Mobile Information Access

M. Satyanarayanan

January 1996
CMU-CS-96-107

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in
IEEE Personal Communications,
Volume 3, No. 1, February 1996

Abstract

The ability to access information on demand when mobile will be a critical capability in the 21st century. In this paper, we examine the fundamental forces at work in mobile computing systems and explain how they constrain the problem of mobile information access. From these constraints, we derive the importance of *adaptivity* as a crucial requirement of mobile clients. We then develop a taxonomy of adaptation strategies, and summarize our research in *application-transparent* and *application-aware* adaptation in the *Coda* and *Odyssey* systems respectively.

This research was supported by the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under contract number F196828-93-C-0193. Additional support was provided by the IBM Corp., Digital Equipment Corp., Intel Corp., Xerox Corp., and AT&T Corp. The U.S. government is authorized to reproduce and distribute reprints for government purposes, notwithstanding any copyright notation thereon.

Keywords: Mobile computing, adaptivity, application-transparent adaptation, application-aware adaptation, laissez-faire strategies, distributed file systems, databases, multimedia data, weak connectivity, intermittence, low-bandwidth networks, Coda, Odyssey, disconnected operation, trickle reintegration, user-assisted cache management, patience models, IOT, isolation-only transactions, dynamic sets, mobile search

Mobile Information Access

M. Satyanarayanan
School of Computer Science
Carnegie Mellon University

1. Introduction

The ability to access information on demand at any location confers competitive advantage on individuals in an increasingly mobile world. As users become more dependent on this ability, the span of access of data repositories will have to grow. The increasing social acceptance of the home or any other location as a place of work is a further impetus to the development of mechanisms for mobile information access.

These considerations imply that data from shared file systems, relational databases, object-oriented databases and other repositories must be accessible to programs running on mobile computers. For example, a technician servicing a jet engine on a parked aircraft needs access to engineering details of that model of engine as well as past repair records of that specific engine. Similarly, a businessman who is continuing his work on the train home from Manhattan needs access to his business records. Yet another example involves emergency medical response to a case of poisoning: the responding personnel will need rapid access to medical databases describing poison symptoms and antidotes, as well as access to the specific patient's medical records to determine drug sensitivity.

This paper is a status report on our work toward meeting such challenges. We begin by describing a scenario that offers a tantalizing glimpse of the power of mobile information access. We then examine the major obstacles on the path toward this vision. The rest of the paper is a summary of our research on overcoming these obstacles in the context of the *Coda* and *Odyssey* systems.

2. A Vision of Tomorrow

Imagine this hypothetical scenario of a business trip in the year 2000:

> *You are sitting at your office desk, editing a report stored in a shared file system. The machine you are using is a small notebook, but it lets you use the larger and more comfortable display and keyboard on your desk via a tabletop infrared link. Soon it is time to leave for the airport.*
>
> *When the limo arrives, you pick up your notebook and leave. On the ride to the airport you continue your work. Your notebook recognizes that it is no longer on a LAN, but continues communication with the servers via a cellular modem. You finish your editing, save the file, and send mail to your coauthor letting him know that he can now review your edits. You then begin working on the slides for your talk in Paris. Upon arrival at the airport, you board your transatlantic flight and continue working. Although each seat is provided with an outlet for air-to-ground telephone service, your notebook inquires and discovers that telephone charges are very high. It therefore wisely decides to let you operate disconnected and to defer all communication until you have landed.*
>
> *When you arrive in your Paris hotel room, your notebook discovers that the hotel's late-night telephone charges are low, and that there is a HDTV set in your room. It therefore propagates the changes you have made so far, fetches new versions of some of the files you had cached, picks up your mail, and uses the HDTV set as your display. You work late into the night, putting the finishing touches on your slides. The next morning, you present your talk. Your notebook senses the presence of a large wall-sized display in the conference room, and shows your slides on it. Since your talk is about a new piece of user-interface software, you are able to give a live demo of it using the notebook.*
>
> *Once your business is complete, you decide to play tourist for a day before returning home. The concierge at your hotel subscribes you to an excellent guided walking tour, and rents you a heads-up display and headphones. Setting out with your notebook in your backpack, you pick a route from the map displayed. As you walk, you indicate items of interest on the map. A short video describing the unique historical and architectural features of the site is seen, and the accompanying audio commentary is heard. As you pass through a major shopping district, advertisements of sales (translated by your notebook into English) pop up on your display. One of these interests you, and you walk into the store and purchase a gift. The store clerk obtains your travel itinerary from your notebook and arranges for your duty-free purchase to be delivered to the correct gate for your flight home tomorrow.*

You continue on your walking tour for many more hours. Exhausted, you decide to take the metro back to your hotel. On the metro, you watch CNN on your notebook. From time to time, as the train goes through regions of poor reception, the displayed image degenerates from full-motion color to slow-scan black-and-white.

The next morning, you head for the airport, pick up your gift at the gate, and board the flight home. You can relax and watch the movie: your notebook has been recording your purchases and is now automatically preparing an expense report. When you reach home, it will transmit the report to your secretary for reimbursement.

3. Adaptation: the Key to Mobile Information Access

What makes this scenario fiction rather than reality today? Not the absence of proper hardware, since most of the hardware technologies needed are close at hand. What is missing is the software support. Developing this software is the goal of our research.

3.1. Constraints of Mobility

Our goal is made challenging by four fundamental constraints of mobility:

- *Mobile elements are resource-poor relative to static elements.*
  At any given cost and level of technology, considerations of weight, power, size and ergonomics will exact a penalty in computational resources such as processor speed, memory size, and disk capacity. While mobile elements will undoubtedly improve in absolute ability, they will always be resource-poor relative to static elements.

- *Mobility is inherently hazardous.*
  A Wall Street stockbroker is more likely to be mugged on the streets of Manhattan and have his or her laptop stolen than to have the workstation in a locked office be physically subverted. Even if security isn't a problem, portable computers are more vulnerable to loss or damage.

- *Mobile connectivity is highly variable in performance and reliability.*
  Inside some buildings, a mobile element may have reliable, high-speed wireless LAN connectivity. But in other buildings, it may only have modem or ISDN connectivity. Outdoors, it may have to rely on a low-bandwidth wireless WAN with gaps in coverage.

- *Mobile elements rely on a finite energy source.*
  While battery technology will undoubtedly improve over time, the need to be sensitive to power consumption will not diminish. Concern for power consumption must span many levels of hardware and software to be fully effective.

These constraints are not just artifacts of current technology, but are intrinsic to mobility. Together, they complicate the design of mobile information systems and require us to rethink traditional approaches to information access. In addition, *scalability* will be a growing concern because of the ubiquity of mobile computers. *Diversity of data* will be another key concern because the data repositories of tomorrow will be much richer in content than traditional file systems or databases.

3.2. The Need for Adaptation

Mobility exacerbates the tension between autonomy and interdependence that is characteristic of all distributed systems. To function successfully, mobile elements must be *adaptive*. The relative resource poverty of mobile elements as well as their lower trust and robustness argues for reliance on static servers. But the need to cope with unreliable and low-performance networks, as well as the need to be sensitive to power consumption argues for self-reliance.

Any viable approach to mobile computing must strike a balance between these competing concerns. This balance cannot be a static one; as the circumstances of a mobile client change, it must react and dynamically reassign the responsibilities of client and server.

3.3. Taxonomy of Adaptation Strategies

The range of strategies for adaptation is delimited by two extremes, as shown in Figure 1. At one extreme, adaptation is entirely the responsibility of individual applications. While this *laissez-faire* approach avoids the need for system support, it lacks a central arbitrator to resolve incompatible resource demands of different applications and to enforce limits on resource usage. It also makes applications more difficult to write, and fails to amortize the development cost of support for adaptation.

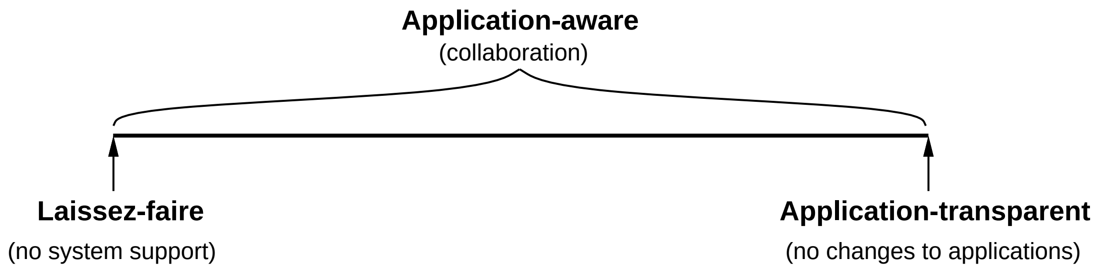


Figure 1: Range of Adaptation Strategies

At the other extreme, referred to as *application-transparent adaptation,* the responsibility for adaptation is borne entirely by the system. This approach is attractive because it is backward compatible with existing applications: they continue to work when mobile without any modifications. The system provides the focal point for resource arbitration and control. The drawback of this approach is that there may situations where the adaptation performed by the system is inadequate or even counter-productive for some applications.

Between these two extremes lies a spectrum of possibilities that we collectively refer to as *application-aware adaptation.* By supporting a collaborative partnership between applications and the system, this approach permits individual applications to determine how best to adapt, but preserves the ability of the system to monitor resources and to enforce allocation decisions.

We have been exploring application-transparent adaptation since about 1990. Our research vehicle has been the *Coda File System,* a descendant of AFS [4]. More recently, we have begun exploration of application-aware adaptation in *Odyssey,* a platform for mobile computing.

4. Coda: Application-Transparent Adaptation

Coda is an experimental file system whose goal is to offer clients continued access to data in the face of server and network failures [16]. It inherits many of the usage and design assumptions of its ancestor, AFS. Clients view Coda as a single, location-transparent shared Unix file system. The Coda namespace is mapped to individual file servers at the granularity of subtrees called *volumes.* At each client, a cache manager, *Venus,* dynamically obtains and caches data as well as volume mappings.

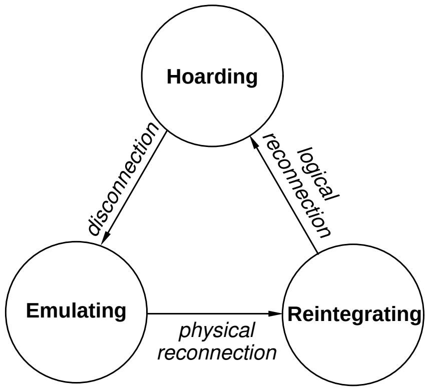
4.1. Disconnected Operation

Disconnected operation, a concept first conceived and demonstrated in Coda, is an important initial step in mobile computing [6, 7, 17]. In this mode of operation, a client continues to have read and write access to data in its cache during temporary network outages. Transparency is preserved from the viewpoint of applications because the system bears the responsibilities of propagating modifications and detecting update conflicts when connectivity is restored.

The ability to operate disconnected can be useful even when connectivity is available. For example, disconnected operation can extend battery life by avoiding wireless transmission and reception. It can reduce communication expense, an important consideration when rates are high. It allows radio silence to be maintained, a vital capability in military applications. And, of course, it is a viable fallback position when network characteristics degrade beyond usability.

4.1.1. Cache Management

To support disconnected operation, Venus operates in one of three states: *hoarding, emulating,* and *reintegrating,* as show in Figure 2. Venus is normally in the hoarding state, relying on servers but always on the alert for possible disconnection. The hoarding state is so named because a key responsibility of Venus in this state is to ensure that critical objects are cached at the moment of disconnection. Upon disconnection, Venus enters the emulating state and remains there for the duration of disconnection. Upon reconnection, Venus enters the reintegrating state, resynchronizes its cache with servers, and then reverts to the hoarding state.



While connected, Venus is in the hoarding state. Upon disconnection, it enters the emulating state and stays there until successful reconnection to a server. It then transits temporarily to the reintegrating state, and thence to the hoarding state, where it resumes connected operation.

Figure 2: Venus State and Transitions for Disconnected Operation

While disconnected, Venus services file system requests by relying solely on the contents of its cache. Since cache misses cannot be serviced or masked, they appear as failures to application programs and users. The persistence of changes made while disconnected is achieved via an operation log, called the *CML,* implemented on top of a transactional facility called *RVM* [18, 19].

Venus implements a number of optimizations to reduce the size of the CML. Before a log record is appended to the CML, Venus checks if it cancels or overrides the effect of earlier records. For example, consider the `create` of a file, followed by a `store`. If they are followed by an `unlink`, all three CML records and the data associated with the `store` can be eliminated. Both trace-driven simulations and measurements of Coda in actual use confirm the effectiveness of log optimizations [14, 17].

Venus combines implicit and explicit sources of information into a priority-based cache management algorithm. The implicit information consists of recent reference history, as in LRU caching algorithms. Explicit information takes the form of a per-client *hoard database (HDB),* whose entries are pathnames identifying objects of interest to the user at that client. A simple front-end program called `hoard` allows a user to update the HDB directly or via command scripts called *hoard profiles.* Venus periodically reevaluates which objects merit retention in the cache via a process known as *hoard walking.*

4.1.2. Conflict Detection and Resolution

Coda addresses the problem of concurrent partitioned updates using an optimistic replica control strategy. This offers the highest degree of availability, since data can be updated in any network partition. Upon reintegration, the system ensures detection of conflicting updates and provides mechanisms to help users recover from these situations.

Coda uses different strategies for handling concurrent updates on directories and files [9]. For directories, Venus possesses enough semantic knowledge to attempt transparent *resolution* of conflicts. Resolution fails only if a newly created name collides with an existing name, if an object updated at the client or the server has been deleted by the other, or if directory attributes have been modified at the server and the client [8].

Since Unix treats files as uninterpreted byte strems, Coda does not possess sufficient semantic knowledge to resolve file conflicts. Rather, it offers a mechanism for installing and transparently invoking *application-specific resolvers (ASRs)* [10]. An ASR is a program that encapsulates the detailed, application-specific knowledge necessary to distinguish genuine inconsistencies from reconcilable differences. Appointment calendars, electronic checkbooks, and project diaries are examples of applications where an application-specific approach to conflict resolution can have high payoff. If an ASR is unsuccessful, the inconsistency is exposed to the user for manual repair.

When the manual repair tool is run on a client, Venus presents the illusion of an in-place "explosion" of inconsistent objects into their distinct versions. Since inconsistencies appear as read-only subtrees in the existing name space, Unix utilities such as `diff` and `grep` can be used to construct appropriate replacements for the inconsistent objects. Upon completion of repair, the exploded subtrees are collapsed, thus reverting to a normal name space.

4.2. Weakly-Connected Operation

Weak connectivity, in the form of intermittent, low-bandwidth, or expensive networks is a fact of life in mobile computing. Disconnected operation can be viewed as the extreme case of weakly-connected operation — the mobile client is effectively using a network of zero bandwidth and infinite latency. But although disconnected operation is viable, it is not a panacea. A disconnected client suffers from many limitations:

- Updates are not visible to other clients.
- Cache misses may impede progress.
- Updates are at risk due to theft, loss or damage.
- Update conflicts become more likely.
- Exhaustion of cache space is a concern.

We have implemented a series of modifications to Coda that alleviate these limitations by exploiting weak connectivity [13]. Our modifications span a number of areas. At the lowest level, the transport protocol has been extended to be robust, efficient and adaptive over a wide range of network bandwidths. Modifications at the higher levels include those needed for rapid cache validation after an intermittent failure, for background propagation of updates over a slow network, and for user-assisted servicing of cache misses when weakly connected.

4.2.1. Rapid Cache Validation

Coda's original technique for cache coherence while connected was based on *callbacks* [4, 16]. When a client is disconnected, it can no longer rely on callbacks. Upon reconnection, it must validate all cached objects before use to detect updates at the server. Unfortunately, the time for this validation can be substantial on a slow network.

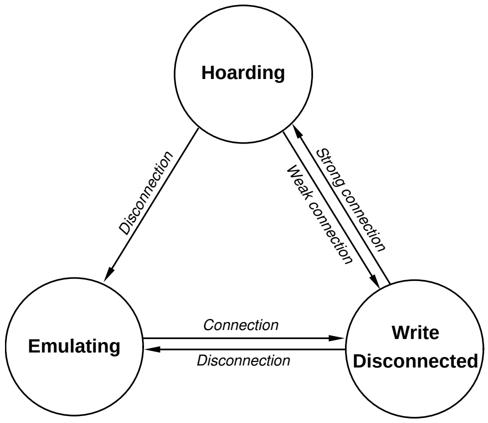
Our solution allows clients to track server state at multiple levels of *granularity.* A server now maintains version stamps for each of its volumes, in addition to stamps on individual objects. When an object is updated, the server increments the version stamp of the object and that of its containing volume. Clients cache volume version stamps in anticipation of disconnection.

When connectivity is restored after a network failure, the client presents volume stamps for validation. If a volume stamp is still valid, so is every object cached from that volume. If a volume stamp is not valid, cached objects from the volume must be validated individually. Even in this case, performance is no worse than in the original scheme. Controlled experiments as well as measurements from Coda in actual use confirm that this approach dramatically improves the speed of cache validation.

4.2.2. Trickle Reintegration

Trickle reintegration is a mechanism that propagates updates to servers asynchronously, while minimally impacting foreground activity. Supporting trickle reintegration required major modifications to the structure of Venus. As depicted in Figure 2, reintegration was originally a transient state through which Venus passed *en route* to the hoarding state. Since reintegration is now an ongoing background process, the transient state has been replaced by a stable one called the *write disconnected* state. Figure 3 shows the new states of Venus and the main transitions between them.

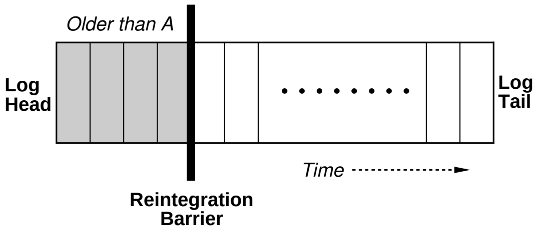
Trickle reintegration reduces the effectiveness of log optimizations, because records are propagated to the server earlier than when disconnected. Thus they have less opportunity to be eliminated at the client. A good design must balance two factors. On the one hand, records should spend enough time in the CML for optimizations to be effective. On the other hand, updates should be propagated to servers with reasonable promptness. Our solution,



This figure shows the states of Venus, as modified to handle weak connectivity. The state labelled "Write Disconnected" replaces the reintegrating state in Figure 2. In this state, Venus relies on trickle reintegration to propagate changes to servers. The transition from the emulating to the write disconnected state occurs on any connection, regardless of strength. All outstanding updates are reintegrated before the transition to the hoarding state occurs.

Figure 3: Venus States and Transitions for Exploiting Weak Connectivity

illustrated in Figure 4, uses a simple technique based on *aging.* A record is not eligible for reintegration until it has spent a minimal amount of time in the CML. This amount of time, called the *aging window*, (A), establishes a limit on the effectiveness of log optimizations. Based on the results of trace-driven simulations, we have set the default value of A to 10 minutes.



This figure depicts a typical CML scenario while weakly connected. A is the aging window. The shaded records in this figure are being reintegrated. They are protected from concurrent activity at the client by the reintegration barrier. For `store` records, the corresponding file data is locked; if contention occurs later, a shadow copy is created and the lock released.

Figure 4: CML During Trickle Reintegration

At the beginning of reintegration, a logical divider called the *reintegration barrier* is placed in the CML. During reintegration, which may take a while on a slow network, the portion of the CML to the left of the reintegration barrier is frozen. Only records to the right are examined for optimization. If reintegration is successful, the barrier and all records to its left are removed. If a network or server failure causes reintegration to be aborted, the barrier as well as any records rendered superfluous by new updates are removed.

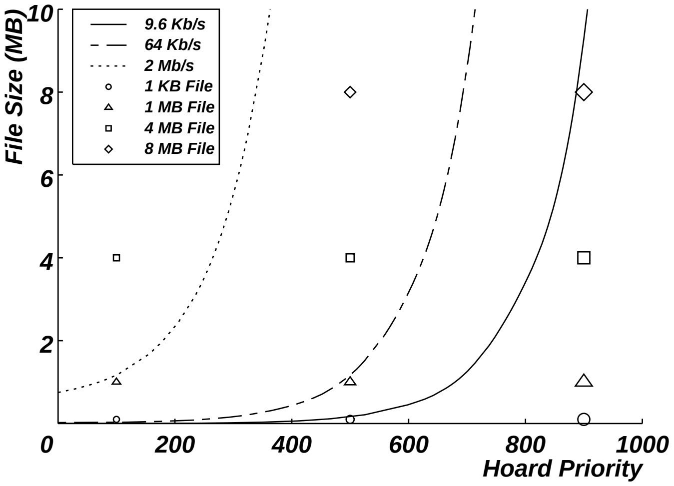
Reintegrating all records older than A in one *chunk* could saturate a slow network for an extended period. The performance of a concurrent high priority network event, such as the servicing of a cache miss, could then be severely degraded. To avoid this problem, we have made reintegration chunk size adaptive, thus bounding the duration of degradation. If a file is very large, we transfer it as a series of *fragments,* each smaller than the currently acceptable chunk size. If a failure occurs, file transfer is resumed after the last successful fragment.

4.2.3. User-Assisted Cache Miss Handling

When weakly connected, the performance impact of cache misses is often too large to ignore. In many cases, a user would rather be told that a large file is missing than be forced to wait for it to be fetched over a weak connection. But there are also situations where a file is so critical that a user is willing to suffer considerable delay. We refer to the maximum time that a user is willing to wait for a particular file as his *patience threshold* for that file.

Coda incorporates a *user patience model* to provide adaptivity in cache miss handling. This model helps maintain usability at all bandwidths by balancing two factors that intrude upon transparency. At very low bandwidths, the delays in fetching large files annoy users more than the need for interaction. As bandwidth rises, delays shrink and interaction becomes more annoying. To preserve usability, Coda handles more cases transparently. In the limit, at strong connectivity, cache misses are fully transparent.

Our initial user patience model is logarithmic, based on the conjecture that patience is similar to other human processes such as vision. Figure 5 illustrates this model. Rather than expressing the patience threshold in terms of seconds, we have converted it into the size of the largest file that can be fetched in that time at a given bandwidth. If the estimated cache miss service time for a file is below its patience threshold, Venus services the miss transparently; otherwise Venus reports the miss by returning an error. At any time, users can examine the history of recent cache misses and augment the hoard database appropriately. They can also interactively control the files fetched in hoard walks.



Each curve in this graph expresses patience threshold, (τ), in terms of file size. Superimposed on these curves are points representing files of various sizes hoarded at priorities 100, 500, and 900. At 9.6 Kb/s, only the files at priority 900 and the 1KB file at priority 500 are below τ. At 64 Kb/s, the 1MB file at priority 500 is also below τ. At 2Mb/s, all files except the 4MB and 8MB files at priority 100 are below τ.

Figure 5: Patience Threshold versus Hoard Priority

4.3. Isolation-Only Transactions

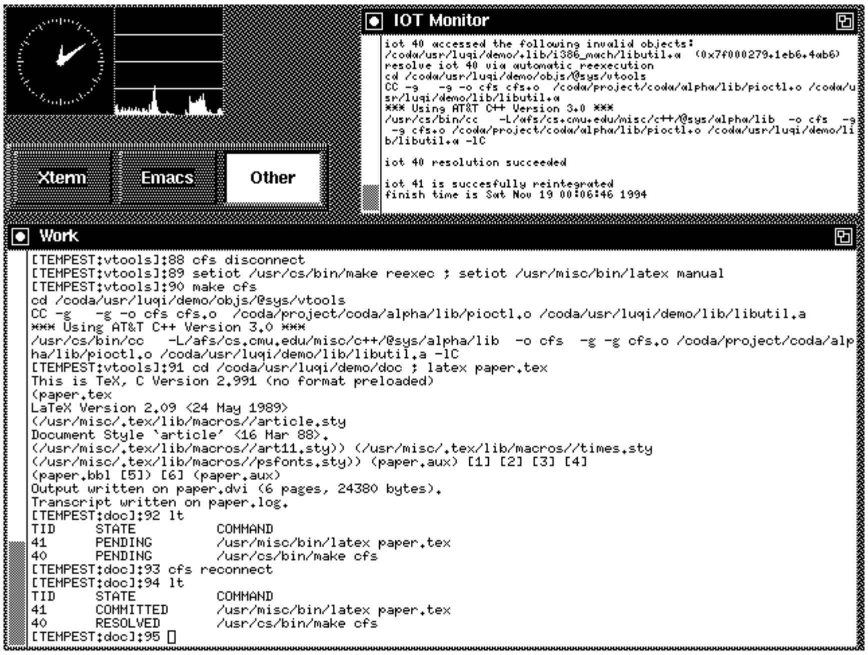
Coda's emulation of the Unix file system model has the benefit of compatibility with existing applications. Unfortunately, the Unix model is weak in terms of consistency support for concurrent file accesses. In particular, Unix has no notion of *read-write file conflicts.* This deficiency becomes especially acute in mobile computing, because extended periods of disconnected or weakly-connected operation may increase the probability of read-write inconsistencies.

Consider, for example, a CEO using a disconnected laptop to work on a report for an upcoming shareholder's meeting. Before disconnection he caches a spreadsheet with the most recent budget figures available. He writes his report based on the numbers in that spreadsheet. During his absence, new budget figures become available and the server's copy of the spreadsheet is updated. When the CEO returns and reintegrates, he needs to discover that his report is based on stale budget data. Note that this is not a write-write conflict, since no one else has updated his report. Rather it is a read-write conflict, between the spreadsheet and the report. No Unix system has the ability to detect and deal with such problems.

We have extended Coda with a new mechanism called *isolation-only transactions (IOTs)* to alleviate this shortcoming [11]. The IOT mechanism offers improved consistency for applications in a convenient and easy to use fashion. The mechanism is efficient, minimally demanding of resource-poor mobile clients, and upward compatible with existing Unix software.

An IOT is a sequence of file operations that are treated as a unit for purposes of conflict detection and resolution. The name ''IOT'' stems from the fact that this mechanism focuses solely on the *isolation* aspect of the classic ACID transactional properties [2]. In other words, IOTs do not guarantee failure atomicity and only conditionally guarantee permanence. The IOT subsystem of Venus performs automatic read/write conflict detection based on certain serializability constraints. It supports a variety of conflict resolution mechanisms such as re-execution and the use of ASRs.

Coda provides two ways to use IOTs. Users can use a special IOT shell to transactionally encapsulate selected unmodified Unix applications. Alternatively, they can modify applications using the IOT programming interface. Figure 6 shows an example of the use of IOTs in Coda.



The work window displays the process of disconnecting the client, setting `make` and `latex` as transactions in the special IOT shell, executing a `make` transaction and a `latex` transaction, reconnecting the client, and checking transaction status using the `lt`(list transaction) command. At reconnection time the `make` transaction is invalidated because the linked library `libutil.a` was updated on the servers. The IOT monitor window shows the automatic re-execution of the `make` transaction and the committing of the `latex` transaction.

Figure 6: Example of IOT Usage

4.4. Status and Experience

4.4.1. Evolution

Disconnected operation in Coda was implemented over a period of two to three years. A version of disconnected operation with minimal functionality was demonstrated in October 1990. A more complete version was functional in early 1991 and has been used since then by members of the Coda project.

Work on the extensions for weak connectivity began in 1993. The transport protocol extensions and rapid cache validation mechanism have been in regular use for over a year. The trickle reintegration and user advice mechanisms were implemented between 1994 and early 1995, and have recently been released for general use.

A prototype implementation of IOT support in Coda has been completed. An evaluation of this prototype based on controlled experiments confirms that the resource demands of IOTs are indeed acceptable in a mobile environment. This prototype now awaits more extensive use.

4.4.2. Current Deployment

Coda is currently deployed to a user community of Coda developers and other computer science researchers. Our deployment is currently on Mach 2.6, but we are porting Coda to NetBSD. We have over 40 user accounts, of which about 25 are used regularly. Many users run Coda on both their desktop workstations and their laptops. We have a total of about 35 Coda clients, evenly divided between workstations and laptops. The laptops are 486-based DEC 425SL's and IBM ThinkPad 701C's, while the workstations are mostly DECStation 5000/200's. These clients access almost 4.0 GB of data stored on Coda servers. Indeed, there are many more people wishing to use Coda than we can accommodate with hardware or support services.

4.4.3. Empirical Study

How will people use mobile computing? The answer to this question is important because it will critically influence the future designs of mobile computing systems. As a first step in answering this question, we have instrumented our deployed Coda system and have been conducting an ongoing empirical study of system and user behavior [14].

Our data shows that Coda clients do experience various kinds of service failures, but that Coda is able to mask these failures effectively. Our observations confirm many earlier simulation-based predictions on resource usage, as well as many anecdotal reports from our user community. Our study has also produced some surprises. For example, the number of transient failures observed has been far larger than anticipated. Another surprise is the tendency of users to limit mutation activity while voluntarily disconnected.

4.4.4. Source Code Distribution

Since 1992 Coda has been distributed in source code form to several sites outside of CMU. Porting Coda to a new machine type has proved to be relatively straightforward. Most of the code is outside the kernel. The only in-kernel code, a VFS driver [21], is small and entirely machine independent. Porting simply involves recompiling the Coda client and server code, and ensuring that the kernel works on the specific piece of hardware.

5. Odyssey: Application-Aware Adaptation

Although the viability of application-transparent adaptation has been demonstrated by Coda, there are important situations where it is inadequate. This is likely to be especially true of applications involving multimedia data such as videos and maps. Further, the Coda approach relies heavily on caching, and is likely to fall short when there is no temporal locality to exploit. This situation is likely to arise in scenarios involving search of data repositories from mobile clients.

We are exploring solutions to these problems in the context of Odyssey. Our approach is not to invent a new operating system but to extend Unix with a small yet powerful set of extensions for mobile computing. In keeping with this minimalist philosophy, we also strive to keep changes to existing applications small and consistent with Unix programming idiom. Since application transparency is a degenerate case of application awareness, we expect to eventually incorporate Coda as part of Odyssey. However, the initial development of the two systems is proceeding along separate paths.

5.1. Support for Application-Aware Adaptation

The need for application-aware adaptation can be seen from a simple example. Consider a movie player capable of displaying stored video images. Below a certain bandwidth and network quality, it will not be possible for the player to display the images in full-motion color. Extensive compression will help, but cannot solve the problem completely. But if the application is also capable of displaying the image in slow-scan black and white, it could automatically do so when bandwidth falls below a critical threshold.

5.1.1. Fidelity

That slow-scan black and white display is a reasonable form of degradation is specific to video data. Other data types may have entirely different forms of degradation that are meaningful to them. For example, increasing the minimum feature size displayed may be an appropriate form of degradation for map data. We define *fidelity* as the degree to which a copy of data presented for use at a client matches the reference copy at a server. Fidelity has many dimensions. One well-known, universal dimension is consistency; other dimensions depend on the type of data in question. The dimensions of fidelity are natural axes of adaptation for mobility. But the adaptation cannot be solely determined by the type of data; it also depends on the application. For example, if the application were a video editor rather than a video player, slowing the frame rate would be a more appropriate form of degradation than dropping frames to preserve frame rate.

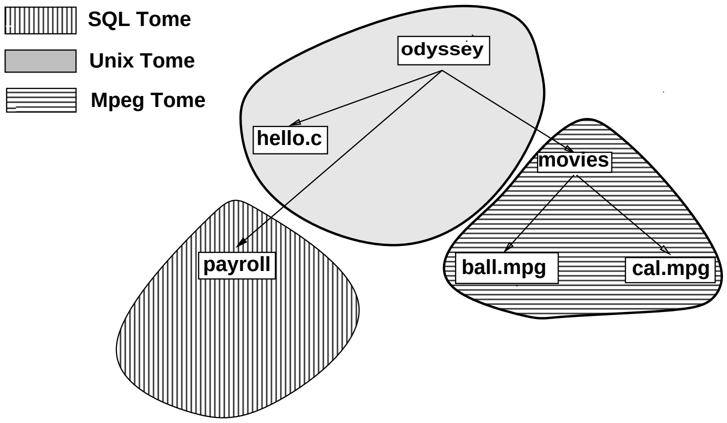
5.1.2. Resource Negotiation API

Odyssey provides an interface for resource negotiation [15]. The resources in question may be generic, such as network bandwidth, cache space, processor cycles, or battery life. Resources may also be application-specific, such as the number of queries left to a limited-subscription stock quotation service.

An application initially tries to access data at the highest level of fidelity. If the resources needed for this exceed what is currently available, Odyssey informs the application of this fact. The application then selects a fidelity level consistent with available resources. It also registers a window of tolerance for each resource of interest. At a later time, if the availability of a resource improves or degrades beyond this window, Odyssey will notify the application. It is the application's responsibility to then renegotiate the resources needed for an appropriate level of fidelity.

5.1.3. Name Space and Client Structure

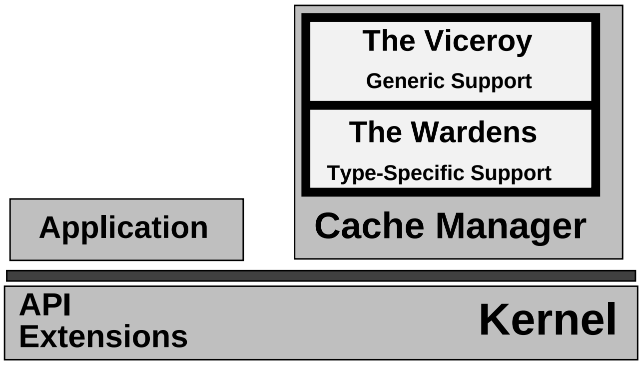
Odyssey provides a single, global namespace to its clients, as shown in Figure 7. This namespace is broken into subspaces called *tomes.* Tomes are conceptually similar to volumes in Coda and AFS, but incorporate the notion of type. The type of a tome determines the type-specific resources, operations, and dimensions of fidelity for all items in the tome.



This figure illustrates a sample Odyssey namespace. In this example, there are three tomes, each of a different type. The first tome, rooted at `odyssey`, contains the single UNIX file `hello.c`. The second, rooted at `payroll`, is a database. Note that no nodes appear inside of `payroll`; it is named associatively rather than hierarchically. The third tome, rooted at `movies`, contains two MPEG movies, `ball.mpg` and `cal.mpg`.

Figure 7: Tomes in Odyssey

As illustrated in Figure 8, the structure of an Odyssey client reflects the decomposition of functionality into generic and type-specific components. Generic functionality is implemented by the *viceroy*, whose most important task is to act as the single point of resource control in the system. The viceroy also services requests for generic resources, and plays a central role in resource negotiation.



This figure illustrates the architecture of an Odyssey client. Odyssey applications make use of the Odyssey API extensions along with the operating system's API. Operations on Odyssey objects are redirected by the kernel to the cache manager, which is at user level for ease of implementation. The cache manager is split into two logical pieces: the viceroy, providing generic support, and a set of wardens, each supporting a single type.

Figure 8: Odyssey Client Architecture

Type-specific functionality is implemented in cache managers subordinate to the viceroy, called *wardens*. There is one warden for each tome type, and it is invoked by the viceroy to service requests on Odyssey objects of that type. The wardens are responsible for implementing the access methods on objects of their type, and for providing support for different levels of fidelity. They also provide reasonable default policies to allow a modicum of backward compatibility with legacy applications.

5.2. Support for Dynamic Sets

How does one support search of data repositories from mobile clients that are weakly connected? Since there is little temporal locality to exploit, caching is unlikely to be helpful. Instead, our approach is to exploit the *associativity* inherent in search operations to overlap prefetching of data over slow networks with the computation or think time involved in a search task.

The vehicle we are using for this aspect of our research is a new operating system abstraction called *dynamic sets* [22, 23]. The essence of the abstraction is the explicit grouping of sets of file accesses and the communication of this grouping by applications to the operating system. This simple abstraction can have surprisingly powerful performance implications for mobile search. Figure 9 presents the most important system calls in the Odyssey API for dynamic sets.

```
setHandle setOpen(char *setPathname);

errorCode setClose(setHandle set);

fileDesc  setIterate(setHandle set, int flags);

errorCode setDigest(setHandle set, char *buf, int count);
```

A dynamic set is created by calling `setOpen` with a *set pathname*, and receiving a *set handle* for the open set in return. The system can expand the set into its members and fetch these members as aggressively as resources warrant. Once open, the membership of a set can be browsed using `setDigest`. An individual member can be accessed using `setIterate`, which returns a file descriptor as if an `open` had been performed on the member selected. The system is free to iterate through the set in any order. `setClose` terminates use of a set handle.

Figure 9: Core Subset of Dynamic Sets API

By using dynamic sets, an application discloses the membership of a group of related files. This disclosure offers the system a strong hint of future file accesses that can be exploited for prefetching. In addition, by using a set to represent the grouping, the system is free to optimize the order in which the set members are fetched. For example, if some members of the set happen to be cached they can be returned first. The fetching of the later members can be overlapped with the processing of earlier members.

5.3. Status and Experience

We have completed a simple, skeletal implementation of the Odyssey architecture. This includes a library implementation of the API for application-aware adaptation, as well as the wardens, servers, and applications for video and map data. Although the prototype is rudimentary in many respects, it provides initial evidence of the overall validity of our approach. Based on this positive feedback, we are implementing a more complete, in-kernel prototype.

Our work in dynamic sets has gone through two phases. In the first phase, we built a user-level library implementation of the dynamic sets API. Although the prototype's absolute performance was modest due to implementation inefficiencies, it was adequate to confirm the substantial benefit of dynamic sets. We codified our experience into a validated performance model, and used it to explore whether the effort of a more complete and efficient implementation of dynamic sets was justified. Based on the encouraging results of our analysis, we have embarked on the second phase and are close to completing an in-kernel implementation of dynamic sets.

6. Conclusion

Our work bears a complementary relationship to the other efforts described in this special issue. Mobile IP, described by Johnson and Maltz [5], represents a networking layer below Coda and Odyssey. The different quality streams of Generative Video, described by Moura et al [12], correspond to different levels of video fidelity on an Odyssey client. The applications described by Bruegge and Bennington [1] could benefit from Coda's support for mobile file access. The Wireless Andrew Network, described by Hills and Johnson [3], provides the infrastructure nececssary for the Coda user community to remain connected while mobile. Finally, the wearable computers described by Smailagic and Siewiorek [20] are now powerful enough to run Coda, thus enabling a new and unique class of applications.

The ability to access information on demand when mobile will be a critical capability in the 21st century. As elaborated in this paper, adaptation is the key to this capability. Our research is exploring two different approaches to adaptation: application-transparent and application-aware. Our experience with Coda confirms that application-transparent adaptation is indeed effective in many cases. In circumstances where it is inadequate, our initial experience with Odyssey suggests that application-aware adaptation is the appropriate strategy.

Acknowledgements

Although this paper has a single author, it is based on the contributions of many past and present Coda and Odyssey project members, including Jay Kistler, Puneet Kumar, David Steere, Lily Mummert, Maria Ebling, Hank Mashburn, Brian Noble, Josh Raiff, Qi Lu, Morgan Price, Bob Baron, Dushyanth Narayanan, and Eric Tilton. Perhaps the most important contribution of all has been made by the Coda user community, through its bold willingness to use and help improve an experimental system.

Further Reading

This paper provides only the briefest overview of our work. A detailed annotated guide to Coda and Odyssey papers may be found on the World Wide Web at this URL:

`http://www.cs.cmu.edu/afs/cs.cmu.edu/project/coda/Web/coda.html`

References

[1] Bruegge, B., Bennington, B.
Applications of Mobile Computing and Communications.
IEEE Personal Communications 3(1), February, 1996.

[2] Gray, J., Reuter, A.
Transaction Processing: Concepts and Techniques.
Morgan Kaufmann, 1993.

[3] Hills, A., Johnson, D.B.
Wireless Data Network Infrastructure at Carnegie Mellon University.
IEEE Personal Communications 3(1), February, 1996.

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

[5] Johnson, D.B., Maltz, D.A.
Protocols for Adaptive Wireless and Mobile Networking.
IEEE Personal Communications 3(1), February, 1996.

[6] Kistler, J.J., Satyanarayanan, M.
Disconnected Operation in the Coda File System.
ACM Transactions on Computer Systems 10(1), February, 1992.

[7] Kistler, J.J.
Disconnected Operation in a Distributed File System.
PhD thesis, Department of Computer Science, Carnegie Mellon University, May, 1993.

[8] Kumar, P., Satyanarayanan, M.
Log-Based Directory Resolution in the Coda File System.
In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*. San Diego, CA, January, 1993.

[9] Kumar, P.
Mitigating the Effects of Optimistic Replication in a Distributed File System.
PhD thesis, School of Computer Science, Carnegie Mellon University, December, 1994.

[10] Kumar, P., Satyanarayanan, M.
Flexible and Safe Resolution of File Conflicts.
In *Procedings of the 1995 USENIX Technical Conference*. New Orleans, LA, January, 1995.

[11] Lu, Q., Satyanarayanan, M.
Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions.
In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*. Orcas Island, WA, May, 1995.

[12] Moura, J.M.F., Jasinchi, R.S., Shiojiri, H., Lin, J-C.
Video Over Wireless.
IEEE Personal Communications 3(1), February, 1996.

[13] Mummert, L.B., Ebling, M.R., Satyanarayanan, M.
Exploiting Weak Connectivity for Mobile File Access.
In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. Copper Mountain Resort, CO, December, 1995.

[14] Noble, B., Satyanarayanan, M.
An Empirical Study of a Highly-Available File System.
In *Proceedings of the 1994 ACM Sigmetrics Conference*. Nashville, TN, May, 1994.

[15] Noble, B., Price, M., Satyanarayanan, M.
A Programming Interface for Application-Aware Adaptation in Mobile Computing.
In *Proceedings of the 1995 USENIX Symposium on Mobile and Location-Independent Computing*. Ann Arbor, MI, April, 1995.

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

[17] Satyanarayanan, M., Kistler, J.J., Mummert, L.B., Ebling, M.R., Kumar, P., Lu, Q.
Experience with Disconnected Operation in a Mobile Computing Environment.
In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing*. Cambridge, MA, August, 1993.

[18] Satyanarayanan, M., Mashburn, H.H., Kumar, P., Steere, D.C., Kistler, J.J.
Lightweight Recoverable Virtual Memory.
ACM Transactions on Computer Systems 12(1):33-57, February, 1994.

[19] Satyanarayanan, M., Mashburn, H.H., Kumar, P., Steere, D.C., Kistler, J.J.
Corrigendum: Lightweight Recoverable Virtual Memory.
ACM Transactions on Computer Systems 12(2):165-172, May, 1994.

[20] Smailagic, A., Siewiorek, D.P.
Modalities of Interaction with CMU Wearable Computers.
IEEE Personal Communications 3(1), February, 1996.

[21] Steere, D.C., Kistler, J.J., Satyanarayanan, M.
Efficient User-Level Cache File Management on the Sun Vnode Interface.
In *Summer Usenix Conference Proceedings, Anaheim*. June, 1990.

[22] Steere, D.C., Satyanarayanan, M.
Using Dynamic Sets to Overcome High I/O Latencies During Search.
In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*. Orcas Island, WA, May, 1995.

[23] Steere, D.C.
Using Dynamic Sets to Speed Search in World Wide Information Systems.
Technical Report CMU-CS-95-174, School of Computer Science, Carnegie Mellon University, March, 1995.