to the machine does not vary. It should also estimate that half of that bandwidth is available to the second bitstream client immediately after it is started.

To study sensitivity of the results to offered load, the experiment was run three different times, varying the demand placed by each individual instance of bitstream: 10% of the nominal bandwidth, 45%, or 100%. In the first two cases each instance is rate-controlled; in the other they are not. Thus, when the pair of bitstream applications were both running, the total bandwidth they attempt to consume is 20%, 90%, and 200% of that nominally available. In the first case, it is trivial to satisfy both bitstream demands. In the second, the two bitstream clients together attempt to consume at a slightly higher rate than can empirically be achieved given the 120KB nominal bandwidth; a single SFTP stream can, in the long term, only achieve 94KB/second, but 95% of the nominal bandwidth is 114KB. Clearly, in the third case the two streams will directly compete for more bandwidth than is available.

The results of these experiments are shown in Figure 7.6. The total bandwidth estimated for the machine is depicted in the upper curve; the bandwidth estimated to be available to the second, additional stream is shown in the lower curve. Recall that the viceroy will estimate bandwidth available to the two streams based on the 80/20 rule: 80% of the total



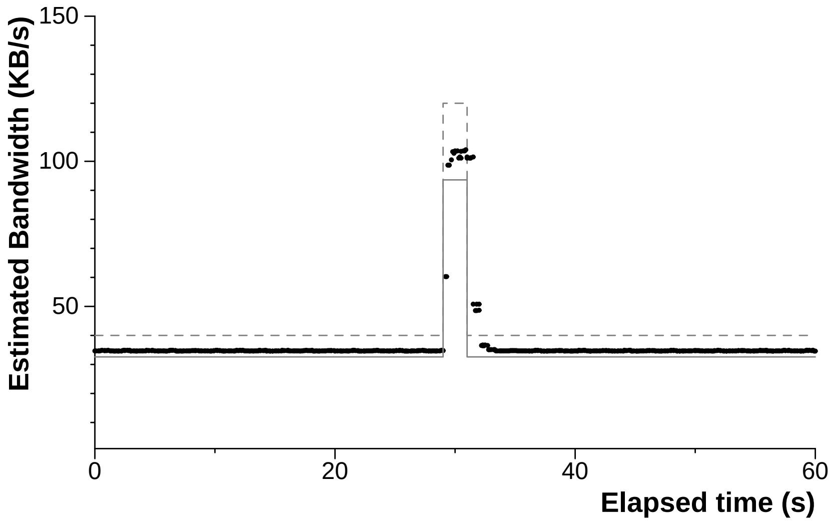
(a) Step-Up Waveform



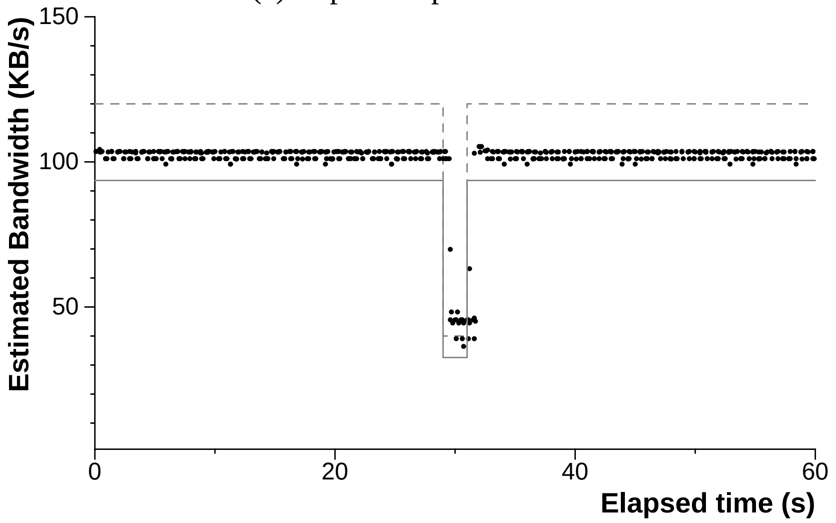
(b) Step-Down Waveform

This figure shows the agility of bandwidth estimation in the face of the Step-Up and Step-Down reference waveforms. Each graph merges the results from five trials, and each bandwidth observation is represented by a single dot on the graph. The dashed lines represent the nominal bandwidth of the emulated network, as specified by the synthetic traces used for emulation. The dotted lines are the measured, instantaneous throughputs obtained using a large bulk transfer between client and server. Ideally, all samples would lie between the dashed and dotted lines.

Figure 7.4: Supply Estimation Agility: Step-Up and Step-Down



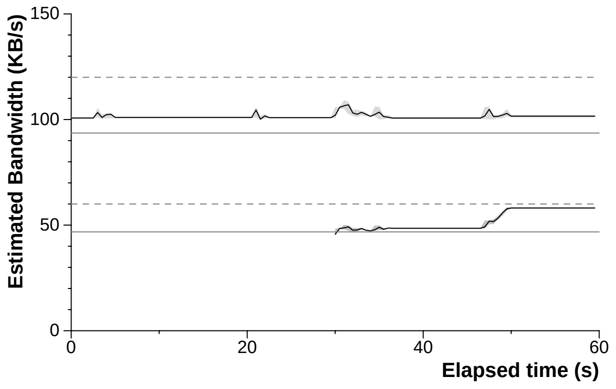
(a) Impulse-Up Waveform



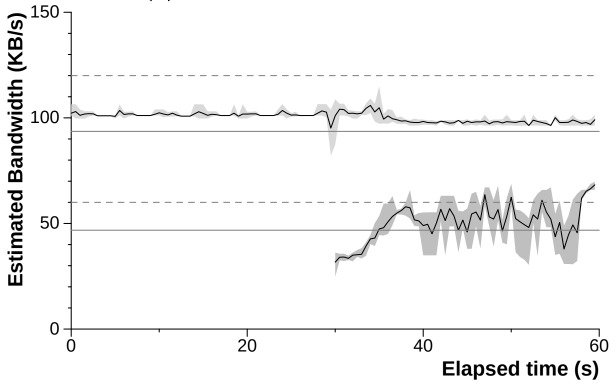
(b) Impulse-Down Waveform

This figure shows the agility of bandwidth estimation in the face of the Impulse-Up and Impulse-Down reference waveforms. Each graph merges the results from five trials, and each bandwidth observation is represented by a single dot on the graph. The dashed lines represent the nominal bandwidth of the emulated network, as specified by the synthetic traces used for emulation. The dotted lines are the measured, instantaneous throughputs obtained using a large bulk transfer between client and server. Ideally, all samples would lie between the dashed and dotted lines.

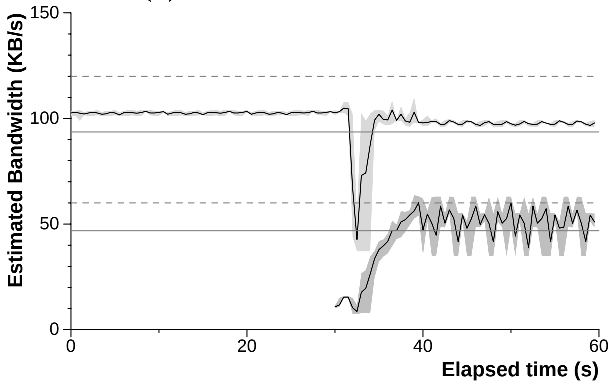
Figure 7.5: Supply Estimation Agility: Impulse-Up and Impulse-Down



(a) 10% utilization/stream



(b) 45% utilization/stream



(c) 100% (attempted) utilization/stream

This figure shows the agility of bandwidth estimation in the face of varying demand. The upper curve is the total estimated bandwidth; the lower is the bandwidth available to the second stream, which starts after 30 seconds of measurement. The pairs of straight lines show the nominal ranges for each curve; a perfectly agile system would always show the upper and lower curves within their respective pairs. Each graph depicts the results of five trials. The solid lines show averages, and gray regions show the spread between observed minimum and maximum values.

Figure 7.6: Demand Estimation Agility

bandwidth is allocated based on recent use, and the remainder is divided fairly.

Rather than show individual observations, each curve shows the average as the solid line in the middle of the curve, along with the range of observed values as the grey enclosing the curve. As before, the experiment was primed for thirty seconds with a single bitstream application in order to eliminate start-up artifacts.

In the low-utilization case, the viceroy correctly estimates both the total bandwidth as well as that available to the second stream. For the other two cases, settling time for the second stream can be long. In the mid-utilization case, the second stream first reaches nominal value within 5.8 seconds, on average. In the high-utilization case, it does so within 7.6 seconds on average. These settling times are due to the proportional division of bandwidth; since the ongoing stream has a history of using the link, and the new one does not, the former is given a larger proportional share until the latter catches up.

Additionally, the two high-utilization cases exhibit two troubling effects. The first is a transient in the estimation of total bandwidth upon commencement of the second stream. The second is oscillation in the estimate of the proportion available to the second stream.

The transient, while visible in both the 45% and 100% case, is most severe in the latter; in that case, the bandwidth available to the machine as a whole is estimated to be about 38K — less than 40% of the minimum expected value. It takes as long as 4.5 seconds for the viceroy to recover from this transient. This transient is likely caused by the connection establishment routine used by RPC2 interrupting an in-progress data transfer; since the first stream is constantly consuming data, any new connection is almost certain to interrupt a window's transfer.

The oscillation in estimated values is due to the combination of bitstream prefetching and the viceroy's mechanism to estimate the portion of total bandwidth to each connection. The prefetch algorithm works with fairly large chunks — 64KB at a time. At the network speed provided by modulation in these experiments, this takes slightly more than half a second. The viceroy, in estimating the proportion of total bandwidth available to each connection, biases that estimate in favor of recent use.

These two combine to produce the oscillating effect seen in Figures 7.6(b) and 7.6(c). The latter experiment, where each connection effectively takes turns using the machine's link, shows very regular oscillation between the first and second streams. However, the former, where each synthetic application occasionally pauses, shows a much less regular pattern.

7.2.3 Notifications

The next component to examine is the cost of notifying an application that the availability of some resource has changed in a significant way. Not surprisingly, this cost is larger than that to place a resource request, but comparable a null RPC call between two processes on the client. Since requests make use of several global data structures, process notification costs depend on the total number of processes with requests outstanding, whether or not they are to be notified.

To measure this cost the experiment sets up two test resources in the viceroy: `to-notify` and `not-notified`. At the beginning of each experiment, two sets of synthetic clients start up; each client in the first set places a resource request on the `to-notify` resource, while each of the others places a request on the `not-notified` resource. Once this has happened, the viceroy changes its estimate of the `to-notify` resource, and notifies each application in the first set; the total time required to perform this notification is recorded. One would expect that the number of processes in the `to-notify` set would be the largest contributing factor to this time, while the `not-notified` set would have a smaller impact.

Notified	Not Notified	Time (ms)	
1	0	1.17	(0.06)
	10	1.23	(0.03)
	20	1.35	(0.10)
2	0	1.87	(0.04)
	10	2.08	(0.01)
	20	2.20	(0.17)
5	0	4.09	(0.19)
	10	4.27	(0.05)
	20	4.45	(0.09)
10	0	8.11	(0.41)
	10	8.27	(0.20)
	20	8.37	(0.18)

This table shows the costs of notifying processes of changes in resource availability. Each major row gives the number of processes to be notified; each minor row gives the number of extra processes with requests outstanding on another resource for which no notifications are generated. The time values represent an average over five trials, with standard deviations given in parentheses.

Figure 7.7: Cost of Notification

The results for a set of these experiments appear in Figure 7.7. The cost to notify one process is comparable to the cost of intra-machine RPC on the client. However, as the number of processes that are notified increases, the time to notify all processes increases sub-linearly; this is because resource notifications are posted with asynchronous upcalls; the viceroy posts all notifications before yielding control. While the cost of notification does go up slightly with the presence of additional processes that are not notified, it does so only modestly, between 10 and 20 μs per process. However, these additional costs are worth further investigation.

7.2.4 Type-Specific Operations

The last system-limited component in the decision loop of Figure 7.3 is the cost to request a type-specific operation, and obtain its result. These operations, implemented by wardens, are typically used for changing fidelity or for performing type-specific access methods. Type-specific operations pass arguments in an unstructured buffer to the warden responsible for that type, and receive results in a separate unstructured buffer.

The costs of performing a type-specific operation can be broken down into three components: the time to pass the arguments to the warden, the time for the warden to perform the operation, and the time to return the results back to the caller. The second of these is properly the responsibility of the warden, not the system. The other two, however, are imposed by the system, and vary based on the size of the buffers to be passed.

To measure these costs, the root warden was extended with two trivial type-specific operations; `write` for passing buffers of varying sizes to the warden, and another, `read` for returning buffers of varying sizes back to the calling application; the latter passes in a very small buffer to the warden with the expected size of the result. A synthetic application then uses these type-specific operations, timing ten thousand iterations. This application was run five times for each size and direction of transfer.

Bytes	Write (μs)		Read (μs)	
4	578	(2.9)	599	(3.2)
256	585	(1.4)	608	(4.2)
512	606	(7.9)	624	(1.9)
1024	622	(3.2)	641	(5.3)
2048	696	(15.2)	693	(7.0)
4096	767	(16.9)	745	(9.5)
8192	1,949	(28.1)	1,934	(46.3)
16384	2,693	(24.0)	2,817	(31.9)
32768	4,616	(53.9)	4,749	(272.4)
65536	8,827	(193.3)	9,370	(244.4)

This table shows the system-imposed costs, in microseconds, to perform type-specific operations. Note that these costs do not include any warden-imposed processing delays; these can be unboundedly long. The first column gives the size of the buffer transferred by the type-specific operation. The second column gives the time to transfer a buffer from the application to the warden, while the third gives the time to transfer a buffer of the appropriate size from a warden to the calling application. Each time is an average of five trials, with standard deviations given in parentheses.

Figure 7.8: Costs of Type-Specific Operations

The costs to invoke type-specific operations are shown in Figure 7.8. At small sizes — 4 KB or less — the costs are dominated by the time to pass the request from the application through the kernel and up to the viceroy and warden. At 8 KB or higher, transfer times begin to dominate the costs of inter-process communication through the interceptor.

The jump from 4KB to 8KB is particularly interesting; the time for 8KB is more than twice the time for 4KB. Profiling the kernel shows that copying data buffers larger than 8KB between user and kernel space gives rise to significant fault overhead in the virtual memory subsystem. Performing two regressions on the data in Figure 7.8, one for sizes up to 4KB, and one for sizes 8KB and greater, gives rise to the following piecewise-linear function:

$$T_{\text{write}} = \begin{cases} 579\mu\text{s} + 48\text{ns}(b) & \text{if } b <= 4\text{KB} \\ 791\mu\text{s} + 121\text{ns}(b) & \text{if } b >= 8\text{KB} \end{cases} \tag{7.1}$$

$$T_{\text{read}} = \begin{cases} 603\mu\text{s} + 36\text{ns}(b) & \text{if } b <= 4\text{KB} \\ 705\mu\text{s} + 131\text{ns}(b) & \text{if } b >= 8\text{KB} \end{cases} \tag{7.2}$$

In other words, per-byte costs for sizes greater than 8KB are three times that for sizes 4KB or less.

To ensure that this overhead is not due to a performance problem in `tsop` itself, an experiment was designed to measure the costs in copying data from user space to kernel space. A simple system call was added to the kernel which is passed a pointer to a buffer, and the size of that buffer. It allocates an equally-sized buffer in the kernel, copies the user-space data to kernel-space, then disposes of the buffer and returns. The results of this experiment, shown in Figure 7.9, show a severe jump in copy costs at 16KB or larger. In fact, the jump occurs at exactly 8,193 bytes; the average costs for such a copy are 1,017 μs. The `tsop` results show a jump at an argument size of 8KB because the `tsop` must transfer some additional data along with the argument buffer, putting it over the copying limit. A regression on the copy costs alone suggest costs of 10 ns per byte for copies of 8KB or less, and 90 ns per byte for copies of greater than 8KB.

7.3 Individual Applications

The evaluation next turns to the degree to which adaptive strategies can cope with sharp changes in bandwidth. Each application's adaptive data access strategy is compared to a set of possible static strategies; each was run subject to the reference waveforms in Figure 6.1 to see which of these best satisfies the application's goals. Unlike the experiments of the last section, answering this question involves a complete, end-to-end system of application, viceroy, warden, and server.

In these experiments, each application executed the same workload regardless of strategy. Five trials of each strategy were taken for each waveform. As in the bandwidth experiments of Section 7.2.2, each trial was primed for thirty seconds before beginning measurement to eliminate startup transients. The only difference between runs was the fidelity policy chosen by the applications; all other factors remained the same.

Bytes	`copyin` time (μs)	
4	10	(0.2)
256	14	(0.1)
512	17	(0.5)
1024	24	(0.1)
2048	36	(0.1)
4096	91	(2.4)
8192	126	(0.4)
16384	1,403	(11.6)
32768	2,684	(49.2)
65536	5,408	(128)
131072	11,670	(187)

This table shows the time required for a system call to copy a buffer of size b bytes from user-space to kernel-space, in microseconds. The first column gives the size of the buffer copied. The second column gives the time to copy that buffer; this is an average of five trials, with standard deviation given in parentheses.

Figure 7.9: Costs of Copying Data from User to Kernel Spaces

7.3.1 Video Player

XAnim has three static strategies in addition to its adaptive strategy of playing the highest quality frames without loss: always play JPEG(99) frames, always JPEG(50), and always black-and-white. The higher bandwidth of the reference waveforms is sufficient to fetch JPEG(99) frames without loss. At the low bandwidth, however, at best JPEG(50) frames can be fetched without loss.

As defined in Section 5.2, JPEG(99), JPEG(50), and black and white frames have fidelities of 1.0, 0.5, and 0.01 respectively. For a single execution, XAnim's achieved fidelity is the average of the fidelities of all displayed frames. Each reference waveform is one minute long; since the movie tracks are all encoded at ten frames per second, there are 600 frames to display during a single execution.

Figure 7.10 summarizes the performance and fidelity of each strategy over the four reference waveforms. Each row in the table gives the results for a separate waveform. The first three columns give the results for the three static strategies; the fourth column gives results for the adaptive strategy.

Neither of the first two strategies — always black and white and always JPEG(50) — drop an appreciable number of frames. This is not surprising. The lowest bandwidth level is sufficient to fetch the JPEG(50) track without loss, so switching between it and the higher bandwidth should not cause frames to be dropped. However, the JPEG(99) strategy drops more than 35% of all frames on the Step waveforms, and more than 54% of all frames on the Impulse-Up waveform; the periods of low bandwidth cannot support the JPEG(99) track.

Waveform	B/W Fidelity = 0.01 Drops		JPEG(50) Fidelity = 0.5 Drops		JPEG(99) Fidelity = 1.0 Drops		Adaptive Drops		Adaptive Fidelity	
Step-Up	0	(0.0)	3	(1.8)	169	(0.8)	7	(2.2)	0.73	(0.01)
Step-Down	0	(0.0)	5	(11.2)	169	(2.4)	25	(8.9)	0.76	(0.01)
Impulse-Up	0	(0.0)	3	(0.7)	325	(4.3)	23	(7.4)	0.50	(0.01)
Impulse-Down	0	(0.0)	0	(0.0)	12	(5.7)	14	(6.5)	0.98	(0.01)

This table gives the achieved fidelity and number of frames dropped by XAnim under various strategies for each of the four reference waveforms. Each observation in the table is the mean of five trials, with standard deviations given in parentheses. Notice that Odyssey achieves fidelity as good as or better than the JPEG(50) strategy in all cases, but performs as well or better than JPEG(99) within experimental error.

Figure 7.10: Video Player Performance and Fidelity

Because the adaptive strategy adjusts with changing bandwidth, it provides a better balancing between fidelity and performance on the reference waveforms. For both the Step-Up and Step-Down waveforms, the adaptive strategy achieves fidelities very near 0.75; it correctly displays half of its frames from the JPEG(99) track, and half from the JPEG(50) track. In Step-Up, it is slightly lower. This is because the warden yields lower quality frames while waiting for the first block of frames to arrive after the increase in bandwidth. Likewise, it is slightly higher in Step-Down due to the delay in detecting the drop in bandwidth. During the Impulse-Up waveform, only JPEG(50) frames are displayed by the adaptive strategy, as expected. Likewise, during Impulse-Down, JPEG(99) frames are displayed most of the time.

In all cases, the adaptive strategy drops less than 5% of frames to be displayed. Most dropped frames occur during downward transitions. Immediately before such transitions, XAnim is displaying JPEG(99) frames. When the transition happens, a block of such frames is in transit. This block will necessarily be late, and the frames contained by it will be dropped.

The adaptive strategy comes closest to achieving XAnim's goal of playing the highest quality frames without dropping them. While the JPEG(50) and black and white strategies drop slightly fewer frames, the adaptive strategy achieves a much better fidelity on all waveforms other than Impulse-Up, where they are equal. Conversely, the adaptive strategy drops far fewer frames than the JPEG(99) strategy for all waveforms other than Impulse-Down, where the two strategies are indistinguishable.

7.3.2 Web Browser

The Web browser has four static strategies in addition to its adaptive strategy of fetching the best image possible in twice the Ethernet fetch time: always fetch images compressed to JPEG quality 5, always fetch JPEG(25) images, always JPEG(50), or always full quality. The workload used in these experiments is the repeated fetching of a single 22 KB image as fast as possible.

As defined in Section 5.3, full-quality, JPEG(50), JPEG(25), and JPEG(5) images are assigned fidelities of 1.0, 0.5, 0.25, and 0.05, respectively. For a single execution, Netscape's achieved fidelity is the average of the fidelities of all displayed images.

The performance metric is the average time to fetch and display an image during an execution. The time to fetch and display the workload image on an unloaded Ethernet is 0.2 seconds, giving a target fetch time of 0.4 seconds. The high bandwidth level in the reference waveforms is sufficient to fetch full-quality images; at the low level, JPEG(50) is the best possible, though it is very close to the transition point between JPEG(50) and full quality.

Waveform	JPEG(5) Fidelity 0.05 Time (s)	JPEG(25) Fidelity 0.25 Time (s)	JPEG(50) Fidelity 0.5 Time (s)	Full Quality Fidelity 1.0 Time (s)	Adaptive Time (s)	Adaptive Fidelity
Ethernet	—	—	—	0.20 (0.00)	—	—
Step-Up	0.25 (0.01)	0.30 (0.01)	0.29 (0.01)	0.46 (0.01)	0.35 (0.05)	0.78 (0.08)
Step-Down	0.25 (0.01)	0.30 (0.01)	0.29 (0.01)	0.46 (0.00)	0.35 (0.03)	0.77 (0.04)
Impulse-Up	0.27 (0.01)	0.33 (0.01)	0.34 (0.00)	0.71 (0.00)	0.42 (0.06)	0.63 (0.08)
Impulse-Down	0.24 (0.01)	0.27 (0.02)	0.29 (0.01)	0.34 (0.01)	0.36 (0.02)	0.99 (0.01)

This table gives the fidelity and average time for Netscape to fetch and display our test image under various strategies for each of the four reference waveforms. Each observation is the mean of five trials; standard deviations are given in parentheses. Notice that Odyssey achieves a better fidelity than JPEG(50) in all cases and, unlike the full-quality strategy, meets our performance goal within experimental error for all cases.

Figure 7.11: Web Browser Performance and Fidelity

Figure 7.11 summarizes the performance and fidelity of each strategy over the four reference waveforms. For comparison, the table also includes the time to fetch the image over unloaded Ethernet. The first four columns give the results for the four static strategies; the fifth column gives the results for the adaptive strategy.

All of the first three strategies — JPEG(5), JPEG(25), and JPEG(50) — fetch the image within the target of 0.4 seconds, in line with the expectation that the lowest bandwidth level can support qualities up to JPEG(50). However, the strategy of always fetching the best-quality image only meets the performance goal for the Impulse-Down case; in this last waveform, the bandwidth is insufficient to support the strategy only during the very short impulse.

For all waveforms, the adaptive strategy achieves a higher fidelity than 0.5 within the performance goal. Furthermore, for the Impulse-Down waveform, the adaptive strategy is indistinguishable from the strategy of fetching the best image at all times. For both the Step-Up and Step-Down waveforms, the adaptive strategy achieves fidelities close to 0.75

by fetching JPEG(50) images during low bandwidth and full-quality images during high bandwidth. The actual fidelities are higher because the cellophane occasionally switches to the best quality when the bandwidth estimate temporarily reaches the transition point. This phenomenon is also evident in the Impulse-Down waveform; fully one quarter of all displayed images were full quality.

As with XAnim, the adaptive strategy provides the best match for the cellophane's goal. It provides better fidelities than the JPEG(5), JPEG(25), and JPEG(50) strategies, while meeting the performance goal in all cases. In contrast, the full-quality strategy meets its performance goal only in the Impulse-Down case, where it is indistinguishable from the adaptive strategy.

7.3.3 Speech Recognition

	Recognition Time (sec.)		
Waveform	Always Hybrid	Always Remote	Odyssey
Step-Up	0.80 (0.00)	0.91 (0.00)	0.80 (0.00)
Step-Down	0.80 (0.00)	0.90 (0.00)	0.80 (0.00)
Impulse-Up	0.85 (0.00)	1.11 (0.00)	0.85 (0.00)
Impulse-Down	0.76 (0.00)	0.77 (0.00)	0.76 (0.01)

This table gives the average time, in seconds, to recognize the benchmark utterance for the two static strategies as well as the adaptive strategy for each of the four reference waveforms. Each observation is the mean of five trials. Standard deviations are shown in parentheses. Note that Odyssey correctly reproduces the always-hybrid case, which is optimal at our reference bandwidth levels.

Figure 7.12: Speech Recognizer Performance

The speech recognition system has two static strategies in addition to its adaptive strategy: always recognize remotely, or always perform hybrid recognition. Recall that the client will not resort to local recognition unless the bandwidth falls to zero. Regardless of strategy, though, the quality of speech recognition is the same. Thus, there is only a single fidelity; the adaptive strategy aims only to minimize the time for recognition.

For the speech experiments, we recognized a single, short phrase, repeating the recognition as quickly as possible. Figure 7.12 gives the recognition times. The first two columns give the times for the two static strategies, the third for the adaptive one.

At both bandwidth levels in the reference traces, hybrid translation is always the correct strategy. As the results show, the adaptive strategy correctly duplicates the always-hybrid strategy. At higher bandwidths, an adaptive strategy would have benefits similar to those shown in Figures 7.10 and 7.11

7.4 Concurrent Applications

The final evaluation task is to examine the degree to which a central point of resource control allows for more accurate adaptation decisions when compared to per-application resource management. To do so, the unified bandwidth estimation strategy described in Section 4.5.2 is compared to two types of per-application estimation under a variety of circumstances. In the first of these estimation mechanisms, each application naively assumes that it will receive the full nominal bandwidth of the link. In the second, each application estimates its bandwidth based only on its own network traffic, rather than that of the entire machine.

These mechanisms are compared in different experiments. In the first, each Odyssey application is run concurrently, subject to the fifteen minute reference waveform depicted in Figure 6.2. In this experiment, each of the three applications exhibits very different data rates; XAnim consumes the most bandwidth, while Janus consumes the least.

In the second experiment, two copies of XAnim are run simultaneously, each playing a copy of the same movie, streamed from different servers. This gauges the effect of running two applications with the same data rate under unified and per-application estimation strategies.

In the third and final experiment, two copies of XAnim are run subject to an empirical trace of one of the scenarios described in Section 6.5.2. This trace re-creates a much more complex environment than that of the synthetic reference traces, but is a useful example of the value of unified estimation in a more realistic setting.

7.4.1 Different Data Rates

The first concurrent experiment consists of one copy of each of the three Odyssey applications, running concurrently. XAnim plays a single movie that is longer than the fifteen minute reference waveform. Netscape repeatedly fetches a single, uncached image, and Janus repeatedly recognizes the same, short utterance.

This experiments is repeated for three bandwidth estimation mechanisms: one uses Odyssey's centralized bandwidth estimation, and the other two use alternate forms of estimation — *blind optimism*, and *laissez-faire*. This section begins by describing these two mechanisms, and then turns to the comparison between them and Odyssey's unified estimator.

The first alternative, blind optimism, captures what a naive mobile client might do in the presence of an overlay network. Whenever this client changes from one network technology to another, it notifies all of its applications of the nominal speed of the new network. Each then proceeds under the assumption that it will receive all of the nominal bandwidth. Such notification is immediate; there is no delay necessitated by observation. However, it is overly optimistic, as it cannot account for contention between applications.

Blind optimism is implemented in the viceroy using the network modulation upcalls described in Section 7.1; the transition points in the waveform are marked to generate timestamp upcalls. When the viceroy receives a modulation upcall, it uses the bandwidth value in the upcall to set the bandwidth of each active connection.

The second alternative, *laissez-faire*, duplicates the behavior of applications adapting in isolation. In this approach, each application estimates the network bandwidth independently from the others. While *laissez-faire* estimation is more accurate than blind optimism, it does suffer from the delays inherent in passive estimation.

Laissez-faire estimation is implemented with a simple modification to the viceroy's unified estimation algorithm. Rather than build a master log of all connections, the viceroy examines each log independently; it sets the bandwidth for a connection based only on the information known to that connection.[1]

In comparing these two alternative mechanisms with Odyssey's global estimator, all other factors remain the same. Compared to a real implementation of blind optimism or *laissez-faire*, this does add the costs of placing resource requests, and notifying when they are violated. However, as shown in Sections 7.2.1 and 7.2.3, these costs together are approximately two milliseconds; quite small considering the time scales of the applications in question.

	Video		Web		Speech
	Drops	Fidelity	Seconds	Fidelity	Seconds
Odyssey	1018 (48.6)	0.25 (0.00)	0.54 (0.02)	0.47 (0.01)	1.00 (0.01)
Laissez-Faire	2249 (80.2)	0.39 (0.01)	0.95 (0.03)	0.93 (0.01)	1.21 (0.01)
Blind Optimism	5320 (23.3)	0.80 (0.00)	1.20 (0.00)	1.00 (0.00)	1.26 (0.02)

This table demonstrates the benefit of Odyssey's centralized resource management by comparing it to two implementations of uncoordinated estimation. In this experiment, XAnim, Netscape, and Janus are all run concurrently. The fidelity and performance metrics for these applications are the same as in Figures 7.10–7.12. The total number of video frames to display is 9,000 in all cases.
Notice that by degrading the fidelity of fetched video and web data, Odyssey comes closer to each application's performance goals by factors of 2-5. Such a trade-off is made possible by Odyssey's more accurate estimation of bandwidth available to each application. Each observation in this table is the mean of five trials, with standard deviations given in parentheses.

Figure 7.13: Concurrent Applications: Different Data Rates

Figure 7.13 summarizes the results of this comparison. Each row depicts the results for a particular estimation mechanism: Odyssey's unified estimation, *laissez-faire* estimation, and blind optimism. The first two columns give the performance and fidelity metrics for XAnim and Netscape, respectively. The third column gives the average time for speech

[1]This does not correctly model true *laissez-faire* in the cases where a single application has more than one connection. However, for the applications considered here, this is not an issue.

recognition.

Clearly, Odyssey's centralized resource management provides significant benefits over both *laissez-faire* and blind optimism. By correctly accounting for bandwidth competition, the Web browser and video player fetch data at lower fidelity, thus enabling all applications to come much closer to their performance goals. XAnim under global estimation drops a factor of 2 to 5 fewer frames than the other strategies, and Web pages are loaded and displayed roughly twice as fast. The resulting decrease in network utilization improves speech recognition time as well.

However, even under centralized resource estimation, the applications do not entirely meet their performance goals. There are two main reasons for this. First, Odyssey provides only passive estimation, not active control; thus individual applications still tend to interfere with one another by inefficient scheduling of the network device. Second, the network is not the only over-constrained resource in this experiment; the CPU is also in heavy contention, causing late frames and longer display times due to delays in frame and image decoding.

7.4.2 Similar Data Rates

The three applications in the experiment of Section 7.4.1 have vastly different data rates. In this experiment, which examines the behavior of the estimators for applications with similar data rates, two different XAnim clients play the movie from the previous experiment, served from separate servers. They are again subject to the 15-minute tourist waveform, and unified estimation is compared to per-application estimation.

One would expect that in the case of the unified estimator, one of the video players is given preferential treatment over the other; it is called the *favored* player. This is due in part to the mechanism dividing bandwidth between multiple, competing connections. Connections with a recent history of more network use will be told that they will get a larger share of future available bandwidth. Therefore, when averaging the five trials of this experiment, the player with the highest achieved fidelity is always considered to be Video #1.

The results for this experiment are shown in Figure 7.14. The video players, when run under blind optimism, frequently stalled out in the middle of the trace. This is due to an over-taxing of the network, combined with some implementation inefficiencies in the video warden; thus, the blind-optimism result could not be reported.

The results show that, for the unified estimator, there is indeed a favored player — one with a higher achieved fidelity. However, the favored player also has a lower number of dropped frames. Furthermore, the *laissez-faire* estimator also shows evidence of a favored player, with both higher fidelity and a lower number of dropped frames. This cannot be explained simply by higher bandwidth estimations.

This likely results from an implementation problem in the video warden; for example, some global lock that is shared between two video streams to separate servers. This problem, in the presence of a heavily subscribed network, causes one of the video players to fall behind; it is also responsible for the inability to complete the blind optimism trials. Thus,

	Video #1		Video #2	
	Drops	Fidelity	Drops	Fidelity
Odyssey	801 (620)	0.33 (0.03)	2905 (812)	0.16 (0.06)
Laissez-Faire	2787 (763)	0.55 (0.07)	4538 (1090)	0.41 (0.09)

This table demonstrates the benefit of Odyssey's centralized resource management when the concurrent applications have similar bandwidth demands. In this experiment, two instances of XAnim are playing the same movie from two different servers, using the fidelity and performance metrics shown in Figure 7.10. The first column, Video #1, gives the average fidelity and performance of the *favored* video player over the five trials. For each trial, the video player achieving the highest fidelity is considered favored. Under blind optimism, the two video players were unable to complete the movie due to overtaxing of the network and inefficiencies in the video warden. The total number of video frames to be displayed is 9,000 in all cases.

Figure 7.14: Concurrent Applications: Same Data Rates

while the bandwidth estimator's logs do show that one of the connections does receive higher estimates on average, it is not the only factor. In fact, there is no correlation between a connection having a higher bandwidth estimate on average, and that connection's video player having a higher fidelity.

Despite this problem, the results do show that the unified estimator does provide for better decisions on the part of applications than the *laissez-faire* estimator does. The video players, subject to the latter estimator, drop between 30% and 50% of the 9,000 frames to be displayed. In contrast, under the unified estimator, only 9% to 32% of the frames are dropped, even with the warden inefficiencies. As with the experiment in Section 7.4.1, this is due to a more realistic choice of fidelities under unified estimation.

7.4.3 More Realistic Environments

The two experiments in Sections 7.4.1 and 7.4.2 both use the synthetic tourist waveform. While illustrative, those results may not be representative of a real wireless environment. Therefore, this experiment uses an empirical trace of one of the wireless scenarios of Section 6.5.2.

The experiment uses a trace of the Wean scenario. It was chosen for two reasons. First, it exhibits the largest regular fluctuations in bandwidth of any of the scenarios. Second, the loss rates, particularly during the elevator ride, can also give rise to significant decreases in bandwidth. Since the bandwidth of this scenario is larger than that in the reference traces, this experiment also uses two instances of XAnim; this combination is the most demanding of network bandwidth.

The results for this experiment are shown in Figure 7.15. As before, each estimator exhibits a favored video player that achieves both a higher fidelity and a lower percentage

	Video #1		Video #2	
	Drops	Fidelity	Drops	Fidelity
Odyssey	435 (115)	0.70 (0.05)	574 (115)	0.59 (0.05)
Laissez-Faire	630 (75)	0.94 (0.03)	1158 (173)	0.88 (0.04)
Blind Optimism	911 (255)	1.0 (0)	1011 (307)	1.0 (0)

This table demonstrates the benefit of Odyssey's centralized resource management in a realistic environment; this environment is recreated using an empirically-derived modulation trace. In this experiment, two instances of XAnim are playing the same movie from two different servers, using the fidelity and performance metrics shown in Figure 7.10. The first column, Video #1, gives the average fidelity and performance of the *favored* video player over the five trials. For each trial, the video player achieving the highest fidelity is considered favored. The total number of video frames to be displayed is 1,750 in all cases.

Figure 7.15: Concurrent Applications: Realistic Environment

of dropped frames. This is again likely due to difficulties in the video warden.

Perhaps surprisingly, the number of dropped frames is high even under unified estimation. This is partly due to of the large rate of dropped frames in the middle of the trace; they cannot be avoided, and will result in lost frames. Another cause for these drops is the rate at which the effective bandwidth changes in this more realistic scenario; it is quite high, and the two observational estimators — unified and *laissez-faire* — have trouble keeping up. This results in the same sort of behavior as is seen for the video experiment under the Impulse-Up waveform, shown in Figure 7.10. In this waveform, the XAnim client was fooled into fetching frames of too high a quality by a transient increase in bandwidth.

In the face of these difficulties, the unified estimator still provides the best match for XAnim's adaptive goal. Under unified estimation, the players drop between 24% and 32% of their frames. Under *laissez-faire* drops range from 36% to 66%, and blind-optimism drops between 52% and 57% of all frames.

7.5 Summary

This chapter presents the answers to three central questions concerning Odyssey. First, what is the most agile an Odyssey application can be given the current system implementation? Second, can adaptation to changes in bandwidth provide concrete benefits to applications? Third, how important is centralized resource management in providing accurate information to concurrent, adaptive applications?

There are four system-level components that contribute to limits on the agility of any Odyssey application. They are: the time to place a resource request, the delay in recognizing a change in that resource, the time to notify the requesting process of that change, and the overhead in requesting a type-specific operation and receiving its results. In the current prototype, the viceroy detects changes in bandwidth on the order of a few seconds; this dominates the other costs, which together amount to a few milliseconds.

To examine the benefits of adaptive strategies in the face of varying bandwidth, each application was extended to support a set of static strategies. These strategies were compared to the adaptive form of the application, subject to the reference waveforms. For each application, when the adaptive strategy can benefit, it does; otherwise, it mimics the correct static strategy.

Finally, Odyssey's centralized bandwidth estimation mechanism was compared to two per-application mechanisms. In the first, blind optimism, each application is notified of the nominal link bandwidth at the instant it changes. In the second, *laissez-faire*, each application estimates its effective bandwidth in isolation. When compared with one another in the face of concurrent, competing applications, global estimation comes two to five times closer to meeting applications' goals. These results hold without regard to the relative bandwidth needs of the concurrent applications, and bear out under both synthetic environments as well as more realistic and noisy ones.

Together, these experiments provide compelling evidence that Odyssey provides both the necessary and sufficient mechanisms for supporting diverse, concurrent adaptive applications. The system is agile with respect to changes in the availability of resources. Individual applications directly benefit from adaptation, and centralized resource control is necessary to the making of good adaptive decisions.

However, two further steps would be required to provide irrefutable evidence. First, a second application with a different adaptive policy for one of the existing data types would have to be written; this would empirically demonstrate Odyssey's ability to support diverse applications. Second, the viceroy would have to be extended to support all of the over-constrained resources, not just the network; this should enable applications to come much closer to meeting their adaptive goals.

Chapter 8

Related Work

To the best of the author's knowledge, Odyssey is the first system to simultaneously address the problems of adaptation for mobility, application diversity, and application concurrency. It is the first effort to propose and implement an architecture for application-aware adaptation that pays careful attention to the needs of mobile computing. The identification of agility as a key attribute for mobile systems, and the first approach to evaluating it, have both occurred in the context of Odyssey.

That said, Odyssey has benefited considerably from previous work. The most substantial debt is owed to Coda, though other systems have contributed in ways large and small. These contributions are described in Section 8.1. More recently, a number of other researchers, discussed in Section 8.2, have come to address the problem of adaptation in mobile systems.

An important secondary contribution of this dissertation is the suite of trace modulation tools, particularly empirical modulation. This technique is the first to combine three distinct ideas into a single tool:

- trace collection to accurately capture observed network behavior;
- reduction of observations into a time series of parameters for a simple network model;
- application-transparent network emulation through model-driven delays and losses of packets in a layer below the API of an operating system.

However, each of these has been invented in isolation by other researchers, as discussed in Section 8.3. Finally, Section 8.4 concludes with two systems that include techniques of possible use in Odyssey.

8.1 Design and Implementation Heritage

Many systems have contributed to the design and construction of Odyssey. The largest such contribution is from Coda [39, 50, 67], which first demonstrated that client resources could be used to insulate users and applications from the vagaries of mobile information access. As detailed in Chapters 3 and 4, many aspects of Odyssey — including its implementation in user space, the use of an in-kernel interceptor, and single, global name space — were based on positive experience with similar strategies in Coda. The threading and remote procedure call packages were taken from Coda without modification.

The notion of trading file consistency for availability and performance was put forth by systems such as Coda and Ficus [59]. The Bayou project applied similar techniques to databases [77]. It was the recognition that consistency represented only a single dimension of the broader concept of fidelity that led to the design of Odyssey.

Several systems served as the background to the taxonomy of adaptive systems. Mobisaic [81], W4 [9], and Dynamic Documents [36] all cope with mobility within the context of a single application — a web browser. Many similar stand-alone applications exist in the commercial marketplace, due to the inability to modify the operating system. These applications, such as the Eudora mail client [60], have met with substantial success. Finally, toolkits such as Rover [34] and Fox's dynamic distillation infrastructure [24] also take an application-only approach to mobility. The key feature missing from all these systems, with the possible exception of Rover, is the centralized management of resources. Rover does have a single point through which all network traffic passes; however, it is used primarily for scheduling different classes of traffic, and the author is aware of no published work addressing Rover's use for concurrent applications.

Additionally, the genesis of many of the individual ideas in Odyssey can be traced to earlier work. Upcalls were first proposed by Clark in the Swift operating system [15]. The idea of having type-specific behaviors for the standard file interface is a more general form of the extensible name resolution provided by the Semantic File System [25]. While Web proxies were originally intended to deal with firewalls and provide caching [43], they were used in Rover and Dynamic Documents to add functionality to the browser.

Finally, the installation of pieces of code at low levels of the system to encapsulate specialized knowledge about different data types is a common practice in databases [26]. The primary purpose of such code is to improve disk management. The use of wardens in Odyssey resembles this practice, but differs in that wardens support multiple fidelity levels.

8.2 Comparable Systems

Since the first publication of Odyssey's philosophy and design [53, 69], several other researchers have built systems providing some form of mobile data access. Such systems fall into three distinct categories: real-time approaches, software feedback systems, and systems similar in spirit to Odyssey.

8.2.1 Real-Time Approaches

The first set of systems bring real-time techniques to bear on the problem of mobile data access. Such techniques have been an important focus of the high-performance networking community, and have more recently been applied to CPU scheduling [47] and other resources. The basic ideas in real-time systems are admission control and resource reservation. When a new task enters the system, it specifies both the workload it will present and the performance it will demand. If the workload of the task cannot be added without violating any previous performance guarantees, or the requested performance cannot be met, it is rejected. Otherwise, it *reserves* the resources necessary to provide the requested performance, and is admitted to the system.

Resource reservation is geared towards systems where the workload is dynamic, but the environment is relatively static; it can be impossible to maintain previously established guarantees in the presence of a highly variable, mobile environment. Therefore, the real-time community has applied reservation techniques to the problem of mobile data access with two modifications [42, 58]. First, rather than reserving a particular quantity of a resource, they reserve a range; the underlying system transparently adapts within the range. Second, if the range is exceeded or the client moves, a renegotiation involving some or all of the end-to-end path is initiated.

In contrast to these systems, Odyssey abandons the reservation model entirely. If a reservation range is too wide, the burden of adaptation is placed on the system with the range, degenerating to application-transparent adaptation. If the range is very narrow, the application will be forced to renegotiate frequently. Such renegotiations are expensive, though Campbell has made progress for the special case of cell handoff [13].

Another approach that has arisen from the real-time community is to make *market-based* decisions; Abdelzaher [1] provides an example of such a market-based system. In these systems, a new application declares the qualities of data it is prepared to accept, and associates a *utility* with each of these qualities. These utilities are to be expressed in some currency common to all applications — for example, money that the application will pay for the service over time. For current algorithms to be efficient, they must also be long-lived and static. Some central authority then attempts to maximize utility for all applications to which it provides service.

The need for static, long-lived utilities renders a market-based approach less desirable for mobile clients; the resource situations of these clients will change over time, changing the utility of various qualities. For example, when a mobile client discovers that power is scarce, it may want to use its network connection less often; this lowers the utilities associated with high qualities of service. However, since the decision is made centrally, the service provider cannot know of this change and take it into account.

Framed as an end-to-end consideration, ultimate responsibility for coping with changes in resource levels resides with applications on the mobile client. Thus, rather than providing reservation or full centralized management of resources, Odyssey's role is only to improve efficiency, agility and fairness by insulating applications from insignificant variations in resource levels, and by providing a focal point for resource monitoring and allocation.

8.2.2 Software Feedback Systems

Another approach to coping with mobility is the use of software feedback within an application; the two most prominent examples of this are the video player based on McCanne's Receiver-driven Layered Multicast [44], and the video player based on Cen's software feedback framework [14, 31]. Such systems monitor their own performance and react to changes in it. They scale back quality, and hence resource consumption, when application performance is poor, and attempt to discover additional resources by optimistically scaling up usage from time to time.

One advantage of software feedback systems is that they do not need to explicitly correlate performance with resource consumption. For example, a feedback-based video player does not need to know what bandwidth is required for any given fidelity. However, these systems must be constructed carefully. For example, consider a video player playing a video stream; this stream is composed of tracks compressed with progressively more complicated but aggressive lossy algorithms. If the CPU becomes constrained, the video player will begin to drop frames. Switching to a lower fidelity — and conserving network bandwidth — would result in a track requiring even more CPU to decode, worsening the situation.

Further, these approaches are all based on per-application, *laissez-faire* techniques — only an individual application's performance is considered. While the focus on performance rather than resources may alleviate the problems inherent in such an approach, it is unclear how these systems would fare in the presence of concurrent competition for scarce resources.

The final difference between Odyssey's approach and software feedback is the issue of bandwidth discovery. Odyssey uses only passive estimation to discover an increase in network bandwidth. In contrast, software feedback systems scale up usage — effectively an active probe of the network. This strategy is questionable in a mobile environment, where bandwidth is likely to be scarce.

8.2.3 Similar Approaches

A number of researchers examining the problem of mobile data access have taken Odyssey's ideas as a starting point. In Prayer [11], Bharghavan uses many of the ideas originating in Odyssey: the split between policy and mechanism in application-aware adaptation, a central authority to monitor resource availability, and the basing of adaptation decisions on ranges of availability. The fundamental abstractions in Prayer are the resource ranges, called *QoS classes*, and *adaptation blocks*, which resemble critical sections.

At the beginning of an adaptation block, an application specifies the QoS classes that it is prepared to handle, along with a segment of code associated with each class and an *action* to take should the QoS class be violated within the code segment. These actions can include: *block*, which waits until the QoS class is once again satisfied; *best effort*, which ignores the QoS change; *abort*, which exits the current adaptation block; *rollback*, which starts the block from the beginning with renegotiation; or an application-defined handler. The authors claim that such adaptation blocks simplify the construction of adaptive applications compared to Odyssey. At present, the Prayer implementation is a user-level prototype; the authors have not yet presented its evaluation.

In [86], Welling also starts with the collaborative approach inherent in application-aware adaptation. The main focus of this work is on the notification subsystem, a more general form of Odyssey's upcalls. In it, different events may be delivered with different policies, and these are separate from the mechanism of event delivery. For example, a *low memory* event may be delivered to each application in series until enough memory is freed for other uses, while a *low bandwidth* event can be delivered to each application in parallel. At present, it has been used to implement a blind-optimism style of bandwidth detection. However, since there is a single entity responsible for bandwidth estimation, there is no reason why a global estimation strategy could not be used.

As in Prayer, Welling's system deals with multiple ranges of resources at a time, rather than a single range; these are set up at the beginning of an application's lifetime, though they can be changed. This takes some flexibility from the hands of the application. For example, in a highly turbulent environment, some applications may choose to add more hysteresis to their reactions, while others might choose to remain as agile as possible; having relatively static ranges unnecessarily restricts this type of higher-order adaptation.

8.3 Evaluation Methodology

The trace modulation tool suite is a novel combination of three well-known ideas: trace collection to characterize system behavior, reduction of performance observations into a simple, time-varying model, and the application-transparent use of such a model to emulate network performance. While each of these has been explored in isolation by other researchers, the author is not aware of any previous work, published or unpublished, that combines these ideas in a similar manner.

The best-known system for tracing network behavior is the Berkeley Packet Filter [45], which is typically used in conjunction with `tcpdump` [33]. This architecture is efficient, more flexible than that of trace modulation, and has rightly found great favor with the networking community. Trace modulation's collection mechanism differs from the Berkeley Packet Filter in that it records device characteristics in addition to traffic information. While not strictly necessary for trace modulation, such a record of device behavior is invaluable for a better understanding of wireless networks [51].

The notion of reducing complex network observations to simple parameters through controlled workloads is commonly used in modelling physical channels. The technique used to determine bottleneck bandwidth — two ECHO packets sent back-to-back — is quite similar to the packet-pair approach used by Keshav [38]. As in the empirical modulation workload, packet-pair also assumes a symmetric network. The key difference between trace modulation and packet-pair is that the three-packet workload used in empirical trace collection also allows derivation of additional per-packet and per-byte latency experienced by packets, in addition to those delays imposed by the bottleneck bandwidth.

The network emulation tool most similar to our modulation kernel is *hitbox*, which was used in evaluating the performance of the Vegas variant of TCP [2]. Hitbox is an in-kernel modulation layer meant to emulate long-haul network characteristics. Each hitbox host is configured with a single set latency and bandwidth parameters; when connected together in a testbed, they implicitly re-create some desired end-to-end properties. Unlike the traces used for modulation, the parameters to a hitbox host are relatively static; they could not easily be used to model performance that quickly varies. Also, since the individual per-packet delays imposed by hitbox tend to be smaller than those in trace modulation, the clock interrupts on hitbox hosts are more frequent than in the default case, causing some small perturbation in experimental results.

A considerably more flexible system, the Probe/Fault Injection Tool [17], allows any protocol or application layer to be encapsulated by layers above and below. Layers above can be used to generate a test workload, while layers below can be used to perturb existing traffic according to some model. Unlike trace modulation, these layers are driven only by synthetic models, not by empirically derived ones; however, there is no reason why such models could not be used.

The Lancaster emulator [16] uses a central server rather than an emulation layer in each host. This has the advantage that shared-media models can be implemented. However, as with Probe/Fault Injection, this emulator uses only synthetic models.

More broadly, the use of user-level libraries for network emulation is widespread. Examples include the Lancaster emulator, Delayline [30], and the `slow` mechanism of RPC2 [65]. While useful, such libraries have two shortcomings: they require recompilation or relinking of applications, and they only influence traffic to or from the instrumented applications, not the entire host.

8.4 Systems with Potential Contributions

There are two systems that have invented techniques of some use to Odyssey. One takes a much more formal approach in applying control systems to the problem of adaptation, while the other provides global network estimation rather than doing so on a per-client basis.

The first system is Cen's software feedback toolkit, mentioned above in Section 8.2.3. This toolkit applies linear control systems theory directly to the creation of software feedback systems, which themselves are inherently non-linear. The technique involves decomposing the entire range over which a system must adapt into smaller sub-ranges, within which a well-behaved, linear control system is valid. If the system crosses the boundary between two sub-ranges, the system provides a form of *meta-adaptation* that switches to a different control system that is valid for the new range. Within the linear portions, formal analysis can be brought to bear to prove certain properties of the system.

The second system is SPAND — shared passive network performance discovery — that estimates global network performance citeodyssey:seshan97. Each SPAND host records observations of the network performance achieved between it and the hosts with which it communicates. These observations are periodically sent to a *performance server* that groups reports based on domain. When a client queries the performance server for an estimate of the network quality to some target host, the performance server produces an estimate based on the experiences of other hosts within the client's domain in communicating with the target. These estimations are based on passive observation, but the process of collecting them adds some small amount of overhead to the system. This approach could be used to provide a good first estimate when the Odyssey client has no history upon which to rely.

8.5 Summary

Odyssey is the first system to provide adaptive services for diverse, concurrent applications. The core idea of this system, application-aware adaptation, compares favorably with other approaches to adaptation in this domain. This idea has also found favor with other members of the mobile systems community, who have used it as a basis for their own work.

Many previous systems provided inspiration, design, or artifacts used in the construction of Odyssey. The most significant of these is Coda, though many other systems contributed in large and small ways. Additionally, there are two systems, Cen's software feedback system and Seshan's SPAND, which could make obvious contributions to Odyssey.

Chapter 9

Conclusion

Access to data from mobile clients is becoming increasingly important. This trend is being driven by two factors: the increasing power and capability of mobile devices, and more readily available wireless networking technologies. The former is bringing low-cost, portable hardware with multimedia capability into broad use. The latter provides the means with which these small devices can access a broad wealth of shared data.

This dissertation has shown that these mobile clients must adapt to changes in their environments, and that this adaptation is best provided through application-aware adaptation — a collaboration between the system and the applications. Odyssey is the first implemented system to demonstrate the feasibility of application-aware adaptation, and the evaluation of this prototype confirms the importance of this approach to adaptation.

This chapter begins in Section 9.1 with a brief description of the contributions made in the course of demonstrating the thesis. This work has also uncovered many avenues of further inquiry; these are presented in Section 9.2. Finally, Section 9.3 concludes with the major lessons to be taken from the dissertation.

9.1 Contributions

This dissertation makes contributions in three broad areas. The first consists of the conceptual ideas underpinning the work. The second consists of the artifacts — tools and research testbeds — created in the course of the dissertation. The third set of contributions are the lessons taken from the qualitative and quantitative evaluation of these artifacts. Each of these areas is discussed in the following sections in turn.

9.1.1 Conceptual Contributions

Many researchers have recognized the need for adaptation in mobile systems. This work is the first to formalize this notion as the trading of fidelity — a type-specific notion of data quality — for resource consumption. While this forces the exposure of types within the system, access to such type knowledge can have performance advantages through the tailoring of caching policy, choice of transport protocol, and the like.

The second conceptual contribution of this dissertation is the taxonomy of adaptive systems. This taxonomy is based on the relative roles the system and its applications play in making adaptive decisions. The recognition that diverse, concurrent applications are best served by a collaboration with the system was the keystone of this work. This notion is based upon the fact that the system is best positioned to monitor the availability of resources, but the applications must decide how to react to changes in that availability.

The third conceptual contribution of this work was its evaluation strategy. Any evaluation of an adaptive system must simultaneously address both quality and performance. The best quality can always be provided by sacrificing performance, and vice versa. Since wireless networks, by their very nature, are complex and yield irreproducible results, they cannot be used to carry out such an evaluation. This work was the first to propose transient response analysis as the method by which to evaluate adaptive systems.

The architectural design of Odyssey, particularly the decomposition between viceroy and warden, forms the fourth contribution. This decomposition is primarily a matter of following software engineering principles. It allows type information to encapsulated in small code components that do not interfere with one another. There is a single interface between each of these type-aware components and the rest of the system; this interface is the same regardless of underlying type. The notion that type-specific operations need only be interpreted by wardens and applications provides for further insulation.

The final conceptual contribution is the notion of trace modulation. There are two key uses for modulation. The first is the repeatable re-creation of behavior found in a live wireless network. Because such networks provide neither repeatable performance nor good experimental control, such a re-creation is invaluable. The second use of trace modulation is in carrying out transient response analysis, which is central to this work.

9.1.2 Artifacts

This dissertation has produced three substantial artifacts. The first is the upcall mechanism added to NetBSD; a general purpose mechanism that is completely separable from Odyssey. Its use in the measurement testbed greatly simplified data collection and analysis. While its performance could be improved, it imposes no additional constraints — for example, thread models or synchronization — on applications that wish to use it.

The second artifact is the trace modulation tool suite. It has been used for both the evaluation of Odyssey as well as measuring and modelling a wireless installation [51]. The tool can be used to provide either a purely synthetic testbed, or one that closely approximates a live wireless network. Using the tool is no more difficult than traditional, wired network experiments. It can potentially become a widely-used tool in the mobile systems community.

The final, and most important, artifact produced in the course of this work is Odyssey itself. Together with the sample applications, it demonstrates the feasibility of application-aware adaptation, and provides a valuable testbed for understanding the deeper implications of adaptive systems. It forms the basis of most of the future work outlined in Section 9.2.

9.1.3 Evaluation Results

This work has contributed several important lessons from qualitative and quantitative evaluation. In modifying the applications described in Chapter 5, the changes required to make use of application-aware adaptation were minor, and confined to small areas of code. Building the wardens was more challenging, but this can be done once per data type, and leveraged across all applications using that data type. Interestingly, adding type-specific access methods in the wardens often simplified applications.

The evaluation empirically shows that passive bandwidth estimation, in the absence of reservation or admission control, can be very accurate for the time scale at which these applications adapt. In the face of a sharp change in the supply of bandwidth, the estimator takes at most a few seconds to recognize and correctly track that change. In the face of a sharp change in demand for bandwidth, the estimator can take several seconds to detect it. It also demonstrates some oscillation between competing streams; this is a result of the estimator's bias towards recent use.

The delay in detecting changes in available bandwidth is the most significant limitation on agility imposed by Odyssey. This work has shown that the remaining delays are, together, on the order of a few milliseconds. Thus, application-aware adaptation can be implemented with excellent agility.

The work has also shown that, in the face of changing network bandwidth, adaptive data access strategies are superior to static ones; they provide the best balance between performance and data quality. In those situations where there is no benefit to be gained by adaptation, an adaptive strategy correctly mimics the correct static strategy.

Finally, the dissertation empirically demonstrates the critical importance of centralized resource management in supporting concurrent, competing applications. If applications are left to their own devices in estimating available bandwidth, they will not correctly detect the interference generated by other running applications. Such unwarranted optimism leads to poor fidelity choices, and hence poor performance.

9.2 Future Work

This dissertation has opened many avenues of future work. Improvements can be made to each area of Odyssey: the system, the applications, and the evaluation testbed. Some of these are modest in scope, while others are more ambitious. This section explores each of these areas in turn, and concludes with a discussion of the broader research directions suggested by this dissertation.

9.2.1 System Extensions

The first and most pressing system-level need is to add the full complement of resources described in Section 3.3.2. The estimator for network latency is already present; round trip time is calculated in the process of estimating bandwidth. Of the remainder, disk space will be the most straightforward to implement, since Odyssey's cache space is directly under the viceroy's control. The others more challenging. The estimation of available CPU cycles must be in platform-independent units, such as SPECint95 ratings, or they will be unusable. Correctly estimating battery power depends on hardware support, such as that provided by Intel's *smart battery* [32]. Monetary costs are likely to be the most difficult to calculate and use, given the number and complexity of possible rate structures.

One feature of the current design that could be eliminated is the tight coupling between names and types. While the tome structure itself is critical to preserving administrative scalability, this coupling needlessly complicates the handling of composite or container objects — for example, a multimedia object that contains synchronized video and audio streams. Combining tomes with some other naming structure, such as X.500 [12], and explicitly supporting an aggregation type would be useful.

The third area of the system to improve is the algorithm that divides total bandwidth amongst multiple connections. The current practice of weighting based on recent use results in some undesirable effects. It might be useful to make use of the windows of tolerance in bandwidth requests to reduce these effects. However, actual use must remain a key factor in the decision, since an application may request a particular window of tolerance that cannot be met even in the absence of competing connections.

9.2.2 Application Improvements

There are improvements that could be made to each application in Chapter 5. Web objects should have some notion of degradation for elements other than images; they also form a natural basis for exploring the notion of aggregate objects. Adding fidelity to speech recognition — in the form of reduced acoustic models, vocabularies, and grammars — is an important challenge. Finally, both XAnim and the speech recognition system are sensitive to computational power as well as network bandwidth. Extending them to use this resource in making adaptation decisions would be of great benefit.

There are also new applications that would benefit from an adaptive approach. The most important of these from the standpoint of mobile systems is a mapping application. Maps are typically represented as collections of vectors and points that are rendered into images. The sizes of these data structures are large for even moderately complicated maps, and rendering them is computationally expensive. The dimensions of fidelity for map objects include minimum feature size, feature set, and granularity of individual features. Changing any of these can reduce both the representation size and rendering costs.

9.2.3 Trace Modulation Improvements

The trace modulation testbed is an extremely promising tool, but could be made even stronger. The first, and most important, improvement would be to extend it to cope with asymmetric networks. In trace replay, changing from one set of parameters to two for each time slice would enable the re-creation of asymmetric performance. However, empirically capturing such performance would require fine-grained, synchronized clocks.

More practically, the tool itself can be improved and simplified. Rather than having an ad hoc trace collection mechanism, it should be relatively easy to make use of the Berkeley Packet Filter [45] to make collection more flexible. It should also be possible to re-implement the trace replay mechanism as a piece of code that can be dynamically loaded into the kernel, further simplifying its use.

Trace modulation is primarily meant to emulate a wireless network from the perspective of a single host. While this host may experience some form of interference encoded in the replay trace, it is not possible to have the actual workloads from two replay hosts interfere in a meaningful way. Extending trace modulation techniques to multiple hosts using a shared medium is of obvious value. This problem has been addressed by the Lancaster emulator [16], which is a library implementation that must be linked with specific applications, providing delays based on purely synthetic models. However, no host-wide solution that uses empirically-driven models yet exists.

9.2.4 New Directions

Finally, this dissertation has opened the door to four broader research topics, for which Odyssey can serve as a starting point. First, how can users be informed of and involved in adaptation decisions? Second, how can one modify a binary-only application to take advantage of adaptation? Third, how can the ad hoc decomposition of functionality in the speech recognition application be generalized? Fourth, how can the construction of adaptive systems be brought from an ad hoc activity to a principled, engineering discipline?

Studies have shown that variability in response time is an annoyance to users that can make tasks more difficult [71]. It is not difficult to imagine that large, frequent variations in quality would also be troubling. Thus, while very agile applications obtain the best possible quality within performance bounds, such instability is not clearly preferable. Furthermore, consider the case of two applications competing for scarce bandwidth. It may be possible to satisfy a higher fidelity for either if it is favored, but not if the two are treated equally. Making such a decision must involve user input.

Netscape presented a compelling challenge to application-aware adaptation because its source was not available. However, in Netscape's case, the task was greatly simplified by the presence of a proxy interface. The general problem is much harder, though could be solved through binary rewriting, system call interception, or similar techniques.

The speech recognition application suggests the importance of being able to dynamically decide whether to ship data or computation. This capability is currently provided in an ad hoc manner by the speech warden. Extending Odyssey to provide full support for dynamically deciding between function and data shipping would enable us to more thoroughly explore this tradeoff in mobile computing.

One compelling use of the ability to decide between function and data shipping is the searching of distributed repositories. Search tasks exhibit little or no temporal locality, rendering caching ineffective in compensating for a lack of bandwidth. One could address this problem by combining the power of *dynamic sets* [75] with dynamic function vs. data shipping decisions.

Finally, the current state of the art in building adaptive software systems is a black art. Developing systematic principles for their design, as well as techniques for analyzing their agility and stability before they are built, would be valuable. Cen's work [14] is a promising start along these lines, but much remains to be done.

9.3 Concluding Remarks

Ever more capable mobile hardware and wireless networking services combine to make the problem of mobile data access increasingly important. However, properties intrinsic to mobile environments force clients to adapt their behavior by trading data quality for resource consumption.

In addressing the problem of adaptation in the context of Odyssey, a few underlying themes stand out. The keystone of this work — the collaboration between the system and its applications — is a direct application of the *end-to-end argument* to the problem of adaptation. Functionality is placed at the outermost layer possible; the applications must have a say over policy, but cannot accurately measure their environment without system support. Knowledge of object types is placed in the system in order to improve caching and transport decisions.

The second major theme is *simplicity*. Odyssey is designed to be a minimal set of extensions to a common operating system. The API presented to applications offers very few new abstractions, and presents no undue burdens on applications. Rather than inject active probes into the network, the system relies only on passive estimation. The user-level implementation greatly simplified construction, at a modest performance cost.

The final major theme is the importance of careful evaluation. Subjecting Odyssey to controlled, simple, repeatable bandwidth scenarios enabled tractable, convincing analysis. Without such a strategy, very few quantitative conclusions can be reached.

While Odyssey makes a convincing case for application-aware adaptation, it is only a starting point in addressing the general problem of mobile data access. Users must become actively involved in adaptation decisions, and the notion of adaptation must be extended to computational processes as well as data structures. The very process of building adaptive systems is in its infancy. However, Odyssey should serve well as a base from which to explore these problems.

Bibliography

[1] T. F. Abdelzaher, E. M. Atkins, and K. Shin. QoS Negotiation in Real-Time Systems and its Application to Automated Flight Control. In *IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997.

[2] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: Emulation and Experiment. *Computer Communication Review*, 25(4):185–195, August 1995.

[3] D. K. Anand. *Introduction to Control Systems*, volume 8 of *International Series on Systems and Control*. Pergamon Press, Oxford OX3 0BW, England, 1984.

[4] Apple Computer, Inc. *Inside Macintosh: QuickTime*. Addison-Wesley Publishing Company, 1993.

[5] Apple Computer Inc., Cupertino, CA. *Newton Message Pad Handbook*, 1993.

[6] AT&T Global Information Solutions Company. Architecture Specification for Wave-LAN Air Interface.

[7] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The Effects of Asymmetry on TCP Performance. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Budapest, Hungary, September 1997.

[8] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the USENIX 1997 Annual Technical Conference*, Anaheim, CA, January 1997.

[9] J. F. Bartlett. W4 — The Wireless World Wide Web. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[10] B. J. Bennington and C. R. Bartel. Wireless Andrew: Experience Building a High Speed, Campus-Wide Wireless Data Network. In *Proceedings of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Budapest, Hungary, September 1997.

[11] V. Bharghavan and V. Gupta. A Framework for Application Adaptation in Moblie Computing Environments. In *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference*, Bethesda, MD, August 1997.

[12] S. Brabner. X.500: a Global Directory Standard. *Telecommunications (International Edition)*, 23(2), Februrary 1989.

[13] A. Campbell, R.-F. Liao, and Y. Shobatake. QoS Controlled Handoff in Wireless ATM Networks. In *Sixth WINLAB Workshop on Third Generation Wireless Information Networks*, New Brunswick, NJ, March 20-21 1997.

[14] S. Cen. *A Software Feedback Toolkit and its Application In Adaptive Multimedia Systems*. PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, October 1997.

[15] D. D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, December 1985.

[16] N. Davies, G. S. Blair, K. Cheverst, and A. Friday. A Network Emulator to Support the Development of Adaptive Applications. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location Independent Computing*, April 10-11 1995.

[17] S. Dawson and F. Jahanian. Probing and Fault Injection of Dependable Distributed Protocols. *The Computer Jouranl*, 38(4), 1995.

[18] R. P. Draves. A Revised IPC Interface. In *Proceedings of the USENIX Mach Conference*, October 1990.

[19] D. Duchamp. Issues in Wireless Mobile Computing. In *Proceedings of the Third Workshop on Workstation Operating Systems*, Key Biscayne, FL, April 1992.

[20] M.R. Ebling and M. Satyanarayanan. SynRGen: An Extensible File Reference Generator. In *Proceedings of the 1994 ACM SIGMETRICS Conference*, Nashville, TN, May 1994.

[21] D. Eckhardt and P. Steenkiste. Measurement and Analysis of the Error Characteristics of an In-Building Wireless Network. *Computer Communication Review*, 26(4):243–254, October 1996.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. Internet RFC 2068, January 1997.

[23] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4), April 1994.

[24] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the Seventh International ACM Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, October 1996.

[25] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole Jr. Semantic File Systems. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, 1991.

[26] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, 1993.

[27] J. S. Heidemann and G. J. Popek. File-system Development with Stackable Layers. *ACM Transactions on Computer Systems*, 12(1), 1994.

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

[29] L. Huston and P. Honeyman. Partially Connected Operation. In *Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing*, Ann Arbor, MI, April 1995.

[30] D. B. Ingham and G. D. Parrington. Delayline: A Wide-Area Network Emulation Tool. *Computing Systems*, 7(3), 1994.

[31] J. Inouye, S. Cen, C. Pu, and J. Walpole. System Support for Mobile Multimedia Applications. In *Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, St. Louis, MO, May 1997.

[32] Intel Corporation, Hillsboro, OR. *Smart Battery Specifications*, 1994. Co-produced by Duracell, Inc., Bethel, CT.

[33] V. Jacobson, C. Leres, and S. McCanne. *The Tcpdump Manual Page*. Lawrence Berkeley Laboratory, Berkeley, CA.

[34] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, Copper Mountain, CO, December 1995.

[35] ISO/IEC JTC1/SC29/WG11. Coding of Moving Pictures and Associated Audio for Digital Storage Media up to 1.5 Mbit/s. MPEG, International Standard, ISO 11172.

[36] M. F. Kaashoek, T. Pinckney, and J. A. Tauber. Dynamic Documents: Mobile Wireless Access to the WWW. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.

[37] R. H. Katz and E. A. Brewer. The Case for Wireless Overlay Networks. In *SPIE Multimedia and Networking Conference*, January 1996.

[38] S. Keshav. Packet-Pair Flow Control. To appear in IEEE/ACM Transactions on Networking.

[39] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[40] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Summer Usenix Conference Proceedings*, Atlanta, GA, 1986.

[41] T. V. Lakshman, U. Madhow, and B. Suter. Window-based error recovery and flow control with a slow acknoledgement channel: a study of TCP/IP performance. In *Proceedings of the IEEE INFOCOM '97 Conference on Computer Communications*, Kobe, Japan, April 1997.

[42] S. Lu, K.-W. Lee, and V. Bharghavan. Adaptive Service in Mobile Computing Environments. In *Fifth IFIP International Workshop on Quality of Service*, New York, NY, May 1997.

[43] A. Luotonen and K. Altis. World-Wide Web Proxies. *Computer Networks and ISDN Systems*, 27, September 1994.

[44] S. McCanne, V. Jacobsen, and M. Vetterli. Receiver-driven Layered Multicast. In *Proceedings of the ACM SIGCOMM'96 Conference*, Stanford, CA, August 1996.

[45] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Deigo, CA, January 1993.

[46] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[47] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[48] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM'97 Conference*, Cannes, France, September 1997.

[49] L. B. Mummert. *Exploiting Weak Connectivity in a Distributed File System*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1993. CMU-CS-96-195.

[50] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th Symposium on Operating System Prinicples,* Copper Mountain, CO, December 1995.

[51] G. Nguyen, R. Katz, B. Noble, and M. Satyanarayanan. A Trace-Based Approach for Modelling Wireless Channel Behavior. In *Proceedings of the Winter Simulation Conference,* 1996.

[52] B. Noble, G. Nguyen, M. Satyanarayanan, and R. Katz. Mobile Network Tracing. Internet RFC 2041, October 1996.

[53] B. D. Noble, M. Price, and M. Satyanarayanan. A Programming Interface for Application-Aware Adaptation in Mobile Computing. In *Proceedings for the Second USENIX Symposium on Mobile and Location-Independent Computing*, Ann Arbor, Michigan, April 1995. Also as technical report CMU-CS-95-119, School of Computer Science, Carnegie Mellon University.

[54] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-Based Mobile Network Emulation. In *Proceedings of the 1997 ACM SIGCOMM Conference,* Canne, France, September 1997.

[55] J. K. Ousterhout. Why Threads are a Bad Idea (for most purposes). Invited talk, 1996 USENIX Technical Conference, January 25, 1996. As of this writing, slides for this talk are available at http://www.sunlabs.com/ ouster/.

[56] S. K. Park and K. W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the Association for Computing Machinery,* 31, October 1988.

[57] M. Podlipec. The XAnim Home Page. At the time of this writing, the document is available only through the World Wide Web at http://xanim.va.pubnix.com/home.html.

[58] S. Pope and P. Webster. QoS Support for Mobile Computing. In *Fifth IFIP International Workshop on Quality of Service*, New York, NY, May 1997.

[59] G. J. Popek, R. G. Guy, T. W. Page, and J. S. Heidemann. Replication in Ficus Distributed File Systems. In *Proceedings of the Workshop on Management of Replicated Data,* Houston, TX, November 1990.

[60] Qualcomm Inc., San Diego, CA. *Eudora Macintosh User Manual*, 1993.

[61] L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall Signal Processing Series. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1993. A. V. Oppenheim, Series Editor.

[62] D. Raggett. HTML 3.2 Reference Specification. Recommendation REC-html32, World Wide Web Consortium, January 1997.

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

[64] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Usenix Conference Proceedings*, Summer 1985.

[65] M. Satyanarayanan. *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon University, October 1991.

[66] M. Satyanarayanan. Mobile computing: Past, present, and future. In *Proceedings of the IBM Workshop on Mobile Computing*, Austin, TX, January 1994.

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

[68] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1):33–57, February 1994. Corrigendum: May 1994, Vol. 12, No. 2, pp. 165-172.

[69] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price. Application-aware Adaptation for Mobile Computing. In *Proceedings of the 6th ACM SIGOPS European Workshop*, Dagstuhl, Germany, September 1994. Also as technical report CMU-CS-95-183, School of Computer Science, Carnegie Mellon University, and in *Operating Systems Review*, 29 (1), January 1995.

[70] S. Sheng, A. Chandrakasan, and R. Broderson. A Portable Multimedia Terminal. *IEEE Communications Magazine*, 30(12):64–76, December 1992.

[71] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 1998. third edition.

[72] R. Sidebotham. Volumes: The Andrew File System Data Structuring Primitive. In *European Unix User Group Conference Proceedings*, August 1986. Also available as Tech. Rep. CMU-ITC-053, Carnegie Mellon University, Information Technology Center.

[73] A. Smailagic and D. P. Siewiorek. Modalities of Interaction with CMU Wearable Computers. *IEEE Personal Communications*, 3(1), February 1996.

[74] D. C. Steere. *Using Dynamic Sets to Reduce the Aggregate Latency of Data Access*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1996.

[75] D. C. Steere. Exploiting Non-Determinism in Set Iterators to Reduce I/O Latency. In *Proceedings of the 16th ACM Symposium on Operating System Principles,* St. Malo, France, October 1997.

[76] D. C. Steere, J. J. Kistler, and M. Satyanarayanan. Efficient User-Level File Cache Management on the Sun Vnode Interface. In *Summer Usenix Conference Proceedings,* Anaheim, CA, June 1990.

[77] D. B. Terry, M. M. Theimer, K. Petersen, and A. J. Demers. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating System Principles,* Copper Mountain, CO, December 1995.

[78] M. Theimer, A. Demers, and B. Welch. Operating System Issues for PDAs. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*. IEEE, October 1993.

[79] T. E. Truman and R. W. Broderson. A Measurement-based Characterization of the Time Variation of an Indoor Wireless Channel. In *IEEE International Conference on Universal Personal Communications,* San Diego, CA, October 1997.

[80] U.S. Robotics, Inc., Los Altos, CA. *Pilot Handbook*, 1996. Part Number 423-0001.

[81] G. M. Voelker and B. N. Bershad. Mobisaic: An Information System for a Mobile Wireless Computing Environment. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications,* Santa Cruz, CA, December 1994.

[82] A. Waibel. Interactive Translation of Conversational Speech. *IEEE Computer*, 29(7), July 1996.

[83] G. K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4), April 1991.

[84] T. Watson. Effective Wireless Communication through Application Partitioning. In *Proceedings of the Fifth IEEE Hot Topics in Operating Systems Workshop,* Orcas Island, WA, May 1995.

[85] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communictions of the ACM*, 36(7):75–84, July 1993.

[86] G. Welling and B. R. Badrinath. A Framework for Environment Aware Mobile Applications. In *Proceedings of the 14th International Conference on Distributed Computing Systems,* Baltimore, MD, May 1997.