Palo Alto Research Center

Using Property Specifications to Achieve Graceful Disconnected Operation in an Intermittent Mobile Computing Environment

Michael Tso

XEROX

Using Property Specifications to Achieve Graceful Disconnected Operation in an Intermittent Mobile Computing Environment

Michael Tso

CSL-93-8 June 1993 [P93-00018]

CR Categories and Subject Descriptors: **D.4.3** [Operating Systems]: File Systems Management - distibuted file systems, **D.4.7** [Operating Systems]: Organization and Design - interactive systems, **H.5.2** [Information Interfaces and Presentation]: User Interfaces

Additional Keywords and Phrases: caching, disconnected, file system, hints, portable computer, predictable, property specification, splitting, wireless computer

General Terms: Design, Human Factors, Performance, Reliability

XEROX

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

USING PROPERTY SPECIFICATIONS TO ACHIEVE GRACEFUL DISCONNECTED OPERATION IN AN INTERMITTENT MOBILE COMPUTING ENVIRONMENT

by

MICHAEL MAN-HAK TSO

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

MASTER OF SCIENCE in Electrical Engineering and Computer Science

and

BACHELOR OF SCIENCE in Computer Science and Engineering

and

BACHELOR OF SCIENCE in Electrical Science and Engineering

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 1993

Signature of Author_____
Michael M. Tso
MIT Department of EECS, May 21, 1993

Certified by_____
Dr. David D. Clark
Senior Research Scientist, MIT Department of EECS

Certified by_____
Dr. David Goldberg
Research Scientist, Computer Science Laboratory, Xerox Palo Alto Research Center

Accepted by_____
Professor Campbell L. Searle
Chair, MIT Department of EECS Committee on Graduate Students

USING PROPERTY SPECIFICATIONS TO ACHIEVE
GRACEFUL DISCONNECTED OPERATION IN AN
INTERMITTENT MOBILE COMPUTING ENVIRONMENT

by

MICHAEL MAN-HAK TSO

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 1993 in Partial Fulfillment of the Requirements for the Degrees of
MASTER OF SCIENCE in Electrical Engineering and Computer Science,
BACHELOR OF SCIENCE in Computer Science and Engineering, and
BACHELOR OF SCIENCE in Electrical Science and Engineering

Abstract

This thesis studies the problem of providing autonomy and predictable performance for computers with an intermittent network, e.g. wireless computers. For autonomy, we support Coda's programming model in which applications run on the portable computer and access data through a caching distributed file system. Although caching lets applications operate disconnected, it can also cause applications to be unpredictable. The problem is that caching is transparent in existing file system interfaces. Since the availability of files is unpredictable, it is impossible for the application to predict which of its features are (or will be) available. For improving predictability, we introduce a new system level abstraction, Property Specifications which enables system services to expose to the application fundamental aspects of the computing environment in a structured way. The application can become more predictable by unobtrusively informing the user the effects of changes in the environment, such as becoming disconnected, through the application's user interface. The application changes the user's expectation about which features are (or will be) available. In order for applications to efficiently access, monitor, influence and manipulate the exported properties, Property Specifications provide mechanisms for Query, Notification and Hinting. Applications learn about the current state of system properties through Query. Changes in the state of properties are modeled as Environmental Events. Notification enables applications to monitor changes in the environment by binding callback procedures to Environmental Events. Hinting lets applications give optional information to influence and sometimes manipulate the system's properties. Collectively, Query, Notification and Hinting define a framework for structured communication and collaboration between applications and system services.

We designed and implemented Property Specifications for a caching distributed file system and a network interface which exported properties of the file cache and a network with variable latency. We added new features to an existing mail tool (xmh) to improve its predictability and autonomy. In a simulated intermittent environment, our users found these features to be very effective in making xmh more friendly and usable. Property Specifications enabled us to implement these features easily and efficiently.

By specifying the properties of the environment and the functions provided by the implementation in separate interfaces, we give power and flexibility to sophisticated applications while maintaining transparency for ordinary applications.

Thesis Advisor: Dr. David D. Clark
Title: Senior Research Scientist, MIT Laboratory for Computer Science

Thesis Advisor: Dr. David Goldberg
Title: Research Scientist, Computer Science Laboratory, Xerox PARC

Acknowledgments

I thank my advisors David Goldberg and David Clark for their patience and guidance. Their insight in debugging my half baked ideas amazes me. I feel fortunate to have had such brilliant and understanding advisors, from whom I have learned a great deal.

I am grateful to Greg Papadopoulos, Jeannette Wing, and Gregor Kiczales. Even though I was never officially one of their advisees, Greg, Jeannette and Gregor were always willing to listen and provide encouragement. I deeply respect them as colleagues and as friends.

Marvin Theimer, Brent Welch, David Nichols and David Goldberg deserve special thanks for participating in the experimental part of this thesis. Together with Jeannette Wing, Bill Schilit, and Bob Scheifler, they provided a lot of valuable feedback while my ideas for this thesis were still in fermentation.

The preparation of this thesis document benefited greatly from Hector Ayala's Macintosh wizardry as well as Stephen Wong and Joseph DiMare's willingness to loan me their Powerbooks. The Powerbooks were critical in helping me finish this thesis despite my hectic travel schedule this semester.

Thanks to Bill, Dick, Donger, Fergie, Furball, Gilly-Billy, Hanes, Jerko, Joe, Looney, Norton, Prashun, Shin, Toast, Van Veen, and Vasik. Your friendship and sense of humor make me want to repeat the MIT experience again despite all those late nights, mind boggling problem sets, interminable labs, and harrowing exams.

My education would not have been possible without the help of several very dear people. I thank my financee, Cecilia Wong, whose love and encouragement always give me the energy and will to go on. The 10,000 miles that separated us for four years could not divide our hearts. May Poon and Andrew and Cissy Chung opened their homes and hearts to me during those trying years of my adolescence. Without their nurturing, I probably would not have graduated from high school. I owe my inspiration and determination to my mother. Although her body rests, her spirit lives on. Finally, I thank God for His immense love. All I had to offer Him was brokenness and pain, but He made something beautiful of my life.

Table Of Contents

Chapter 1

Introduction

The premise of this thesis is that autonomy and predictable performance (availability and response time) were key to the success of PCs and workstations over timesharing systems. This thesis studies the problem of extending these properties to intermittent computing, e.g. wireless mobile computing. We assume that wireless computers are mostly limited by an **intermittent** network which has low bandwidth, high latency, and unpredictable availability. Computers using radio or infrared networks can be frequently and unpredictably disconnected due to interference and coverage limitations. The proposed solution is a new programming model and system abstraction that support autonomous, predictable operation.

We choose to use Coda's [Kistler] [Satya90] programming model in which the application runs on the portable computer and accesses data through a caching network file system. We call this the **Autonomous programming model**. The main advantage of the Autonomous model is increased autonomy: applications can operate disconnected using files cached on the portable machine. In addition, by splitting the application at the file system layer instead of the window system layer such as X [Scheifler], the user

interface response time is no longer directly dependent on the network round-trip delay. Chapter 3 discusses other programming models and how they are affected by different assumed operating environments.

The main difficulty with using caching to increase availability is that from the application's point of view, the content of a transparent cache is inherently unpredictable. Hence an application whose availability depends on the cache is also unpredictable. A disconnected application is **partially functional** when some of its features require the use of the network or files which are not in the cache. Neither the application and nor the user knows which features work and which do not because the availability of the files needed by the application is unpredictable. This can be frustrating for the user. For example, a user reading NetNews has to try clicking on every article in order to find out which ones he can read. The goal of this thesis is to achieve predictable performance through **graceful disconnected operation** - the ability for partially functional applications to remain user friendly by providing predictable performance.

One way to achieve predictable performance is to guarantee that the data and resources the user needs are always available. This is not possible because we have an intermittent network and we do not have an infinite cache. Our approach is to accept the fact that applications will be partially functional when disconnected, and achieve predictable performance by unobtrusively indicating which features work and which do not. The application's user interface changes as the availability of its features and data changes, affecting the user's expectations about what works and what does not. For example, a disconnected mail tool can gray out unavailable buttons such as "get new mail", and distinguish those messages which are available by italicizing their headers. Similarly, as the user prepares to disconnect voluntarily, the mail tool can tell him what will be unavailable by visually distinguishing those buttons which will not work. These new

functionalities require the application to have intimate knowledge of its computing environment, e.g. the current and expected availability of the files and system services needed by each of its features. Our key innovation is in informing the user of the effects of fundamental properties of the computing environment, such as disconnections, through the application's user interface.

Without adequate support, forcing the application to deal with the operating environment's dynamic properties will cause significant complexities for programmers. We introduce **Property Specifications**, a new abstraction which enables system services to expose fundamental aspects of the computing environment in a structured way. Most existing system interfaces specify only the functions and services provided by the implementation, and we refer to them as **Functional Specifications**. In contrast, Property Specifications define an interface for the application to access the properties of the computing environment without being exposed to irrelevant details of the implementation. Properties which affect the performance and availability of applications may include the availability of files, network latency, and expected battery life.

In addition, we define three basic mechanisms that Property Specifications should provide for applications to efficiently access, monitor, influence and manipulate the exported properties. They are **Query**, **Notification** and **Hinting**. Applications learn about the current state of system properties through Query. Property changes are modeled as **Environmental Events**. Notification enables applications to monitor changes in the environment by binding callback procedures to Environmental Events. Hinting lets applications give optional information to influence and sometimes manipulate the system's properties. Collectively, Query, Notification and Hinting define a framework for structured communication and collaboration between applications and system services.

The actual semantics for Environmental Events and the above mechanisms depend on the specific properties of the particular system service. We designed and implemented Property Specifications for a caching distributed file system and a Network Statistics Monitor. Our file system interface exposes the property that files are moved in or out of the cache. We export four new procedures:

- `FilesAvailable()`, which lets an application query the cache manager about the availability of files;
- `MonitorFiles()`, which lets an application monitor the availability of one or more files, and be called back by the cache manager whenever any of these files are moved in or out of the cache;
- `GiveHint()`, which allows the application to influence the cache manager's replacement policy;
- `MakeAvailable()`, which is an explicit hint given by the application to the cache manager requesting some files to be moved into the cache.

Our network interface exposes the property that network operations have associated latencies which vary over time. We export two procedures:

- `GetLatency()`, which returns the current expected network latency;
- `MonitorLatency()`, which lets the application be called back whenever the network latency moves in or out of its operating range.

We used the above interfaces to implement **wc-xmh** (weakly connected xmh[1] [Peek]), a modified version of xmh with new features to enhance usability during disconnected operation. Users ran wc-xmh in a simulated intermittent environment and found the new features to be effective and user friendly. Our users reaffirmed our belief that predictable

[1]xmh is an X mail tool based on the mh message handling system [Peek].

performance is vital to usability. Wc-xmh's adaptive user interface allowed users to continue working during disconnections. Hinting proved useful for voluntary disconnections as it allowed users to directly negotiate with the application about what features and data objects to make available for future use. The user manipulates application level entities such as folders and features rather than system level entities like files, and is thus hidden from the internal dependencies of the application. Our implementation experience helped us understand the system level tools and abstractions required to reduce the programming complexity needed to obtain application features similar to those we implemented for wc-xmh.

One of the key contributions of this thesis is in separating Property Specifications from Functional Specifications. This gives the application programmer the flexibility of trading programming effort for application robustness. An application which uses only the Functional interface is easy to program but not very robust, e.g. it may crash if the network fails. It takes more effort to program an application using both the Property interface and the Functional interface, but the application would be very robust, e.g. it grays out some buttons when the network fails. The separation of the two interfaces is of key importance because the Property interface can support sophisticated applications which need to look inside black box abstractions without sacrificing any transparency at the Functional interface.

1.1 Outline

In Chapter 2, we use wc-xmh's features to illustrate what we mean by graceful disconnected operation, and compare our goal and approach with related work in distributed file systems and operating systems. In Chapters 3 through 5, we revisit the

sequence of ideas which enabled us to implement the features described in Chapter 2. We choose the Autonomous programming model in Chapter 3 after surveying other programming models for application partitioning. In Chapter 4 we discuss the motivation and mechanisms for separating system interfaces into Functional Specifications and Property Specifications. The Property Specifications for a caching distributed file system, a network service interface and a virtual memory manager are also presented. Chapter 5 describes the lessons we learned during the design and implementation of our prototype system and highlight some ideas for future work. Chapter 6 summarizes our users' experiences with wc-xmh, and conclude by evaluating our ideas in light of our design, engineering and usage experiences.

Chapter 2

Motivation and Related Work

This chapter elaborates on the basic goal of this thesis, to provide new application features for graceful disconnected operation. We draw examples of application features from wc-xmh, such as availability indicators for individual messages, informative error messages, managing future availability and monitoring environmental changes. The general applicability of these features in other applications is also discussed. Section 2.2 describes related work in distributed file systems as well as ideas similar to Property Specifications found in operating systems and programming language implementations.

2.1 User Level Features for Graceful Disconnected Operation

We built a prototype system to demonstrate graceful disconnected operation and explore the design space of Property Specifications. The design of our Property Specifications was heavily influenced by the user level features we wished to support. In the following

sections, we will illustrate some of wc-xmh's features which motivated our system design. In Section 5.2, we describe how we implemented these features in wc-xmh using the tools and abstractions we will describe in the next three chapters.

2.1.1 Availability Indicators

When wc-xmh is disconnected, some of the messages or folders will not be available for browsing and some of the features in menus and buttons may not work. We make clever use of wc-xmh's user interface so the user can be unobtrusively informed of what works and what does not. We annotate the available messages with asterisks and gray out the unavailable buttons and menus. The network performance is displayed by a thermometer: the more asterisks, the higher the network latency. This is illustrated by Figures 2.1a and 2.1b. We note that most of wc-xmh's features are still available even when it is disconnected. The user can browse cached messages and folders, delete or refile any message, commit changes, and compose new messages.

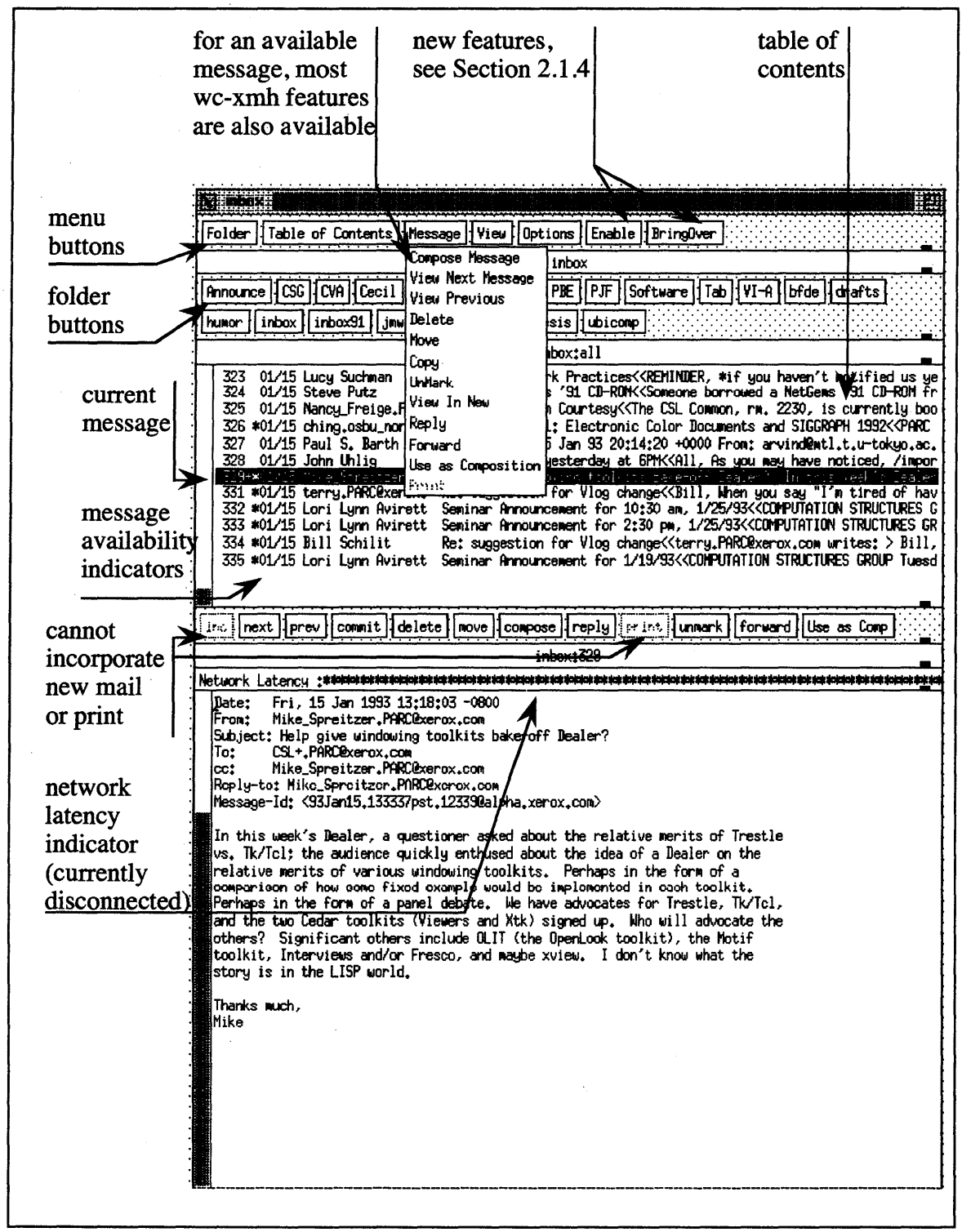

Figure 2.1a - Availability indicators for a cached message during disconnected operation

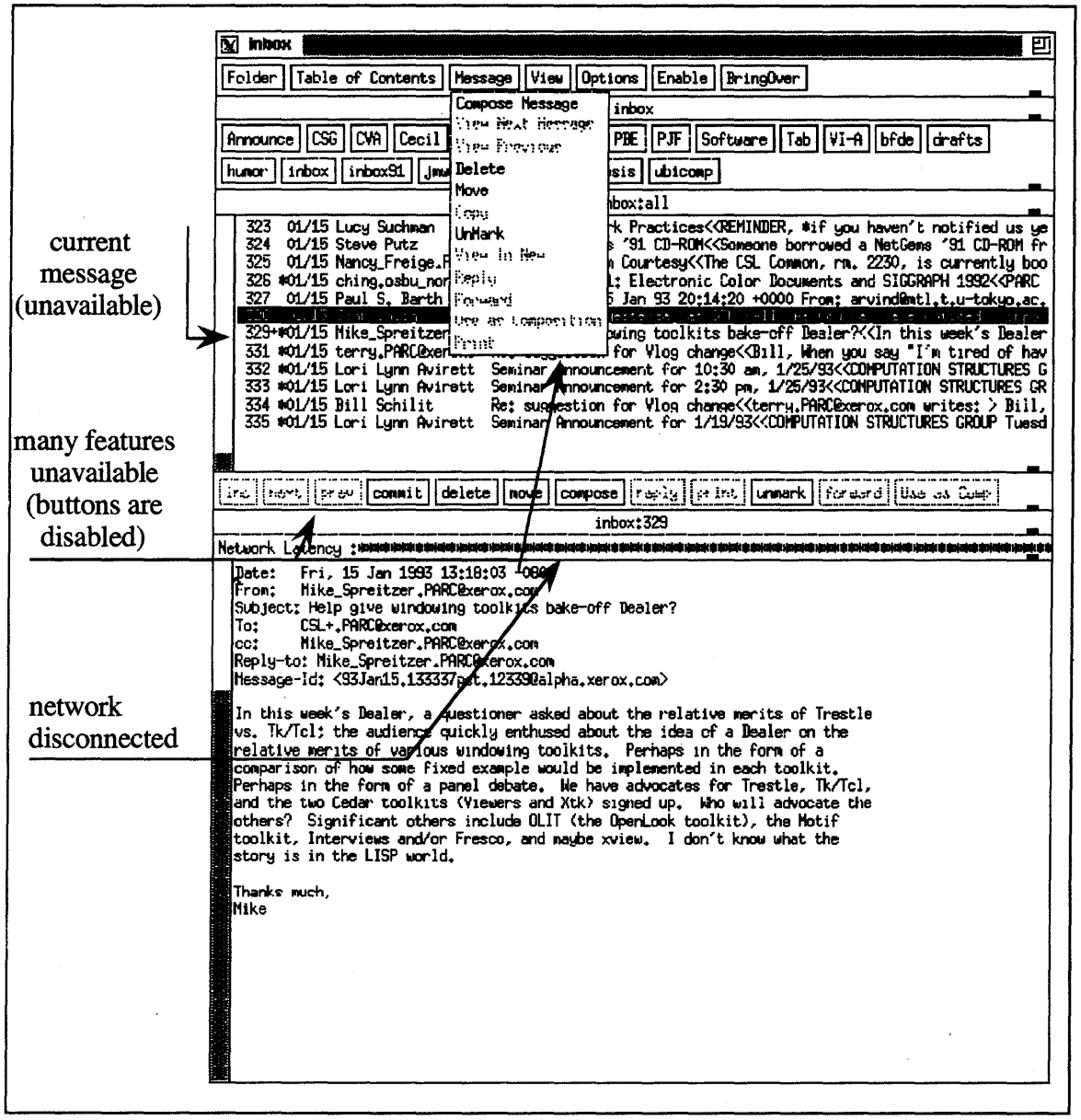

Figure 2.1b - Availability indicators for an uncached message during disconnected operation

2.1.2 Delayed Operation and Friendly Errors

When wc-xmh is disconnected, the user can still compose and send messages. Outgoing messages are queued by wc-xmh in a folder named Out, so the user can easily check

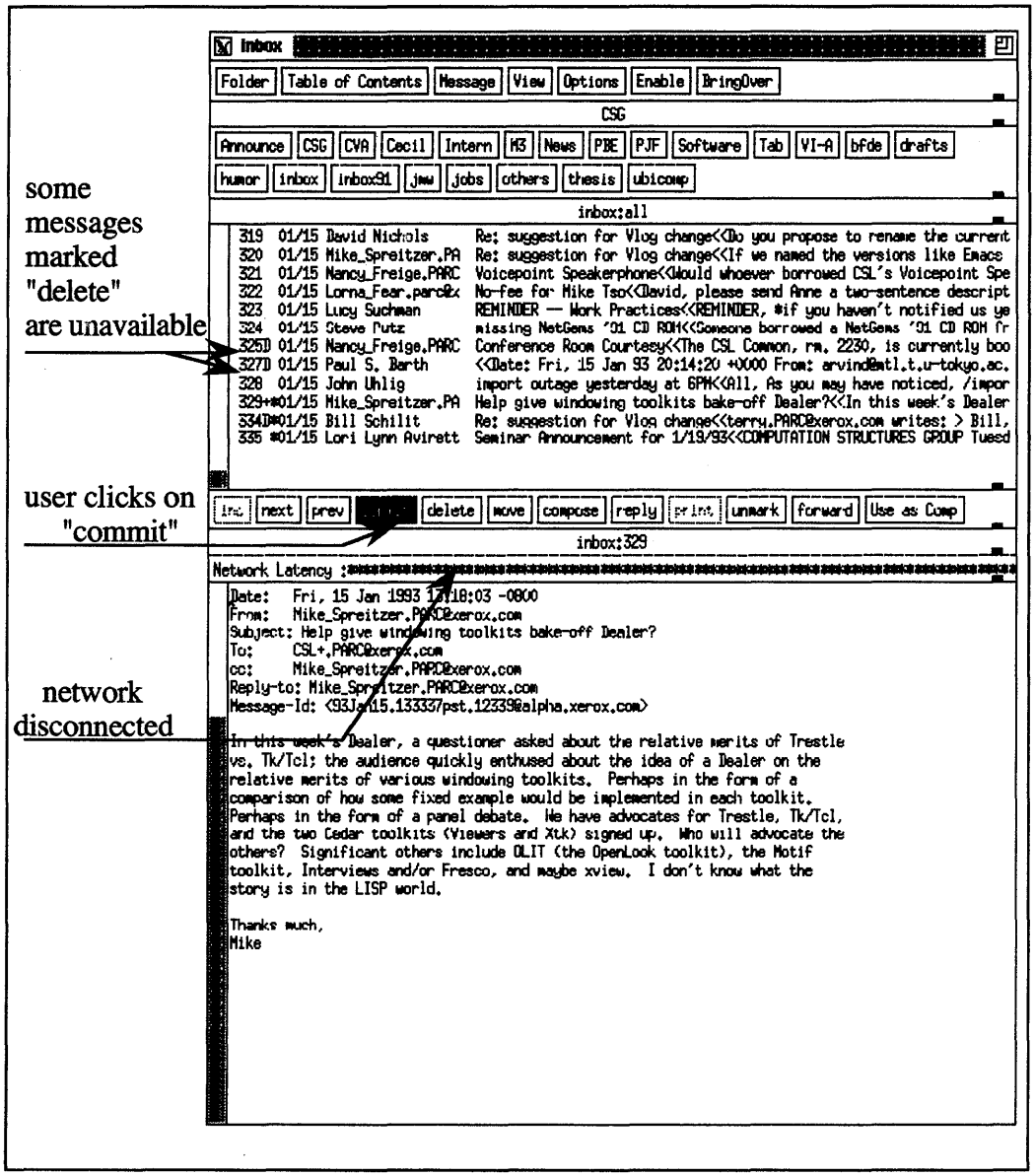

Figure 2.2a - User chooses to commit changes to unavailable messages during disconnection

which pending messages have or have not been sent. He can also edit, delete or move messages in the Out folder just like messages in any other folder. We feel that this feedback information is very important and is best provided within the context of the application, i.e. it is more intuitive for the user to manipulate pending outgoing messages as wc-xmh objects instead of files in the file system.

Not all errors can be prevented by disabling buttons. Handling errors at the system layer simplifies the application but is often too general because there is little information about the overall context of the error. Without knowing which application feature caused the

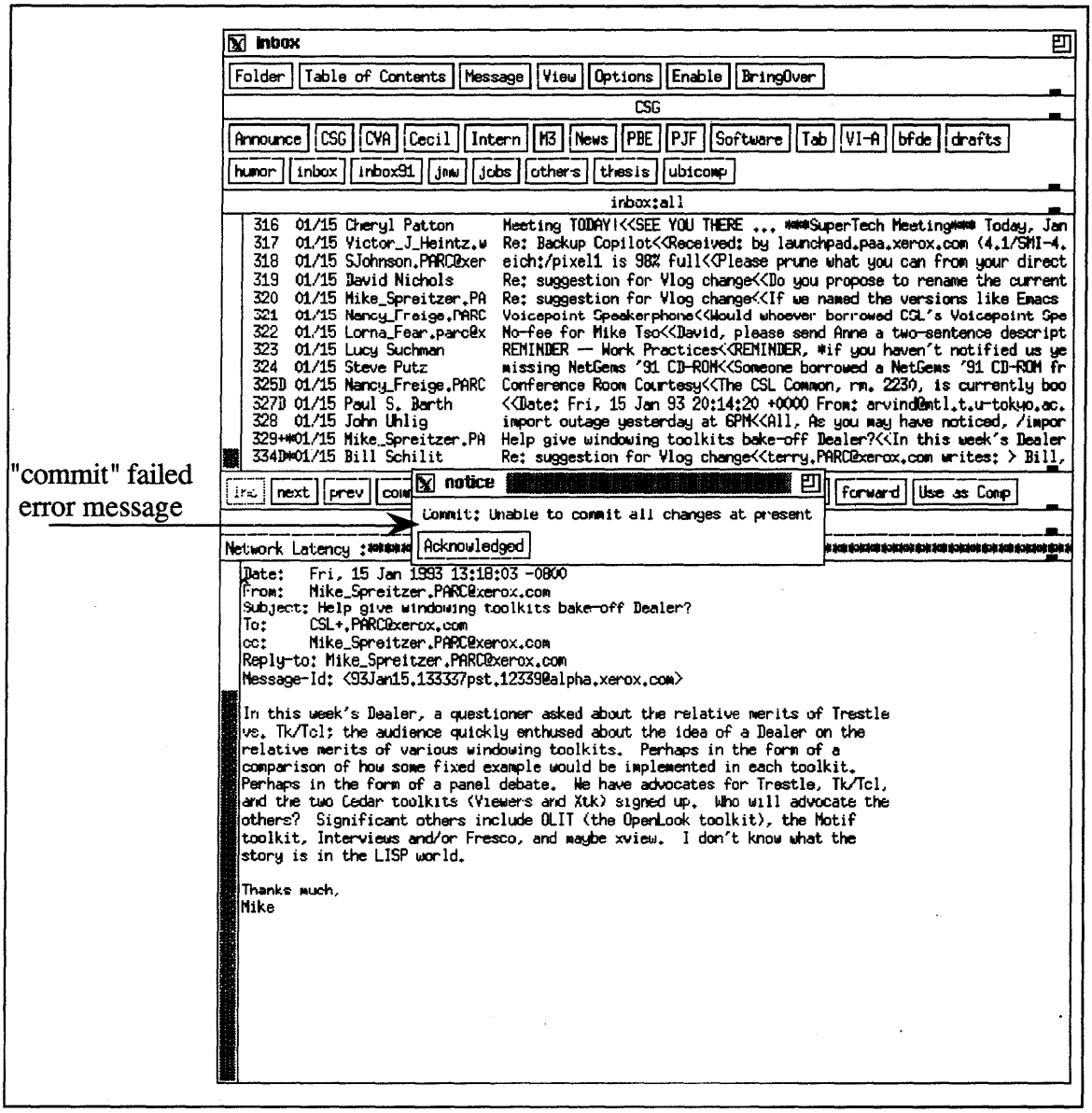

Figure 2.2b - Commit feature exits with a more meaningful error

error, system level error messages are often uninformative. For example, when the user clicks the "compose" button to create a new mail message, he might get a "file not found:

compform" error. The average user would not understand this error: compform is a template file used to create a new message header, and the user's command failed because wc-xmh is disconnected and compform is not in the cache. We believe that in general, the system layer is much better at detecting errors than handling errors. Uninformative error messages are intolerable if errors occur frequently. We allow the application to override the default error handler provided by the system to return more user friendly error messages based on the application's semantics. Figures 2.2a and 2.2b illustrate this idea: wc-xmh had to abort the "commit" command[2] prematurely because it could not verify the existence of unavailable messages. Instead of the file system timing out and printing a low level error to the console like "RPC timed out, retrying...", the error is intercepted by wc-xmh which notifies the user and aborts the "commit" operation instead of hanging on retries.

2.1.3 Smart Availability Management

It is no accident that wc-xmh is almost fully functional when it is disconnected. Maintaining a high level of functionality requires a number of resource files (such as the context, header summaries and filters) to be available. One of the key features of wc-xmh is its ability to influence the cache manager so that these critical files remain available as much as possible. Similarly, wc-xmh influences the cache manager to always make new mail messages available, as shown in Figures 2.3a and 2.3b.

[2] typically, the user marks the changes he wants to make to individual messages, e.g. move or delete, and then hits the "commit" button to actually make the changes at the file system level, e.g. renaming or deleting files.

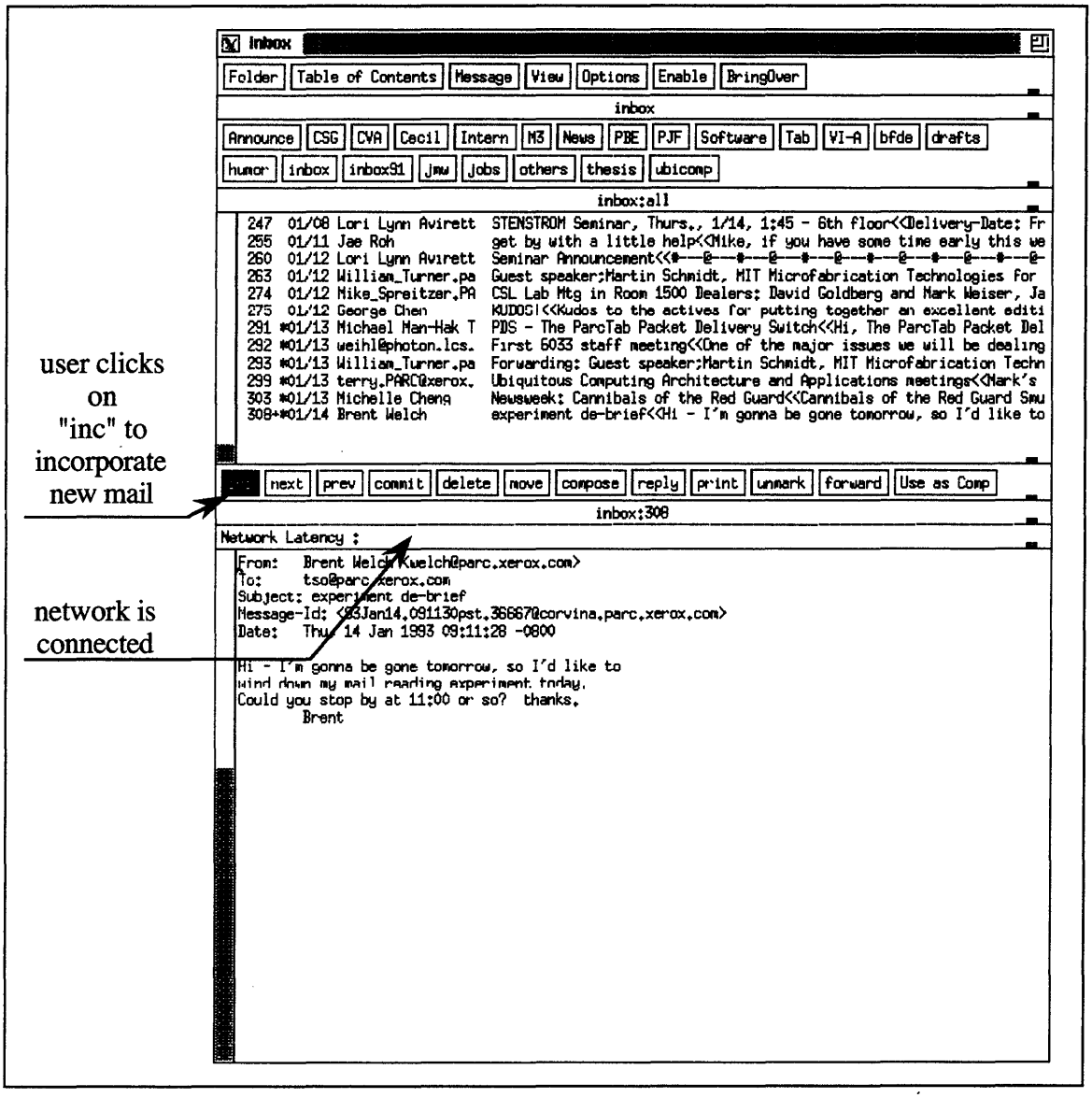

Figure 2.3a - User chooses to get new mail

2.1.4 Dependable Future Availability

When the user plans for voluntary disconnection, wc-xmh lets him check what data and features will be available when he becomes disconnected. If a feature will not be

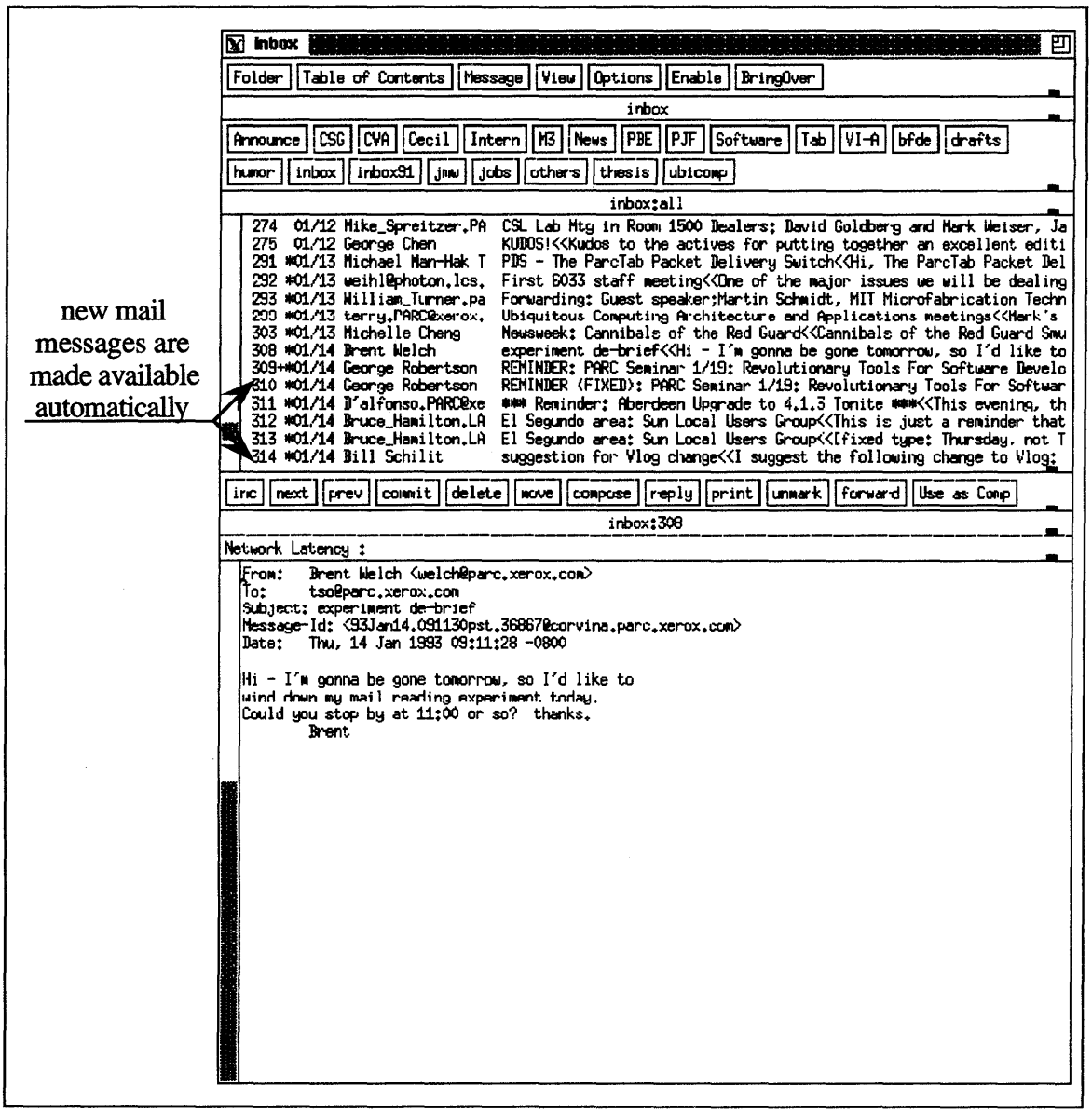

Figure 2.3b - New mail is automatically made available

available, its entry in the "Enable" menu would not be grayed out, as shown in Figure 2.4. The user can make a feature available by selecting the appropriate "Enable". He can also make individual messages or folders available by using the "BringOver" menu, as shown in Figures 2.5a and 2.5b. Wc-xmh informs the user whether his request was successful. The key utility of the "Enable" and "BringOver" features is that they allow the user to establish a level of dependable service across disconnections without having to understand wc-xmh's internal dependencies.

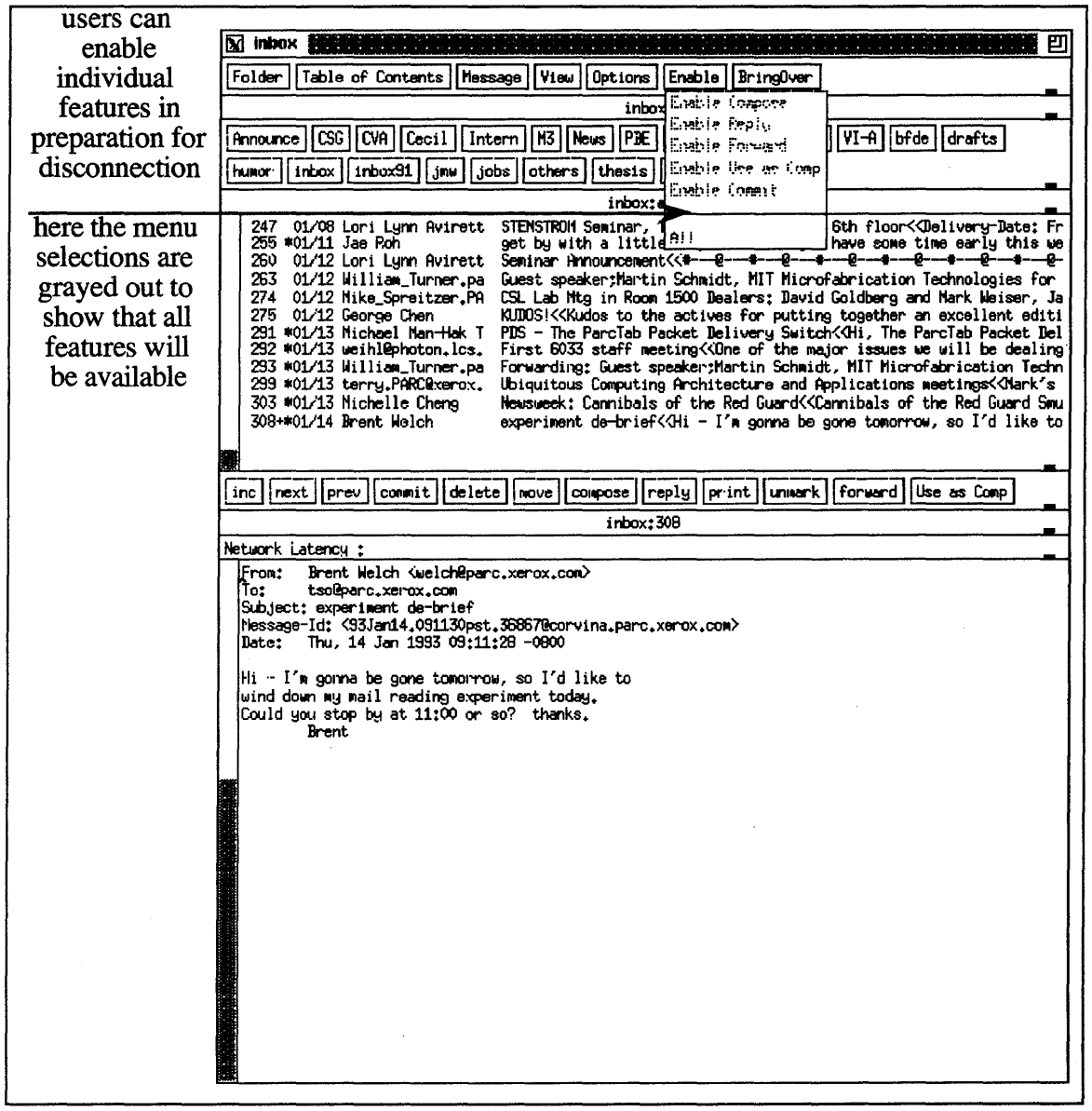

Figure 2.4 - Enable features for dependable future availability

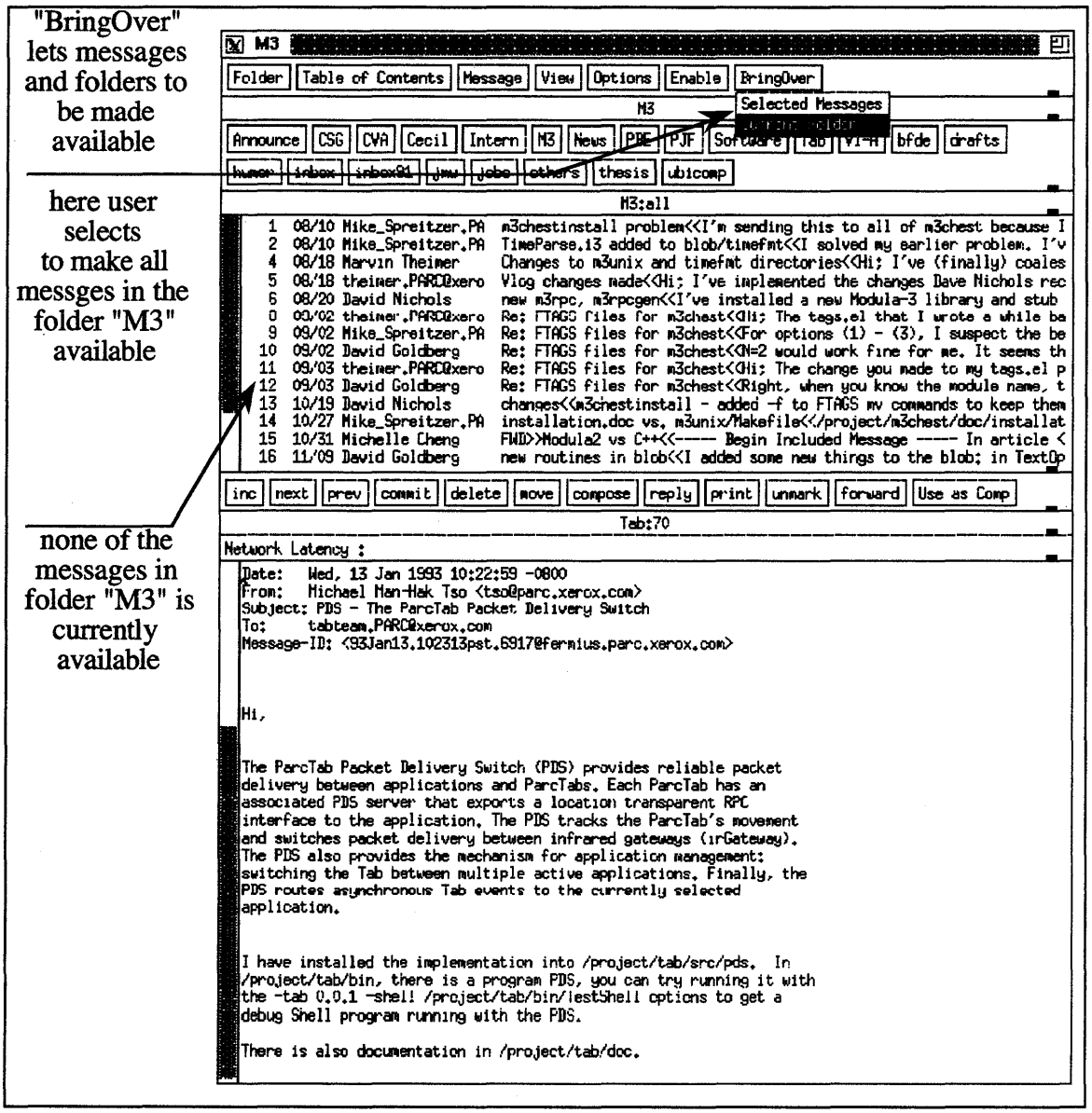

Figure 2.5a - User chooses to make some messages available for voluntary disconnection

Another idea for influencing the user's expectations is to use tri-state buttons. A tri-state button has three visually distinct states: currently available and will be available if disconnected, currently available and will be unavailable if disconnected, and currently unavailable. This is illustrated in Figure 2.6. Tri-state buttons let the user easily anticipate what will not work should a disconnection occur, thus enabling the application

to deliver a predictable level of service regardless of whether disconnections are planned or unplanned.

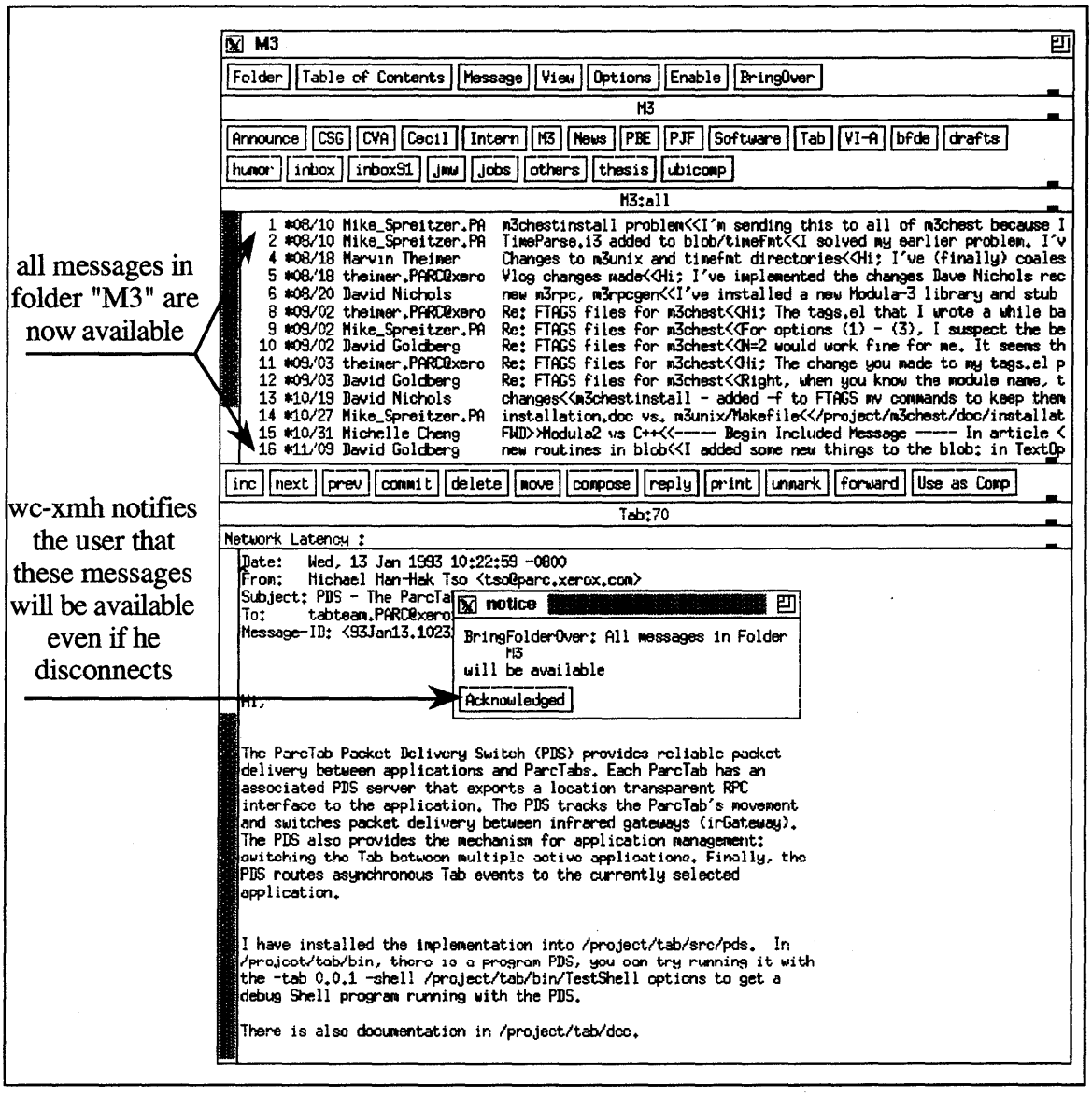

Figure 2.5b - User gets immediate feedback

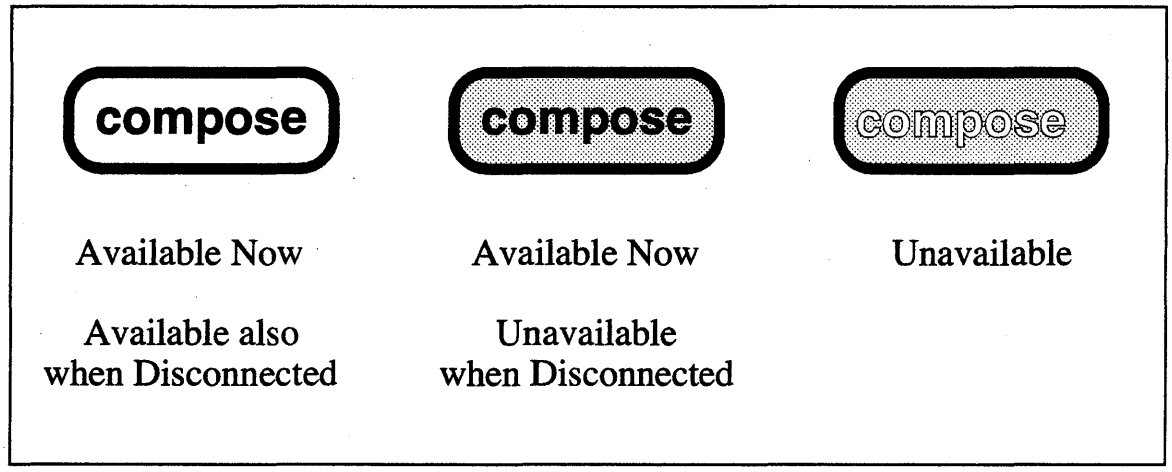

Figure 2.6 - Using Tri-state Buttons to Manage Future Availability

2.1.5 Monitoring and Reacting to Environmental Changes

The computing environment may be changing continuously: the network connectivity may fluctuate and files are paged in and out of the cache. Wc-xmh needs to monitor these changes so the state of its buttons, availability indicators and network thermometer can be updated in a timely fashion. A naive implementation would poll the cache manager and the Network Statistics Monitor. Notification allows these features to be implemented efficiently by invoking callback procedures as soon as changes are detected. Figures 2.7a and 2.7b illustrate the result of invoking a callback procedure after a file needed by the "compose" function is paged out. Typically, instead of drawing attention to the user, wc-xmh's callback procedure would automatically attempt to make "compose" available through Hinting.

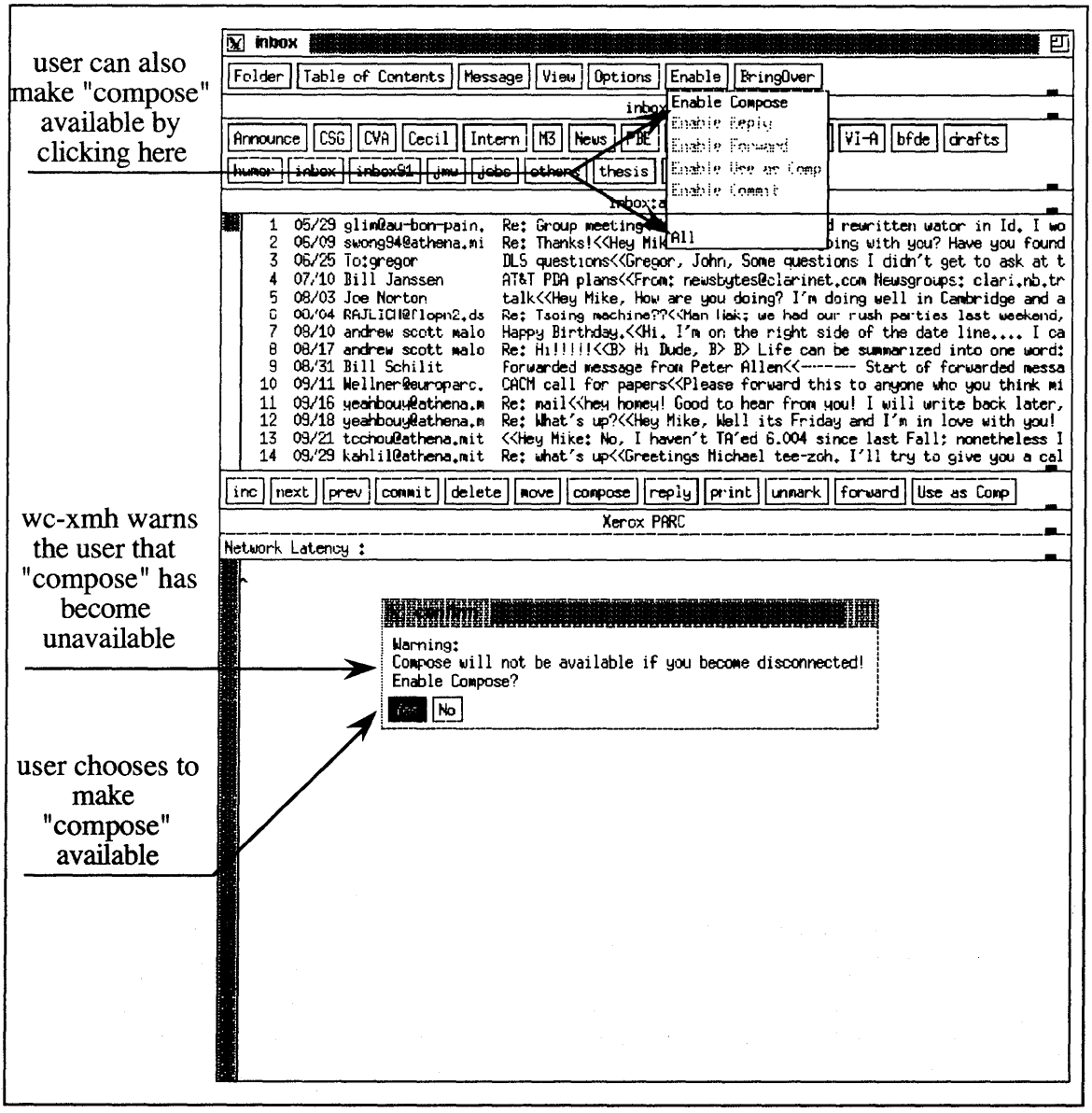

Figure 2.7a - Users are notified if an important feature may become unavailable

2.1.6 Discussion

At first glance, it appears that the graying out of buttons and the availability indicators for messages can be implemented without any special system support. For example, a disconnected application can find out what does not work by pretending to click on every

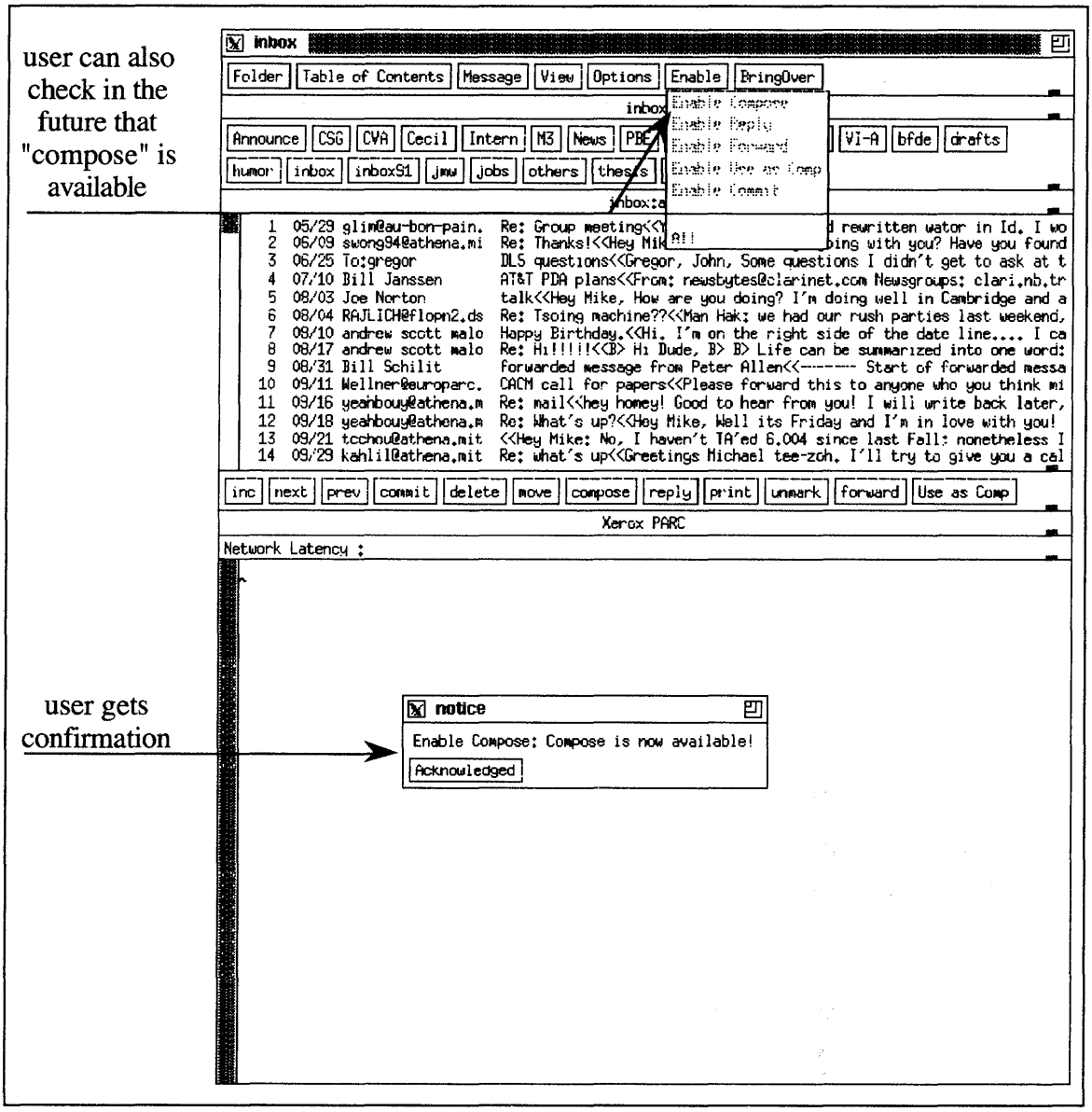

Figure 2.7b - User gets immediate feedback after choosing to make "compose" available

button or menu and seeing if a network timeout error occurs. There are two problems with this approach. First, graying out what will become unavailable (and tri-state buttons) cannot be implemented by the above technique. Second, there may be hundreds of buttons, menus and messages in wc-xmh and simulating a click on each one periodically is both inefficient and tedious.

Wc-xmh's features illustrate some generally applicable ideas for adaptive user interfaces. The idea of graying out buttons and menus can be used in any application with a graphical user interface. Availability indicators are generally applicable for programs which let the user select from a list of possibilities, such as NetNews readers and directory browsers. "Enable" menus are useful for any application to assist the user in planning for future disconnections. For portables running on batteries, a thermometer similar to wc-xmh's network latency indicator showing the amount of battery power remaining would be useful. Tri-state buttons can be generalized to have different colors indicating an estimate of the expected response time if that button is clicked. For example, the "BringOver Folder" button may be colored red because copying all those messages over a slow network can take a long time, while buttons for low latency operations like reading a cached message are colored green.

2.2 Related Work

2.2.1 Disconnected Operation in Coda

Coda [Kistler] [Satya90] also provides disconnected operation using cached data. The key difference between our approach and Coda is that Coda does not change the file system interface and provides no special support for unplanned disconnections. Coda assumes strong connectivity for normal operations. Hoarding allows users to give explicit hints to the cache manager by prioritizing files, but it is less effective for unplanned disconnections than voluntary disconnection. We assume an intermittent network for normal operations and provide full support for both voluntary and involuntary disconnected operation. Caching is transparent to the application in Coda, so applications which depend on the cache for availability cannot predetermine what works

and what does not because the content of the cache is unpredictable. For example, applications using Coda cannot implement features like graying out buttons which are currently unavailable or will become unavailable. When a Coda user is involuntarily disconnected, he must click on every button in order to determine what works and what does not (assuming the application does not hang).

Hoarding in Coda is equivalent to our Hinting mechanism except the cache manager gets hints from the user rather than the application. The user creates a Hoard Profile (sometimes with the help of a trace program) which prioritizes all the files he may use. The main problem with Hoarding is the complexity it places on the user - he must understand the internal dependencies of the application. It is up to the user to ensure that the resources he needs for disconnected operation are made available by keeping the Hoard Profile current and invoking Hoard Walks just before he disconnects. Creating a Hoard Profile is cumbersome because tracing does not always produce the complete dependencies of an application's features. The user does not have fine grained control over what features are made available unless he understands the precise dependencies of each of the application's features. In our system, the user manipulates application level entities like messages, appointments and buttons rather than system level entities like files. The user gets direct feedback from the application about what is available and what will be available. Callbacks enable applications to notify the user if vital features become unavailable. Hoarding in Coda can be implemented as an ordinary application in our system using Hinting.

2.2.2 Adaptive Applications

Schilit's current thesis work [Schilit] uses adaptive applications to address the problem of dynamic system reconfiguration. Although his motivation is similar to ours, his emphasis is different. This thesis investigates the separation of system interfaces and definition of new abstractions, while Schilit's work focuses on mechanisms for communication between applications and system services. He uses a database, the Environmental Database, to maintain attribute-value pairs as a general interprocess communication mechanism. The Localization Manager binds applications' callback requests to persistent queries in the Environmental Database. Unlike our implementation which provided a direct channel of communication between each application and its system services, environmental events in Schilit's system go through two levels of indirection. For an application to be notified about an environmental event, the event must first be reported to the Environmental Database, trigger a query which sends a callback to the Localization Manager, which then notifies the application. Although our experimental system could have been implemented with Schilit's mechanisms, we chose a more direct approach to avoid the delay and race conditions possibly associated with the extra indirections.

2.2.3 File System For Mobile Computing

Tait's current thesis research [Tait] focuses on the tradeoff between consistency and performance in distributed file systems for mobile computing. His file system interface provides two read operations, a strict read which has high synchronization costs and an inexpensive loose read. He exports no other interface for cache or consistency control, and provides no support for disconnected operation.

2.2.4 Application Specific Virtual Memory Management

Allowing applications to control aspects of virtual memory management, such as pinning a page in physical memory, has been implemented in many operating systems [McNamee] [Young87] [Young89] [Cheriton]. The V++ kernel [Harty] support for application controlled external page-cache management is the most recent attempt at overcoming the inadequacies of the conventional "transparent" virtual memory model. Using the abstraction of a page frame cache provided by the kernel, the application can monitor and control the amount of physical memory it has available for execution, the exact contents of this memory, and the scheduling and nature of page-in and page-out. The idea of exposing the virtual memory system to sophisticated applications is similar to our notion of providing two separate interfaces for cache management. V++ allows the application to explicitly control most aspects of how its physical memory is managed. By contrast, our approach is more conservative. We allow the application to influence, but not control, how the file cache is managed. This is because we are using the same technique to solve different problems. External page-cache management in V++ caters for the desire of sophisticated applications whose memory requirements are almost unbounded, such as large simulations or data base systems, to better mask the cost of page faults. Our system improves the usability of applications sharing a file cache by allowing the user to see the consequences of disconnectedness through the context of the application. We were reluctant to give applications explicit control of the file cache because we felt that fairness is important since the portable computer's disk capacity is still fairly limited. In addition, we designed our interfaces so that the desired features can be implemented without introducing unnecessary complications to the application, such as requiring applications to explicitly manage their own cache. Although V++ provides a default memory manager, the application is exposed to the complexity of implementing a custom memory manager even if the paging policy it needs is just slightly different. This

is an important consideration because we expect even ordinary applications such as the mail reader and calendar manager to utilize the support we are providing.

2.2.5 Exposing Abstractions with MetaObject Protocols

Designing programming languages using metaobject protocols [Kiczales] [Rodriguez] is based on the notion that limiting a programmer to using pre-existing implementations (i.e. compilers) as black box abstractions is artificially restrictive; a programmer should be able to, and sometimes needs to, augment the functionality provided by these implementations, without being exposed to arbitrary or irrelevant implementation details. Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation.

At first glance it may appear that Property Specifications bear no relation to metaobject protocols. In fact, our idea of Property Specifications for system services was inspired by the metaobject protocol approach to language design. The power and flexibility of the metaobject protocol originates from exposing traditional black box abstractions in structured ways. Property Specifications give applications the power to operate in an unpredictable computing environment by exposing traditionally sacred black box abstractions such as caching in structured and controlled ways. The common thread linking our work to the metaobject protocol is the notion of designing an abstraction for exposing abstractions. Adding Property Specifications to an existing operating system is like adding windows and knobs to a black box: the application can choose to look into the windows and turn the knobs when the need arises. Adding windows and knobs is better than replacing the black box with a glass box because applications are hidden from irrelevant details of the implementation.

Chapter 3

Programming Models for Application Splitting

The primary motivation for splitting an application is that frontend machines, such as portable computers and display terminals, are often limited in storage and computational capabilities. The desire is to distribute some of the application's computational and data accessing load to more powerful computers on the backend, i.e. a machine (or machines) at the other end of the network. There are many existing and proposed programming models for writing split programs. Broadly speaking, they split the application either at the user interface level or the data access level. Window systems such as X [Scheifler] and NeWS [SunNeWS] [Gosling] provide abstraction boundaries which allow an application to be cleanly split at the user interface level. Similarly, file systems and database interfaces allow clean splits at the data access level. This chapter surveys these programming models and compares their relative merits with respect to the following operating environment constraints:

- Network Reliability - frequency of voluntary and/or involuntary disconnections;
- Network Bandwidth - how much bandwidth is available and how is it shared?

- User Visible Latency - how response time is affected by network latency and availability;
- Frontend Capabilities - compute power, memory and disk capacity;
- Programmability - how easy is it to program, debug and tune applications?
- Flexibility and Adaptability - are programs able to leverage off new resources when they become available? (e.g. improved connectivity);
- Application Migration - can application context be preserved across different instances of the application the user is running on different machines?
- Cost of Data Synchronization - synchronizing cached data and preserving consistency across network partitioning.

3.1 Splitting at the User Interface Level

The main motivation for splitting applications at the user interface level is to accommodate computationally limited frontend machines. The application is partitioned into a user interface (UI) engine and a data processing engine, running on the frontend and backend respectively. The data engine is optimized for information access while the UI engine reduces user visible latency by handling UI events on the frontend. When the network is slow, the goal is to split the application in a way which minimizes the communication between the UI engine and the data engine.

Window systems such as X [Scheifler] provide good abstractions to split applications cleanly. X has four software abstractions: the X server, Xlib [Nye90a], UI Toolkit (Xt and widget set [XtIntrinsics] [Asente]) and the application, as shown in Figure 3.1. The X server controls the display and directs input events to the appropriate X client. Xlib is the programming interface to the X wire protocol [Nye90b] and the raw windowing

system, but is too primitive as an application programming interface. UI Toolkits such as Xaw[3] [XtIntrinsics], Xm[4] [Heller92], and XView[5] [Heller91] address this problem by providing UI building blocks (widgets) such as scroll bars and menus, and allows the application to bind callback procedures to widget activities.

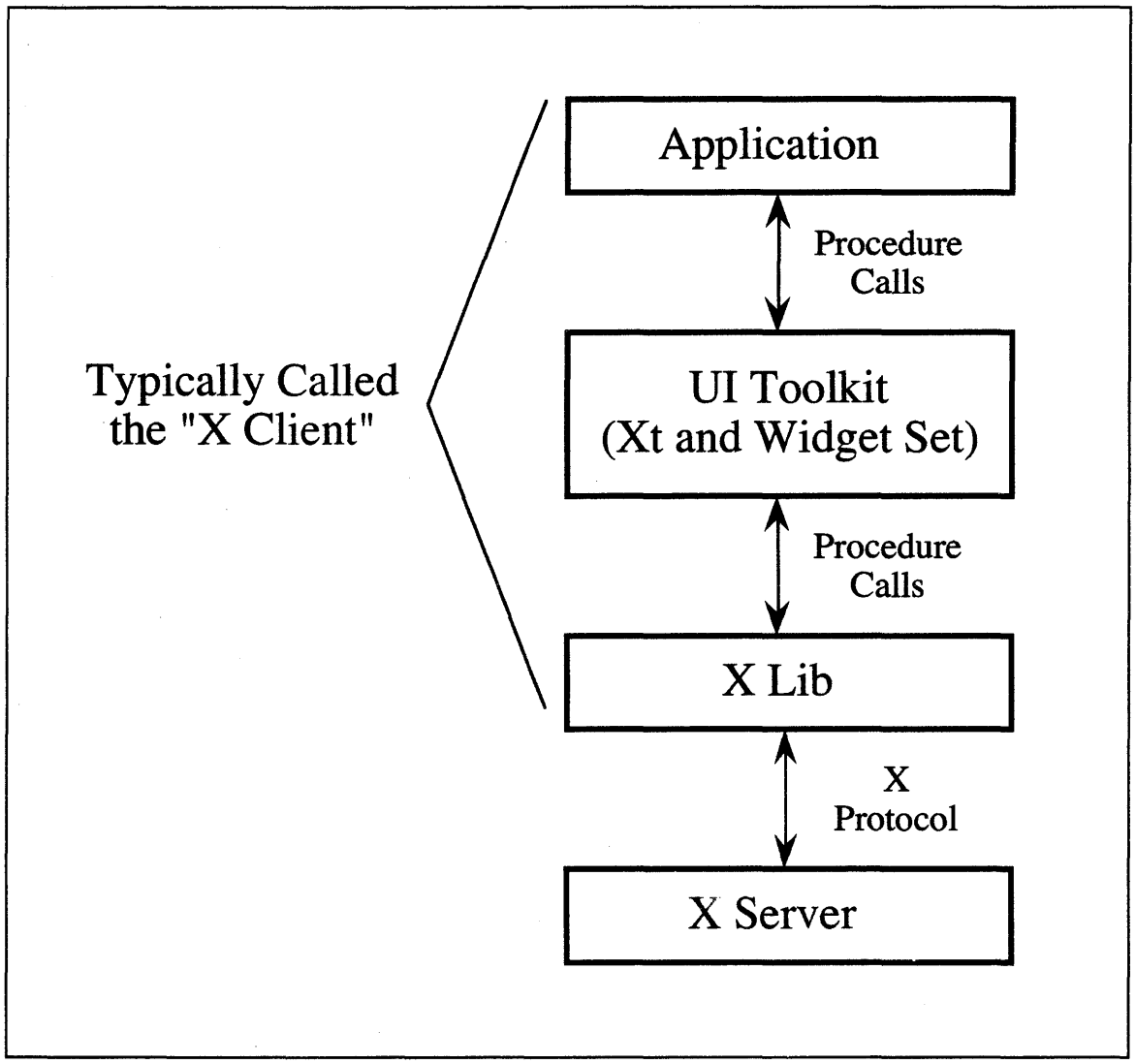

Figure 3.1 - The X system architecture

[3]Athena widget set.
[4]Motif widget set. Based on the OSF/Motif user interface style guide.
[5]Openlook widget set. Based on the Sun/AT&T Openlook user interface standard.

3.1.1 XRemote / LBX

LBX [Fulton] is an emerging standard for running X over telephone lines and other low bandwidth channels based on techniques pioneered by NCD's XRemote™ [Herbert] [Cornelius]. LBX squeezes the X protocol streams from various applications using techniques such as caching, delta replacement, and compression prior to transmission. Backend applications communicate with an LBX "proxy" which appears to be a normal X server running on the backend network. But instead of controlling a display, the proxy converts the X protocol stream to an LBX protocol, and sends it over the low bandwidth link to the real X server (which understands the LBX protocol) on the frontend machine. The LBX architecture is shown in Figure 3.2. For LBX, splitting occurs at the wire protocol level.

LBX reduces the bandwidth requirements of X applications in two ways. First, converting X protocol packets into LBX packets reduces the size of the packets. The conversion process eliminates inefficiencies in the X protocol, reencodes packets more efficiently when possible (e.g. images), replaces packets that can be more efficiently represented as changes against previous packets with their deltas, and compresses the result before transmission. Second, the LBX proxy reduces the number of packets transmitted by caching previous answers to X requests and replying client queries directly. This technique allows the serial line to be bypassed in many cases when different X clients request for the same information from the X server, e.g. font metrics and keysym tables. However, it is worth noting that caching does not reduce the number of roundtrips required for the X client to respond to user input, such as inverting a button after it has been clicked. Thus LBX does not significantly improve the user visible latency for interactive activities.

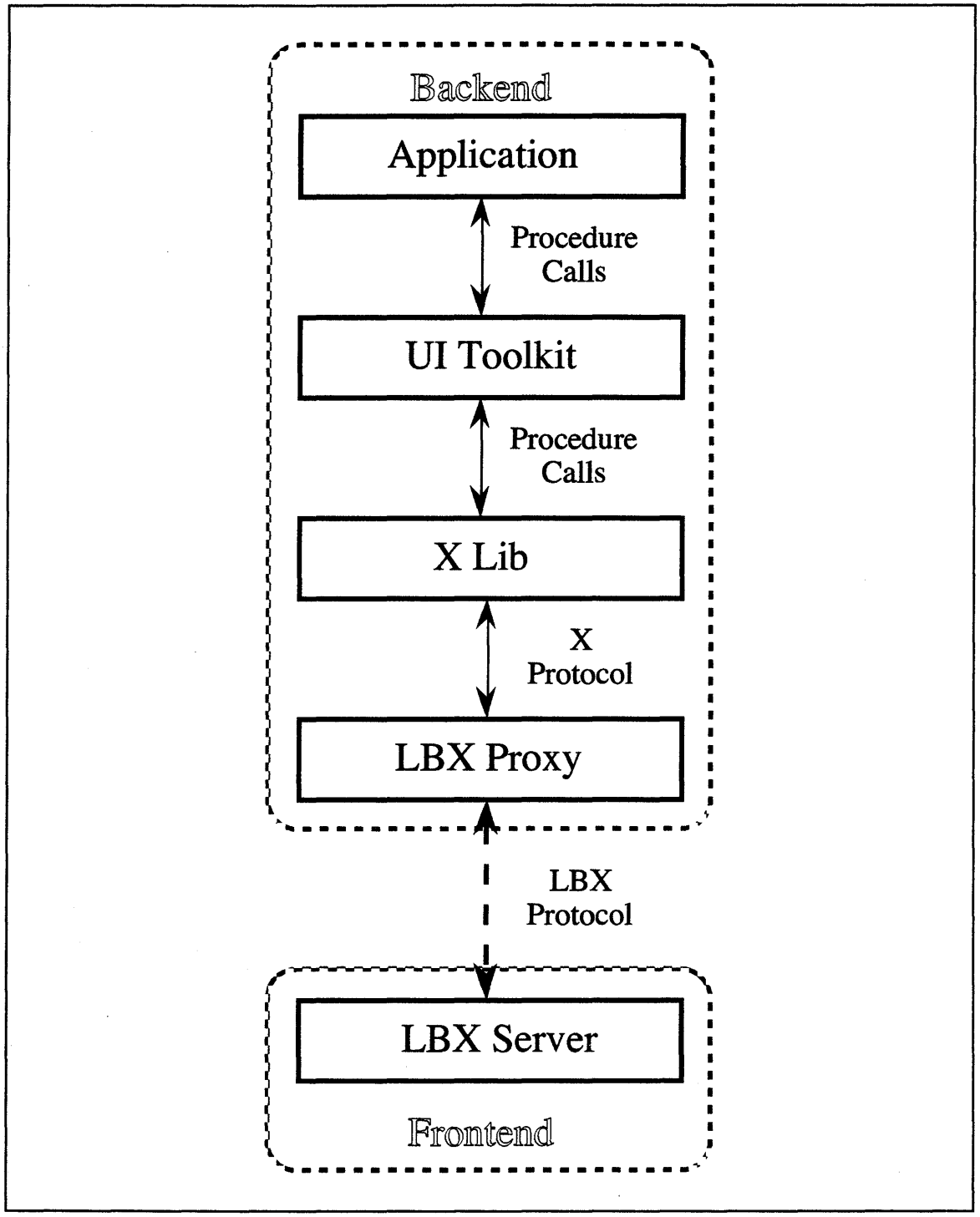

Figure 3.2 - LBX supporting X applications over low bandwidth connections

Besides dramatically reducing the bandwidth requirements of X, one of the other advantages of LBX is its ability to support low end frontend machines such as X terminals. A related approach, the Split Server approach, supports even less powerful frontend machines by splitting the program across the X server. The X server runs on a backend host and sends escape sequences to a frontend graphics terminal which provides only rastering and input handling. This is illustrated in Figure 3.3. This approach may be practical for very small computers such as the ParcTab [Adams], allowing them to be used as mobile I/O devices. But its usefulness is limited because it does not allow for any X application, not even the window manager, to run on the frontend.

Both LBX and the Split Server approach reduce the bandwidth requirements of X applications in an application independent way. But they give the application programmer no flexibility over how the program is split, which is both an advantage and a disadvantage. On one hand, the application programmer is freed from having to hardwire the notion of a slow network into his program. On the other hand, the application programmer cannot improve the performance for those applications which need special features like local button inverting or rubberbanding of windows. The main disadvantage for both approaches is that if the network is intermittent, the application will hang when the network fails because it runs on the backend.

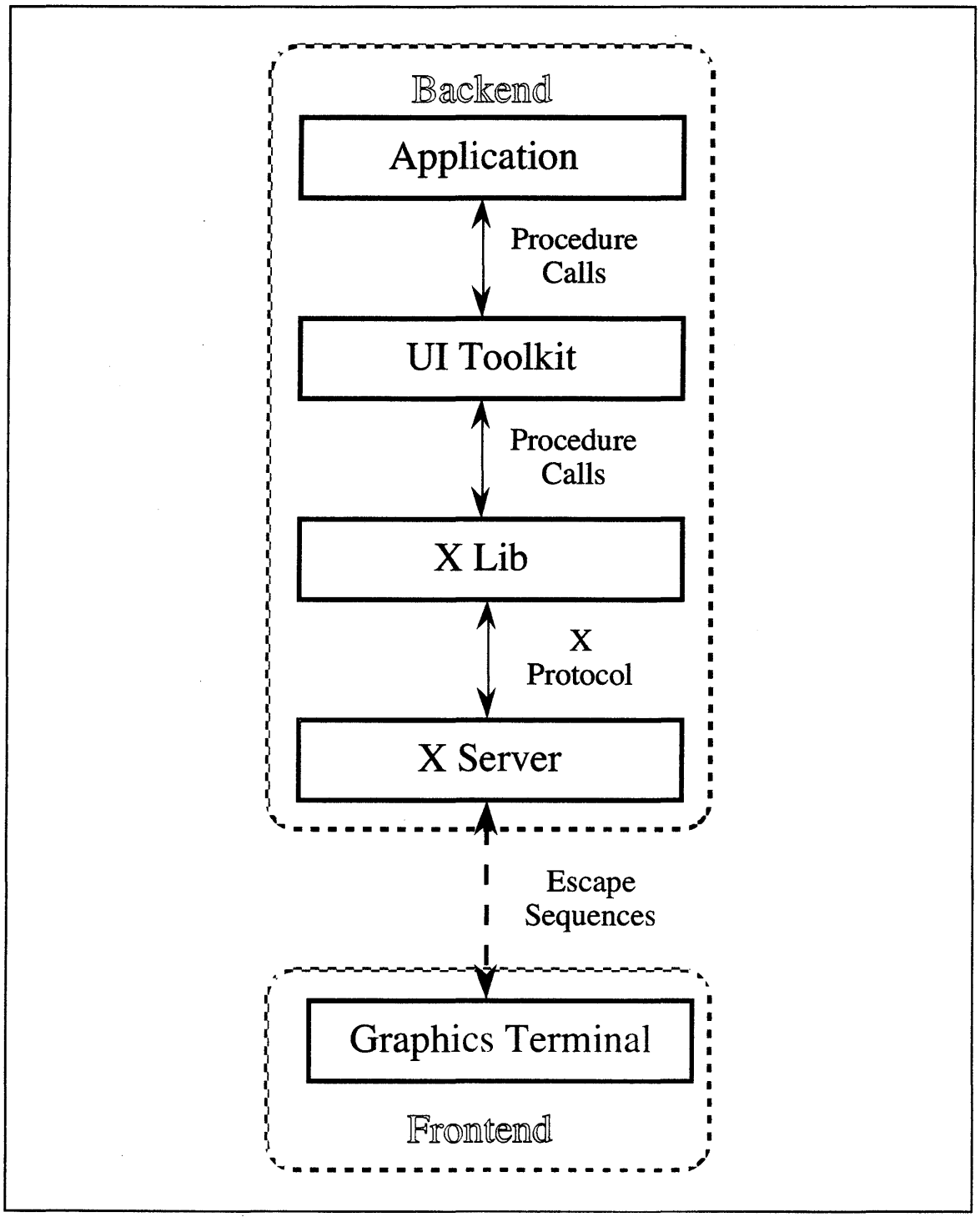

Figure 3.3 - Split Server approach for supporting X applications over low bandwidth connections

3.1.2 Split UI Toolkit

Another idea is to split the program at the UI Toolkit level such that all or part of the UI Toolkit runs on the frontend, as depicted in Figure 3.4. We are unaware of any existing systems that use this approach. It is based on the observation that after setting up the widgets, the UI Toolkit and the application code communicates only at a very high level, i.e. callbacks from widgets or procedure calls to manipulate widgets. Immediate UI activities such as inverting buttons, highlighting selections and scrolling are all internal to the widgets and are done on the frontend. One complication for this approach is that the application and toolkit run in separate domains. Thus a mechanism for sharing data between the toolkit and the client is needed since most toolkits invoke client callbacks with pre-registered pointers to mutable data. The main disadvantage of this approach is the same as for XRemote and LBX: the application does no useful work when the UI Toolkit is disconnected from the backend.

The communication bandwidth is reduced because the UI engine communicates with the backend via callbacks only when there is real work to be done, rather than for every I/O event as in XRemote or LBX. User visible latency is reduced by manipulating widgets locally on the frontend. The application programmer's interface is unchanged and there still is no explicit way to control how the program is split. Another disadvantage of this approach is that applications are restricted to using a predefined set of widgets. Supporting customizable widgets introduces complications as it requires the UI Toolkit to allow dynamic extensions such as those supported by NeWS [SunNeWS] [Gosling] and Tcl [Ousterhout]. They are discussed in the following section.

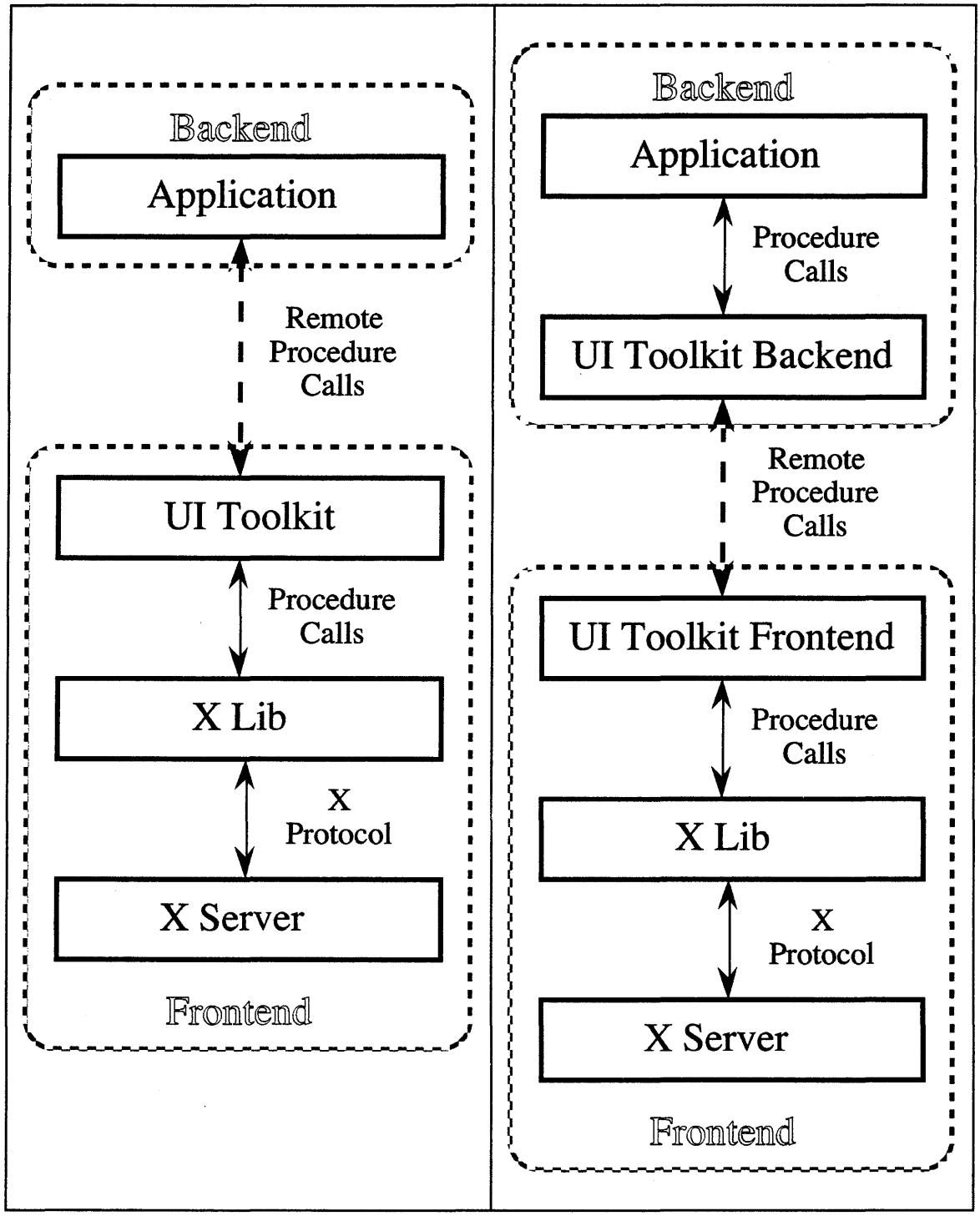

Figure 3.4 - Two ways of application splitting at the UI Toolkit level

3.1.3 Extensible Servers

It is difficult for the UI Toolkit or the display server (X or NeWS server) to provide an interface that is suitable for all applications. NeWS and Tcl are motivated by the desire to support application specific customizations to the window system server. They provide mechanisms for the application to download programs into the frontend server to customize existing widgets, define new widgets, and perform unusual tasks like "rubberbanding" locally. Both systems are targeted towards dividing a client program into two sections: one to perform the basic computation executing on the backend, and one to provide windows or graphics and is interpreted by the server process. Figure 3.5 illustrates their architecture. The mechanisms provided by NeWS and Tcl are flexible enough to allow the program to be arbitrarily partitioned with the restriction that the partition is fixed at runtime. This is different from systems supporting dynamic process migration [Jul] [Douglis] which can change the partition during runtime by moving the program's execution context between the frontend and the backend. We do not discuss dynamic process migration systems in this chapter because their applicability to low bandwidth or intermittent networks is not well understood.

The flexibility offered by NeWS and Tcl is both a strength and a weakness. On the one hand, the programmer can partition the program in any way he chooses. On the other hand, he has no reliable algorithms for deciding what is the best partition. The program is split explicitly by writing it in two parts, making it difficult for the programmer to iterate his design and experiment with different splits. In addition, NeWS has the disadvantage of requiring frontends to be powerful enough to support an elaborate dynamic environment including an interpreter, light weight process management, a UI Toolkit, and automatic memory management.

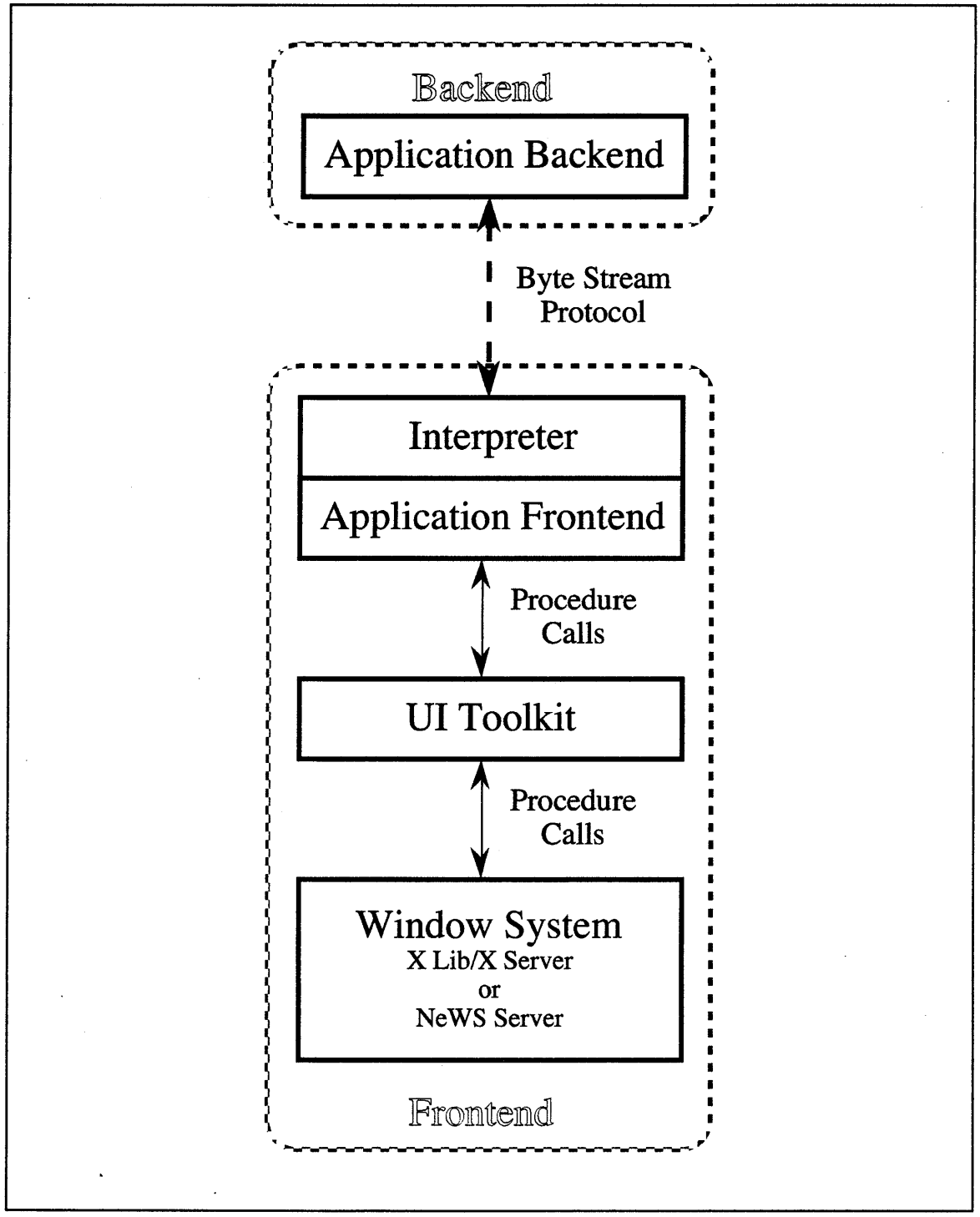

Figure 3.5 - Client Extensible Window System Servers

3.2 Splitting at the Data Access Level

The primary motivation for us to split the application at the data access level is for increased autonomy. The application runs on the frontend and can respond to disconnections in user friendly ways such as graying out buttons. We define the level of autonomy to be the application's ability to operate disconnected. At one extreme, applications for PenPoint [Novobilski], Macintosh [MacOS], MS DOS [Jamsa], MS Windows [Petzold], and MS Windows for Pen Computing [Ward] operating environments use local file systems and can operate completely autonomously. But accessing and sharing large amounts of data are more difficult for these applications. At the other extreme, an application using a distributed file system such as NFS [Sandberg] which does no file caching, is not very useful when disconnected because it has no access to data. Coda [Kistler] [Satya90] provides all the benefits of a distributed file system as well as autonomy. Applications in Coda can operate disconnected provided the files they need are available in the frontend's file cache. The increased autonomy comes at the expense of the cost of synchronizing multiple copies of files.

Autonomy also makes application migration difficult. Application migration is where an application's context is moved from one machine to another, e.g. the user might bring the particular configuration of buffers from the Emacs instance running on his office workstation to his home computer. If the frontend is stateless, migration is easy because only the user interface needs to be moved. We increase the application's level of autonomy by moving part or all of its state and data onto the frontend. Application migration now requires moving an application's dynamic state from one machine to another, a non-trivial task in any network environment.

The following two sections describe two ways to split the application at the data access level.

3.2.1 Remote Evaluation

Remote Evaluation is the ability to evaluate a program expression at a remote computer. Remote Evaluation is designed to support the construction of distributed applications that examine significant amounts of data stored at a remote server, but ultimately return a compact answer to the client. As shown in Figure 3.6, an application can execute entirely on the mobile frontend and use Remote Evaluation when it needs to access remote databases or perform heavy computation. For servers which support remote evaluation, these requests go directly to the servers; otherwise an intermediary proxy is required to translate the Remote Evaluation expressions into corresponding RPCs to database or file servers. Examples of systems supporting Remote Evaluation include REV [Stamos], NCL [Falcone], and NeFS [SunNeFS]. REV was an experimental system built on top of an RPC mechanism in a Lisp environment. The Network Command Language (NCL) is a Lisp like language which enables heterogeneous machines to communicate by programming one another. NeFS was an experimental distributed file system where clients can execute PostScript programs on file servers. None of these systems was ever in wide use.

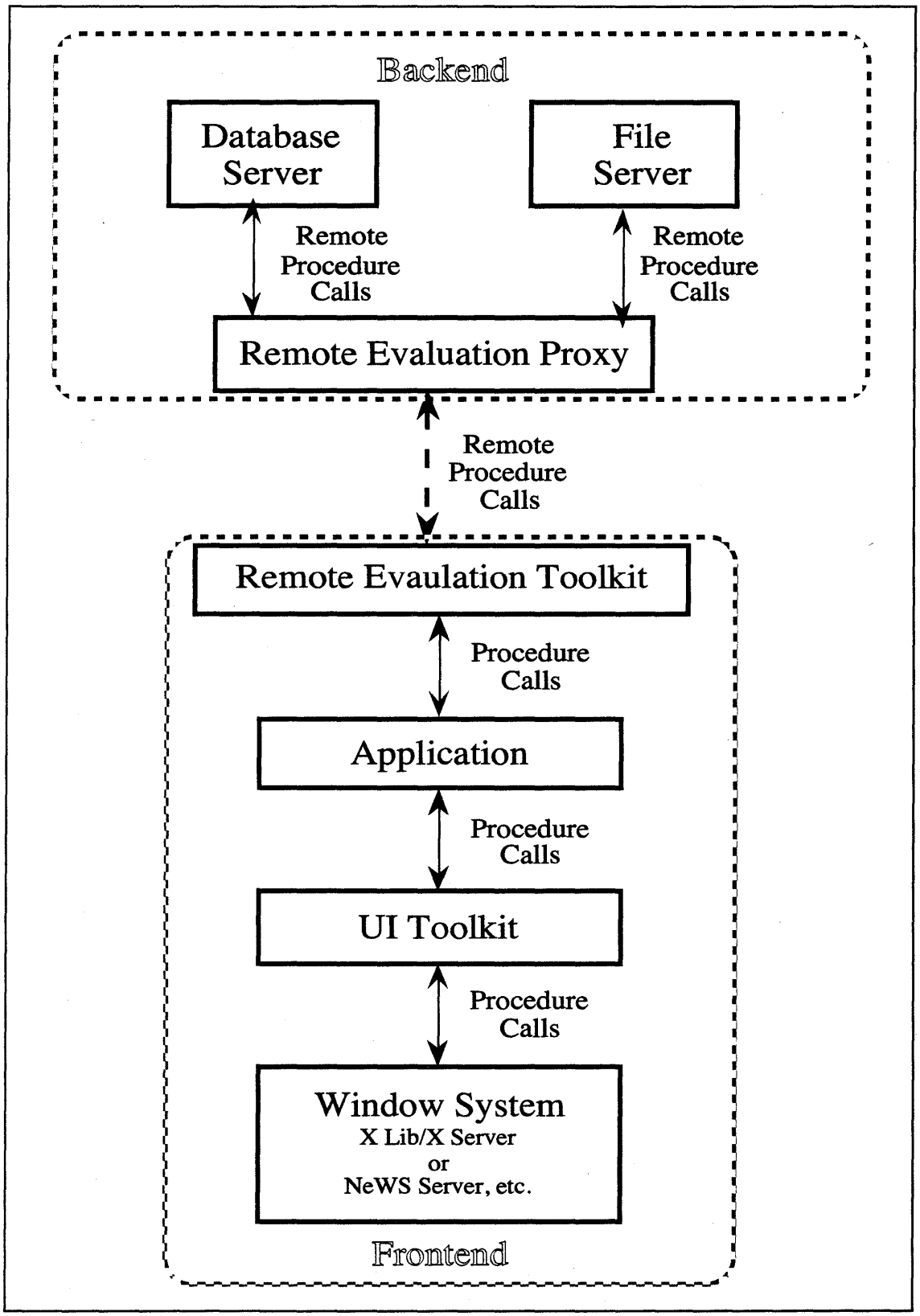

Figure 3.6 - Remote Evaluation for Mobile Clients

Remote Evaluation reduces the application's bandwidth requirements by processing data remotely. It reduces user visible latency by performing all UI and basic computation locally on the frontend. Besides requiring the frontend to be a more powerful computer, the problem with using Remote Evaluation lies in the difficulty in deciding when it is more efficient to use Remote Evaluation rather than copying the data to the frontend and computing locally. For example, a file frequently searched with "grep"[6] should be copied to the frontend instead of using Remote Evaluation for every grep, but only if the file is not too big. A mail program that uses grep to generate a summary of all the message headers would be wise to use Remote Evaluation to avoid copying all the messages to the frontend. But it may choose to copy all the files if the messages are likely to be read soon.

3.2.2 Splitting at the File System Level

Completely standalone applications using only local file systems do not pose any major challenges for disconnected operation. They are less interesting because portable computers are seen as standalone personal computers, rather than entry points to a distributed information and communication system. Splitting the application across a non-caching distributed file system provides access to distributed data only when the frontend is connected. Neither of these models are particularly interesting in our discussion of graceful disconnected operation because in one there is little to be done while in the other, very little can be done. We will focus our discussions on Coda's programming model because it provides both autonomy and distributed information access. We call Coda's programming model the Autonomous model, as illustrated in Figure 3.7.

---

[6]grep - a UNIX command which searches through a list of files for all instances where a matching string or regular expression occurs.

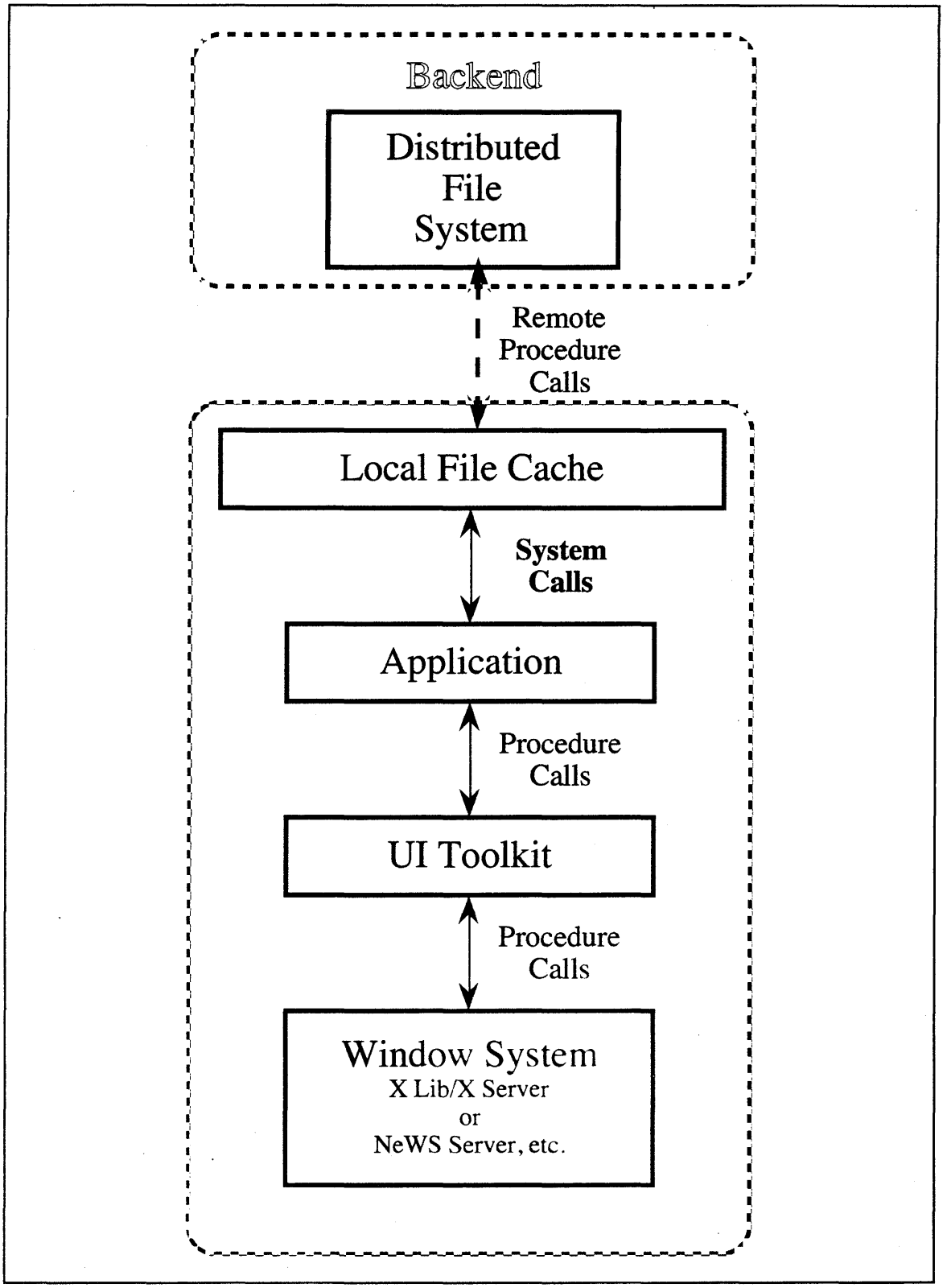

Figure 3.7 - The Autonomous programming model

In the Autonomous model, the application runs entirely on the frontend device and accesses data through a caching distributed file system or Remote Evaluation at remote database servers. The main advantage of Autonomous applications over traditional client/server models is the ability to operate disconnected using cached data, thus reducing, but not eliminating, the application's reliance on the network for availability. The Autonomous programming model is simple because the program is not explicitly partitioned. The system can automatically translate data access requests into file transfers, RPC or Remote Evaluation.

The main disadvantage of the Autonomous programming model is that data consistency problems are amplified by the intermittent environment. Consistency protocols based on callbacks [Satya85] depend on strong connectivity for timely notification. Time based mechanisms such as Leases [Gray] require periodic negotiations between the frontend and backend file server, and is inadequate if disconnections can be lengthy and unplanned. Data consistency has not been a major issue in today's distributed file systems because write sharing of files is uncommon [Satya85] [Kistler] [Nelson88]. Mobile computing is likely to worsen this problem. Each user is likely to use several machines, such as a palmtop, a notebook, and desktop machines in the office and at home. The user's working set of files is likely to be cached by all of these machines, some of which may become disconnected. This is a difficult problem whose solution may lie in the observation that the common case is when files are write shared by the same user. In general, addressing the consistency problem requires an understanding of the trade-off between the level of consistency and the cost of reconnection and consistency protocols. The desired solution depends on the degree of sharing, the frequency of disconnections, and whether the disconnections are planned or unplanned.

An additional disadvantage of the Autonomous model is that frontends must be powerful enough to run the applications locally. We do not see this as a major issue based on the observation that portable computers will continue to be used as personal communicators and information organizers. Typical applications such as editors, mail/news readers, file/directory browsers and calendar managers are not compute intensive.

3.3 Conclusion

Table 3.1 summarizes our discussions in this chapter: the programming models are in rows and the design space is in the columns. It is not intended to be an exhaustive set of design choices but merely a framework for understanding the tradeoffs which confront designers of split programs. A "+" under the bandwidth and latency categories means that the application's bandwidth requirements and user visible latency are significantly improved as compared to the X protocol [Nye90b]. For disconnected operation and application migration, "+" means these features can be supported easily. A "+" under programmability means application programming is relatively easy, under data consistency means relatively cheap mechanisms can ensure consistency, and under frontend capability means frontends do not need to be powerful computers.

There is a pattern is Table 3.1: effective disconnected operation comes at the expense of application mobility and increased complexity in dealing with data consistency. Similarly, the computational and storage requirement for the frontend machine also increases. Selecting a programming model involves understanding the tradeoff between autonomy and consistency in the context of constraints in the operating environment. For example, for a company which provides dialup database services, it may be advantageous to split the application at the window system level because telephone lines

are relatively reliable. This also allows the company to maintain control over all the application software. Performance for interactive applications can be enhanced by putting a UI toolkit on the clients' machines[7].

	Bandwidth	Latency	Disconnected Operation	Frontend Capability	Programmability	Data Consistency	Application Migration
XRemote / LBX	+	-	-	+	+	+	+
Split UI Toolkit	++	+	-	+	+	+	+
Extensible Servers	++	+	-	-	--	+	-
Remote Evaluation	++	+	-	--	-	+	--
Autonomous	++	+	+	--	+	--	--

Table 3.1 - Characteristics of Different Split Programming Models in a mobile computing environment

Our strategy for reducing the application's network bandwidth requirements and user visible latency is by reducing the size and number of user interface level packets such as mouse events. We accomplish this by managing the user interface on the frontend computer as much as possible. This strategy should work well when we split the application at the user interface level. But when the application is split at the data access level, it is unclear whether the network bandwidth the application needs is reduced. The wireless network can be easily saturated if the file cache thrashes or if a query evaluated remotely at a database server returns a lot of hits. Improvements in the cache manager and Remote Evaluation system may reduce the effects of these problems, but these issues are not considered in this thesis.

Programmability is listed in Table 3.1 because the usefulness of a programming model in the real world is in part affected to how easy it is to write applications. The tradeoff here

[7]This example came from discussions the author had with FactSet Data Systems Inc., a Connecticut based company specializing in online financial information systems.

is simplicity versus power. Applications are split implicitly at the window system or the file system level with the XRemote, Split UI Toolkit and Autonomous models. The programmer does not have the flexibility to control how an application's computation is split, but they are easy to program and debug. Extensible server based systems like NeWS and systems which support Remote Evaluation allows application specific code to be executed remotely. Programming such systems is harder because the program is split explicitly. Experimenting with different splits is difficult because the programmer must make substantial modifications to the program.

3.3.1 The Programming Model for this Thesis

Given our stated assumption of an intermittent environment and our goal of supporting graceful disconnected operation, Autonomous is clearly the best programming model. The main advantage of Autonomous applications is their high availability, remaining at least partially functional during disconnections. An additional advantage is that UI events are handled on the frontend, thus decoupling UI response time from the network round-trip delay. We feel that the advantages of the Autonomous model outweigh the problems associated with cache consistency. Although the data consistency and application migration problems are very real, we chose not to address them within the scope of this thesis because we believe that graceful disconnected operation is the first order problem for mobile computing.

Chapter 4

Property Specifications

Much of the effort in building systems over the past two decades has been directed at building system service interfaces which provide a transparent network to the application. Software is layered such that "irrelevant" details such as variations in latency are hidden from the application. The underlying philosophy is that these abstractions reduce complexity and improve programmability. Property Specifications seems to be at odds with this philosophy as they allow applications to be actively engaged in preventing and handling errors. Section 4.1 of this chapter discusses the philosophical justification for separating system interfaces into Functional Specifications and Property Specifications. It also discusses how Property Specifications can be used to build both application specific features and application independent tools for graceful disconnected operation.

Section 4.2 defines and elaborates the mechanisms our new system abstraction should provide. In Section 4.3, we show how to apply these mechanisms to system services. The Property Specifications for a caching distributed file system and a network service interface are presented. Section 4.4 explores some subtle issues in the semantics of the

Property Specifications we had designed. In Section 4.5, we generalize Property Specifications beyond the context of mobile computing, and present a Property Specification for a virtual memory interface.

4.1 Motivation

4.1.1 Separation of Functional and Property Specifications

There are two problems with rigid, transparent system interfaces. First, abstractions often hide the power of the underlying system. For example, Birrell and Nelson [Birrell] found that implementing RPC using the more primitive Unreliable Datagram Packet interface was twice as efficient as using the Reliable Datagram Protocol. Second, the effort invested in building existing applications gives old interfaces tremendous inertia against change, even when the underlying technology they were designed for have changed dramatically. The mobile computing environment is fundamentally different from today's distributed computing environments, where network connectivity is usually reliable. Unfortunately, to date we have not looked very hard at designing new system abstractions designed specifically for intermittent computing environments. The desire for backward compatibility often forces programmers to work with sub-optimal abstractions. Building systems on top of bad abstractions is like putting in screws with a hammer: it takes a lot of effort to attain an unsatisfactory outcome.

In the real world, there is one principle more important than "make it clean": "make it work." In practice, real world applications need to monitor and handle errors and environmental changes regardless of the level of support the system interface provides. For example, Automatic Teller Machines (ATM) must continue to provide service even

when they become disconnected or the central database servers are lost[8]. System designers have been faced with the dilemma of having to reveal some details in the system interface which is irrelevant for some applications but necessary for others. By isolating the underlying properties of the system into a separate interface, we achieve the best of both principles: "making it clean" with Functional Specifications while "making it work" with Property Specifications. Another advantage of providing two separate interfaces is that today's applications can be ported *incrementally*, i.e. existing applications like xmh will still work using only the Functional Interface and extra programming is only necessary if new features are desired.

4.1.2 Using Property Specifications to Provide Application Specific Support

Property Specification is a step towards exploring the continuum in system abstractions from application specific to application independent support. The idea is based on the observation that collectively, the application and the operating system know precisely what the user needs to know: will feature M in application A work. This is because the application knows about the services and resources each of its feature needs while the operating system knows the availability of those services and resources. Hence close collaboration between the system and the application is required for graceful disconnected operation.

Property Specification provides a structured way for efficient information exchange between the application and the system. It fosters a programming model where the

[8]A recent snow storm destroyed the ATM network's database servers in New Jersey. Their backup system in New York City was unavailable because it was already running as a backup system for other servers lost during the World Trade Center bombing. Although the central information service in New Jersey was not restored for another 2 weeks, ATM service was not disrupted because the software on individual ATMs switched to disconnected operations mode. Source: Professor Jerry Saltzer, saltzer@mit.edu.

application uses its information about the environment to prevent errors, the system has the responsibility of detecting errors, and the application has the option of overriding the system's default error handlers with application specific ones. We model changes in the operating environment with **environmental events**, which an application can elect to receive. For example, the discovery of a printer in the vicinity of the user might generate an environmental event for the print spooler[9]. Our experience with programming event driven applications gives us confidence that environmental events is a powerful and elegant abstraction.

4.1.3 Using Property Specifications to Provide Application Independent Support

Wc-xmh only demonstrates how Property Specifications can be used in application specific ways. In fact, Property Specifications also enables a new class of tools which provides application independent support. For example, Coda's Hoarding can be implemented as an application which monitors other applications' file usage patterns and provides a friendly user interface for the user to directly influence the cache manager's decisions. Can we support graceful disconnected operation using only application independent tools? We believe this is unlikely because features such as those described in Chapter 2 cannot be implemented without application specific information. The most we can imagine for an application independent tool is one which intercepts failed system calls and gives the user the option of retrying later when the network is reconnected instead of hanging or crashing the application. This level of help is similar to printing "RPC timed out, retrying..." in the console to give the user the option of either to wait or kill the offending process, as used in the Sprite file system [Welch]. These methods are

[9] The print spooler spools print jobs and puts them in a buffer. When a printer is found nearby, the spooler sends the print job to the printer, perhaps via a wireless network.

much less user friendly than what wc-xmh provides, and are much less desirable because waiting for retries prevents the user from making further progress during disconnections.

4.2 Property Specifications Mechanisms

Property Specifications are different from **Functional Specifications** in that they specify the set of properties a particular system service exports, rather than the functionalities the system service provides. The state of the system's properties represents the operating environment, and environmental events reflect dynamic changes in the system's properties. We define three mechanisms Property Interfaces should provide for accessing, monitoring, influencing and manipulating the exported properties efficiently. They are **Query**, **Notification** and **Hinting**.

- **Query** allows the application to obtain information from system services, for example, querying the file system to find out whether a file is in the cache. On startup, an application queries the system and builds a model for its computing environment. An application can also use Query to update its model or to verify the effects of its actions, such as giving a hint.
- **Notification** lets an application bind callback procedures to environmental events of interest, such as binding a procedure which disables/enables the "get new mail" button to the "change in connectivity" event. The control flow is best described as the system service waking a waiting thread in the application domain. Notification enables the application to monitor and react to environmental changes which affect the availability of its features. The application uses Notification to keep its model of the environment updated.

- **Hinting** enables applications to pass special requests and optional information to influence or customize system services. For example, the application can hint to the cache manager to distinguish resource files from data files so that resource files are less likely to be purged from the cache.

Query and Notification empower the application to interpret the dynamic properties of the system services with respect to its own dependencies, and present the results to the user in user friendly ways. Our experience shows that presenting the state of the operating environment to the user can be of great utility, but users absolutely cannot tolerate an application which does not manage its partial functionality effectively.

Applications communicate specific needs and desires to system services by Hinting. Hinting is like the system providing handles to some of its internal controls so that applications can influence or even customize its behavior. For example, the application can customize the consistency requirements of a file it uses (e.g. write through or write behind.) Hinting in a file system is very useful for voluntary disconnection. A user requests the application to make a particular feature available, and the application hints to the file system that the files needed for that functions should be paged into the cache due to user request. Since the cache is a shared resource, the request is only a hint, but the application is notified of its effects. This is of great utility because the application can provide dependable pre-negotiated service to a voluntarily disconnected user. The user deals with application level entities like folders and features in the context of the particular application rather than system level entities like files. This is an important advantage for user friendliness because the user is hidden from the application's internal dependencies.

4.3 Designing Property Specifications

Query, Notification and Hinting are general and powerful mechanisms that can be applied to a wide range of system services. The actual semantics of these mechanisms depend on the specific properties of the particular system service. In this section, we share our experience in the design of Property Specifications for a caching distributed file system and a network statistics monitor. Since our experience in designing property interfaces is still limited, we present our experimental designs for the reader to draw insights from rather than as an algorithm for designing the best interface.

Our design was driven by the need to balance between the user's requirement for autonomy and predictable performance and the application programmer's desire for a simple and clean system interface. We used a top down process, first creating a list of desired user level features like those described in Chapter 2, from which we extracted the key properties of the underlying system we need to include. Based on the nature of the property, such as how it changes dynamically and whether it should be customizable, we then designed any appropriate mechanisms to access, monitor, influence and manipulate the property. Our design was iterative, it evolved as we gained experience through implementation and use. What we present here is the result of a couple of iterations.

4.3.1 Property Specification for a Caching Distributed File System

The key property specified by our interface in Figure 4.1 is that files are either in or out of the cache. We provide a Query mechanism, `FilesAvailable()`, which allows the application to synchronously inquire the availability of a group of files. `MonitorFiles()` provides Notification, it lets the application to continuously monitor

the paging activities of a group of files with a callback procedure. We define two environmental events, `PagedIn` and `PagedOut`, which encapsulate the property that files are moved in and out of the cache. When one or more files in the group is paged in or out, the callback procedure is invoked by the file system in the application's address space.

`GiveHints()` and `MakeAvailable()` are our Hinting mechanisms which allow the application to influence and customize the cache manager's paging policy. The hint "`UserRequest`" is typically used with `MakeAvailable()` when the files need to be cached due to direct user request, such as for voluntary disconnection. An application also uses `MakeAvailable()` to ensure the availability of the vital resources needed by its features by using the "`AppResource`" hint. `MakeAvailable()` causes the file system to associate the given set of hints with a group of files, and synchronously return the files' availability after attempting to cache them. `MakeAvailable()` needs to be synchronous because the user needs to know whether his request is satisfied. The application can also influence the cache manager's future behavior by associating hints with files using `GiveHints()`. We define three other Hints: "`AllOrNothing`" specifies that the given files are inter-dependent and it is of no value to make only a subset of the files available; "`WriteBehind`" and "`WriteThrough`" lets the application choose the consistency/performance tradeoff for its files. All the hints except for "`WriteThrough`" and "`WriteBehind`" *influence* the cache manager's current and future decisions so their precise effects are *unspecified*. "`WriteThrough`" and "`WriteBehind`" are *customizations* because they have well defined effects on how the file system will manage the given files.

```
INTERFACE FileSystemProperty;

TYPE
     EnvEvent = {PagedIn, PagedOut}; (*Environmental Event*)

     Filename = TEXT;

     Hints = {UserRequest, AppResource, AppData,
               AllOrNothing, WriteBehind, WriteThrough}
     (* Definitions:
               UserRequest - the files are needed due to
          user action
               AppResource - resource and configuration
          files vital for the application's features
               AppData - non-critical application data
               AllOrNothing - files are dependent, caching
          any subset is of no value
               WriteBehind - weak consistency requirement,
          asynchronous paging out OK
               WriteThrough - strong consistency
          (synchronous writes) required
     *)

     CallbackProc = PROCEDURE callback(event : EnvEvent;
          files : ARRAY OF Filename; callback_arg : REFANY);

          (* callback_arg is supplied by the application
               when registers the callback procedure *)


PROCEDURE FilesAvailable(files : ARRAY OF Filename)
                    : ARRAY OF BOOLEAN;
(* Query: the ith boolean is TRUE iff the ith filename in
files is in the cache, and FALSE otherwise *)

PROCEDURE MonitorFiles(files : ARRAY OF Filename;
          callback : CallbackProc; callback_arg : REFANY);
(* Notification: invokes callback in the application's
     address space if one or more of files is paged in
     or out of the cache *)

PROCEDURE GiveHint(files : ARRAY OF Filename;
               hints : SET OF Hints);
(* Hinting: associates hints with all of files *)

PROCEDURE MakeAvailable(files : ARRAY OF Filename;
          hints : SET OF Hints) : ARRAY OF BOOLEAN;
(* Hinting: associates hints with all of files, returns the
resulting availability of files in an array of boolean *)


END FileSystemProperty.
```

Figure 4.1 - The Property Specification for a Caching Distributed File System in Modula-3 [Nelson90]

4.3.2 Property Specification for a Network Statistics Monitor

The packet latency of an intermittent network can vary greatly. The end-to-end latency depends on the level of congestion for the medium and the availability of the network. When the user invokes a function, the latency he experiences is often related to the current network performance, e.g. clicking on a button to read an uncached mail message. In order to provide predictable performance, the application needs to monitor changes in the network latency so it can adapt its user interface accordingly. Existing network interfaces do not provide access to latency information.

Fortunately statistical multiplexing in networks does not result in unpredictable performance parameters. To a first approximation, the expected latency on the next packet is close to the average latency of recent packets. We propose a new system service, the Network Statistics Monitor, which collects performance statistics at the transport layer. This statistics is used as hints for predicting the current network performance. The Network Statistics Monitor exports the property that the performance of the network can vary with time. The Network Property Interface is shown in Figure 4.2. Applications can use `GetLatency()` to get the predicted latency of the network. We allow applications to monitor changes in the predicted network latency by binding a callback procedure to the `LatencyChanged` environmental event using the `MonitorLatency()` call. Applications can register callback procedures for more than one latency range.

At first glance, our use of a latency range appears to be overkill because most applications only need to know if they are connected or not. There are two reasons for using a range instead of a single latency value. First, if a single value is used, the application could be flooded with callbacks if the average latency oscillates around that

value. Second, the definition of connectivity is application specific: the distinction between a slow network and a disconnected network depends on how slow a network the application can tolerate. For example, background printing can tolerate network latencies on the order of minutes but xmh is practically disconnected if the latency is even a few tens of seconds.

```
INTERFACE NetworkProperty;
IMPORT Time;

TYPE
     EnvEvent = {LatencyChanged};   (* Environmental Event *)
     Range = RECORD
               low : Time.T;
               high : Time.T;
             END;
     CallbackProc = PROCEDURE callback(event : EnvEvent;
                    callback_arg : REFANY);

PROCEDURE GetLatency() : Time.T;
(* Query: returns the average network latency in units of
seconds and microseconds *)

PROCEDURE MonitorLatency(threshold : Range;
     callback : CallbackProc; callback_arg : REFANY);
(* Notification: invokes callback in the application's
address space when the average network latency exceeds
threshold.high or falls below threshold.low *)


END NetworkProperty.
```

Figure 4.2 - The Property Specification for a Network Interface

4.4 Subtleties in the Semantics of Query and Notification

There is a subtlety in the semantics of the interfaces given in Figure 4.1 and 4.2. Since paging activities are asynchronous and the cache manager may be serving many applications concurrently, the results from the `FilesAvailable()` and the events

from `MonitorFiles()` are only hints. Those calls give a snapshot of the state of the cache at some point in time between the start and completion of the call, but the state of the cache may well be different by the time the result is returned to the application. This causes race conditions which are especially complex when the application is multithreaded[10].

Let us first consider a single threaded application. All incoming events are queued and handled in turn with an event loop. An application starts up, makes a query on file A and then registers a callback to monitor it, as shown with the pseudo code in Figure 4.3. If A was in the cache at line 1, the button gets enabled in line 2. Now assume A is paged out of the cache before we register the callback procedure in line 3. The button will remain enabled even though A is not available.

```
     ....
1    AisAvailable := FilesAvailable("A");
2    EnableOrDisableButtons(AisAvailable);
3    MonitorFiles("A", EnableOrDisableButtons);
     ...
```

Figure 4.3 - Potential Race Condition in using the FileSystemProperty Interface.

It appears that the problem might be solved by putting line 3 in Figure 4.3 before line 1, as illustrated in Figure 4.4. If A is paged out between lines 2 and 3, the button will be incorrectly enabled after line 3, but a PagedOut event will invoke the callback to disable the button later. It seems to work for the single threaded case. Now let's assume the application is multithreaded, and the callback procedure is invoked before line 3 in a different thread. We see that the fix in Figure 4.4 does not work either: at line 2, A is still in the cache; the callback disables the button before line 3; but line 3 enables the button again using the state of the cache obtained in line 2.

[10]A multithreaded application has multiple simultaneous points of execution in a shared address space. Refer to Chapter 4 of [Nelson91] for an introductory discussion to concurrent programming using threads.

```
     ....
1    MonitorFiles("A", EnableOrDisableButtons);
2    AisAvailable := FilesAvailable("A");
3    EnableOrDisableButtons(AisAvailable);
     ...
```

Figure 4.4 - Fix for Race Condition described in Figure 4.3 for Single Threaded Applications

The problem in Figure 4.4 can be solved by executing lines 2 and 3 atomically, that is, disabling callbacks between lines 2 and 3. If the callback is invoked after line 3, it would leave the button in the correct state. If the callback is invoked before line 2, there is no problem because the result of `FilesAvailable()` in line 2 is up to date. Figure 4.5 shows how the multithreaded application's problem can be fixed using mutual exclusion. In general, an application should register callbacks for all the files it is interested in before querying the cache. Similar problems in the Network Monitor Interface can be solved in the same way.

```
...
VAR CallbacksMu : MUTEX;  (* lock for mutual exclusion *)
...
PROCEDURE EnableOrDisableButtons(...) =
BEGIN
  LOCK CallbacksMu DO  (* acquire MUTEX to proceed *)
     .....
  END;  (* release MUTEX *)
END EnableOrDisableButtons;
...
...
BEGIN (* Main Body of Program *)
  ...
  MonitorFiles("A", EnableOrDisableButtons);
  LOCK CallbacksMu DO
     AisAvailable := FilesAvailable("A");
     EnableOrDisableButtons(AisAvailable);
  END;

  ...

END.
```

Figure 4.5 - Disabling Callbacks using Mutual Exclusion

A careful reader might notice that our effectiveness in managing partially functional applications depends on the quality of the hints we get from the Query and Notification mechanisms. This is only a problem if events are generated faster than we can handle them, e.g. if file A is paged in and then paged out while we are still in the callback procedure, the button will incorrectly enabled until we complete next callback for the `PagedOut` event. The only time this can happen is if the cache is thrashing or if the network latency oscillates. One solution is to detect these conditions and suppress callbacks until the system stabilizes. We ignored this problem in our implementation because it does not occur frequently enough to justify the additional programming complexity.

4.5 Generalizing Property Specifications

Although we claim that Property Specifications are a generally useful abstraction for system service interface design, our discussions have focused on Property Specifications in the context of mobile computing. This section describes how Property Specifications can be applied to virtual memory (VM) management. Our discussion here is aimed at VM systems in general, not just in the context of mobile computing.

4.5.1 The Traditional Virtual Memory Interface

Like caching, virtual memory management has traditionally been transparent to applications. The basic idea is to use primary storage as a cache for secondary storage.

The VM manager is essentially a cache manager which moves chunks of data (or pages) between primary and secondary storage.

For most applications, the transparent VM interface is a feature. Application programmers are freed from the tedious task of storage management. But for some applications, the VM interface seems to hide too much. One class of applications that wants less transparency is applications which need to keep key data structures in primary storage for performance reasons. For example, both the UNIX Fast File System (FFS) [McKusick] and the Sprite Log-structured File System (LFS) [Rosenblum] cache inodes[11] in main memory to reduce disk accesses. The problem is that the VM manager can swap these data structures out to disk without informing or asking for the applications' opinion, causing poor performance or even incorrect behavior. Property Specifications solves this problem without sacrificing the transparency preferred by most applications.

4.5.2 Property Specifications for a Virtual Memory Interface

For simplicity, we define a memory object to be an arbitrary chunk of storage allocated by `malloc()` and deallocated by `free()`. The key property we choose to expose is that a memory object has two states: either it is entirely in primary storage or otherwise (part or all in secondary storage). The Property Specification, as shown in Figure 4.6, exports three procedures. `IsInPrimary()` lets an application query the location of a memory object. The application can monitor the paging activities of a memory object with `MonitorMemoryObj()`. We also allow the application to explicitly request to

[11]In the UNIX file system, every file and directory is represented by an inode. It is a data structure internal to the file system and contains the file's attributes (access rights, owner, etc.) and the physical location of the file on disk.

have a memory object "pinned" in primary storage by calling `KeepInPrimary()`. `KeepInPrimary()`'s semantics is similar to `MakeAvailable()` of the file system Property Specification: it is a one time request which the VM manager can accept or refuse. The VM manager can limit the amount of primary storage each application can pin down to guard against overly demanding applications.

```
INTERFACE VMProperty;

FROM VMFunctional IMPORT MEMORY_OBJ; (* Import definition *)

TYPE

     EnvEvent = {InToPrimary, OutOfPrimary};

     MEMORY_OBJ = REF ARRAY OF CHAR;

     CallbackProc = PROCEDURE callback(event : EnvEvent;
               mem : MEMORY_OBJ; callback_arg : REFANY);
          (* callback_arg is supplied by the application
          when registers the callback procedure *)


PROCEDURE IsInPrimary (mem : MEMORY_OBJ) : BOOLEAN;
(* Query: Returns TRUE if all of mem is in primary
     memory, FALSE otherwise *)

PROCEDURE MonitorMemoryObj (mem : MEMORY_OBJ;
          callback : CallbackProc; callback_arg : REFANY);
(* Notification: invokes callback in the
     application's address space if any part of mem is
     moved out of primary memory, or if all of mem is
     moved into primary memory *)

PROCEDURE KeepInPrimary (mem : MEMORY_OBJ) : BOOLEAN;
(* Hinting: RETURNS TRUE if the VM manager can promise
     to keep all of mem in primary storage until
     mem is freed, FALSE otherwise *)

END VMProperty.
```

Figure 4.6 - Property Specifications for the Virtual Memory Interface

With the VMProperty interface, it is trivial for LFS and FFS to monitor the paging activities of their critical data structures and to keep them in primary storage if necessary.

FFS also caches parts of files in memory, and it can now pin those pages in primary storage to prevent the VM manager from making another copy of the files on the swap disk. The VMProperty interface allows us to provide adequate support for sophisticated applications such as LFS and FFS while maintaining transparency for ordinary applications.

Chapter 5

Implementation

We built a prototype system to clarify, demonstrate and evaluate our ideas. Our rewards have been threefold. First, we gained a better understanding of the engineering and semantic issues in realizing Property Specifications. Second, we experienced first hand how an application programmer might use Property Specifications. Third, we verified the effectiveness of our approach through using wc-xmh. The feedback we got from users instigated changes and simplifications to our interface design.

This chapter describes the design and implementation of our prototype system. In Section 5.1, we provide the system overview and describe the major design decisions we faced for each component of the system. In Section 5.2, we explain our decision to modify xmh and describe how wc-xmh's features were implemented. We save the trials and tribulations we experienced during those frustrating debugging sessions for Section 5.3. In Section 5.4, we present some ideas for future research.

5.1 Implementing Property Specifications

5.1.1 System Overview

As shown in Figure 5.1, the prototype system consists of a simulator for an intermittent network (LinkSim), a user level cache manager and file system (file system with Property Interface or FPI), a Network Statistics Monitor (NSM), and a modified version of xmh (weakly connected xmh or wc-xmh, as illustrated in Chapter 2.) We implemented FPI and NSM as specified in Figures 4.1 and 4.2. Any distributed file system would have been adequate as our underlying file system, but we chose NFS [Sandberg] because it does not cache files. Since our implementation was for UNIX workstations, it is no surprise that we chose to test our ideas by redesigning the file system and network interfaces: both are key components of the UNIX distributed computing environment. Our decision to simulate a mobile environment was due to the flexibility and control we needed to better explore the design space, e.g. being able to easily change the frequency and duration of unplanned disconnections.

We chose to implement the NSM, FPI and LinkSim as separate processes because they are functionally and logically independent. In a production system, these three entities will most probably exist separately: the NSM will be part of the transport layer network interface, the FPI will be part of the file system, and LinkSim will be replaced by the true characteristics of the network's link layer. Using RPC as our primary interprocess communication mechanism forced us to focus on our original goal of designing clean abstraction boundaries. It would have been harder for us to stay focused had we implemented the NSM, FPI and LinkSim as a single UNIX process with multiple threads.

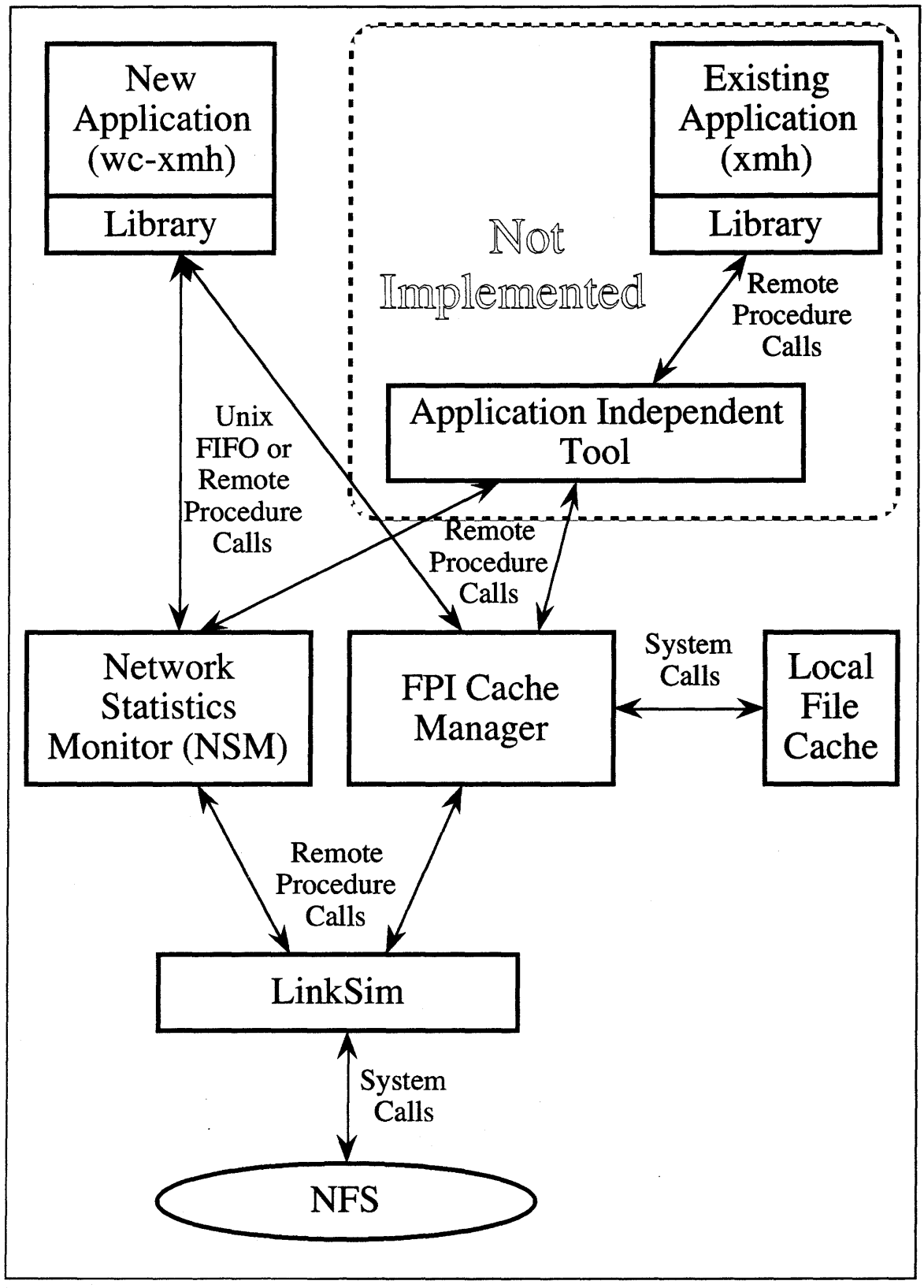

Figure 5.1 - Overview of the Prototype System

The implementation was done entirely on a UNIX workstation. FPI, NSM and LinkSim were all written in Modula-3 [Nelson90] with the exception of the application libraries for FPI and NSM, which were written in C. Wc-xmh added about 1,000 lines of C to xmh's 13,000. We chose Modula-3 as our primary implementation language because it provided lightweight threads, objects oriented programming, modules, garbage collection and type checking, which all contributed to the shortening of development time. NSM and FPI are multithreaded so they can handle multiple client applications concurrently, mimicking concurrency in the kernel. For interprocess communication, we used Sun RPC [SunRPC], Xerox PARC Modula-3 RPC [ParcRPC] and UNIX FIFO files.

5.1.2 Implementing Notification

Both the FPI and NSM have an application library which is linked into every application. The actual services are implemented in the FPI and NSM servers. The libraries provide wrappers which initialize RPC connections and cause the RPC's to the servers to look like system calls local to the application. The key function of the libraries is in managing callback procedures. This is necessary because the callback threads need to execute in the application's address space. Each library maintains a table of callback procedures and arguments, indexed by environmental events. When an application registers a callback on a particular event, the library registers with the FPI or NSM server to receive the event and inserts the given callback procedure into the table entry for that event. An event can arrive either by RPC or on a UNIX FIFO, at which point the library extracts the appropriate callback procedure from the table and invokes it. In our implementation for the Xt[12] toolkit [XtIntrinsics] [Asente], we mounted a UNIX FIFO as an input source for Xt, and caused Xt to call our table lookup procedure whenever the FIFO is ready for

[12]Xt is a toolkit for the X window system. Wc-xmh was implemented using Xt and Xaw [XtIntrinsics], the MIT Athena widget set.

reading. The lookup procedure then invokes the application's callback procedure before returning control to the Xt event loop.

5.1.3 Implementing the File System Property Specifications

As shown in Figure 5.1, FPI has two components: an application library and a cache manager. The application library provides wrappers for all the file system calls, similar to libc.a[13], but re-directs those calls to our user level file system instead of the kernel. We implemented FPI as a user level process instead of modifying the NFS code in the UNIX kernel. This enabled us to easily experiment with different file system interfaces and implementations without dealing with the complexity of kernel programming or affecting other processes running on the workstation. In order to transparently route system calls away from the kernel, we implemented wrappers for existing file system calls such as open and close as well as the new calls we added. Applications which use the FPI must be linked with the FPI library (libfpi.a). The UNIX linker resolves library calls on a "first come, first served" basis, thus in order to have libfpi.a's wrappers shadow those supplied by libc.a, libfpi.a must be linked before libc.a. For example, the `open()` call from an application linked with libfpi.a would go to our user level file system rather than the kernel `open()` call, whose wrapper is in libc.a. Although our user level file system is functionally backward compatible with the existing UNIX and NFS file system semantics, it is not binary compatible. Existing applications linked with libc.a must be relinked against libfpi.a even if they only want to use FPI's functional interface. In a production system where the FPI is implemented in the kernel, the new file system will be binary compatible.

[13]libc.a is the UNIX C library, which includes wrappers for system calls.

FPI's cache manager maintains a file cache on the workstation's local disk. The cache manager's replacement and write back policies are determined in part by the hints given by applications. For each file, the cache manager keeps the set union of all the hints given by different applications. Files marked "`UserRequest`" are given the highest priority, followed by "`AppResource`" and then "`AppData`". The priority of a file is the sum of all the hints it is associated with, so an "`AppResource`" file shared by many applications may have a higher priority than a file with a single "`UserRequest`" hint. Our replacement algorithm also ages all hints so their influence deteriorates with time and lack of use. The algorithm was kept quite simple because we were not trying to find optimal use for hints in the cache manager.

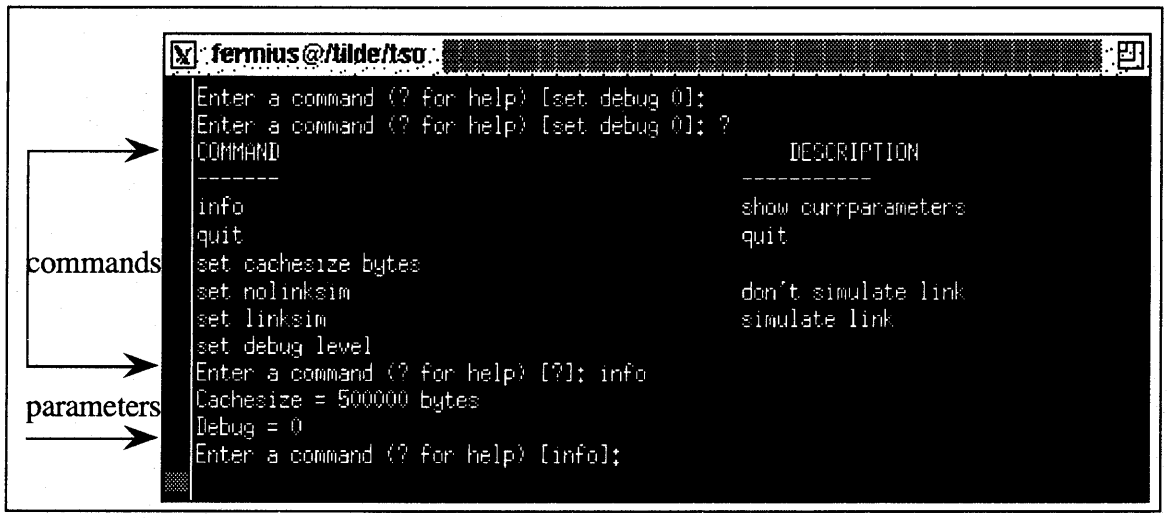

Figure 5.2 - A Command Interpreter for FPI

For our user experiments, it was useful for the cache size to be dynamically configurable. The FPI server has a command interpreter as shown in Figure 5.2. It allows the user to change the cache size dynamically, and to choose whether or not to simulate the network with LinkSim. We were able to observe the effects of a thrashing cache on wc-xmh by reducing the cache size. All network operations in FPI such as file copying were delayed

by the latency value supplied by LinkSim. Large files were delayed proportionally more than small files.

5.1.4 Implementing LinkSim

LinkSim is an event driven simulator for the packet latency of an intermittent network. It has three states: disconnected, connected and interference. The latency for both disconnected and interference modes is infinite. Disconnected mode simulates the user moving out of range of the communication medium or being forced to disconnect for congestion or cost reasons. Disconnected mode typically lasts from seconds to minutes whereas interference is temporary, lasting for a few seconds in most cases. Interference mode corresponds to the user moving near a phone, refrigerator or other sources of interference for radio and infrared networks. The simulator goes between connected and disconnected states, spending a random duration in each. A certain percentage (probinterf) of time in connected mode is spent in interference mode, and the duration of the interference is also a randomly distributed. We chose to model these durations with exponential distributions because we did not need to specify the variance. The means of the exponential distributions and probinterf, the percentage of time spent in interference mode, can be set dynamically using a command interpreter, as shown in Figure 5.3. The default means were 10 minutes for connected mode, 5 minutes for disconnected, and 2 seconds for interference. On average, 10% of the connected time is in interference mode. Increasing the interference percentage has the effect of increasing the frequency of interference.

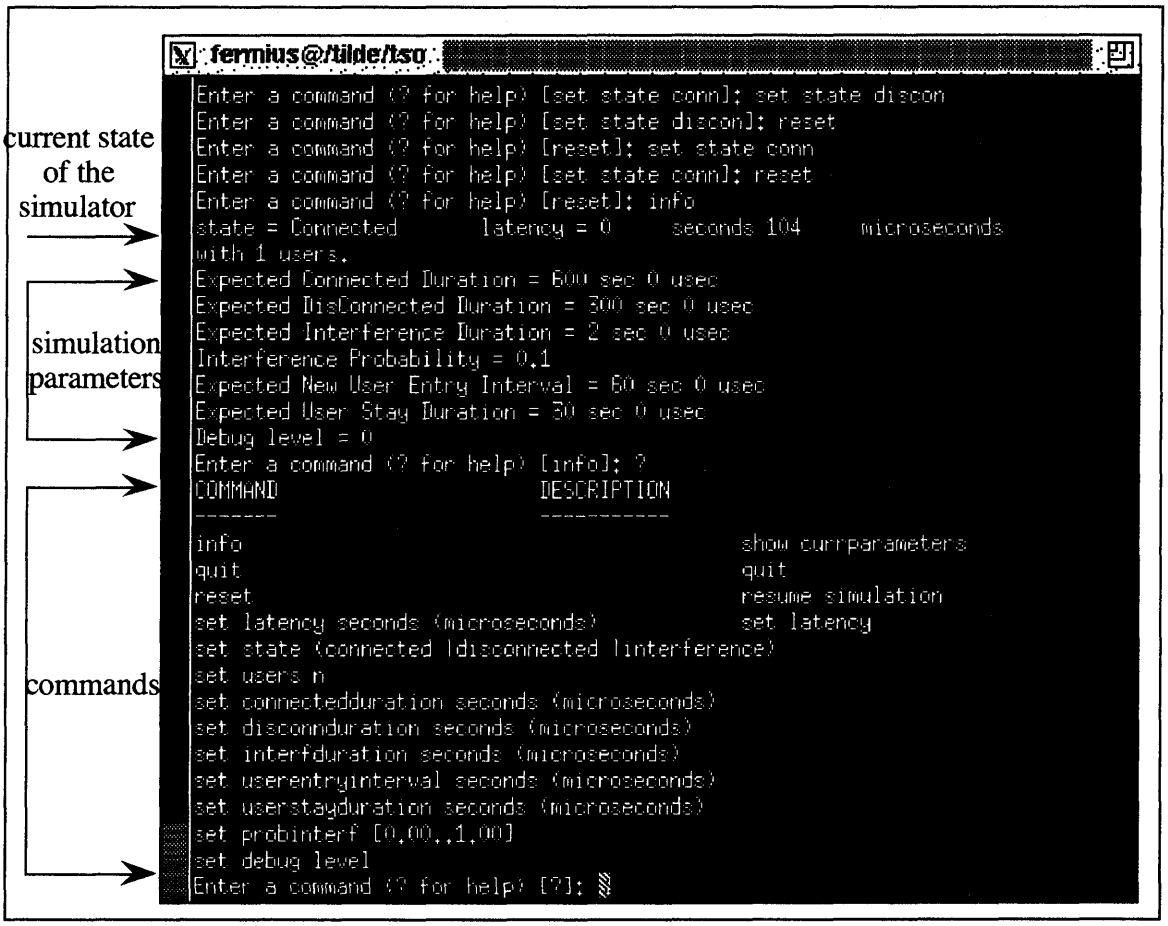

Figure 5.3 - The LinkSim Command Interpreter

While in connected mode, the latency is randomly distributed but is a function of the base latency and the number of users sharing the medium. We modeled the interval between new users arriving and the duration of each user's stay as exponentially distributed random numbers. They can also be set using the command interpreter. The user can also explicitly set the simulation state and latency. This was useful for simulating voluntary disconnections. Figure 5.4 illustrates the output of a typical run by LinkSim. LinkSim makes RPCs to FPI and NSM whenever the latency changes, this proved to be much more efficient than having FPI and NSM polling LinkSim.

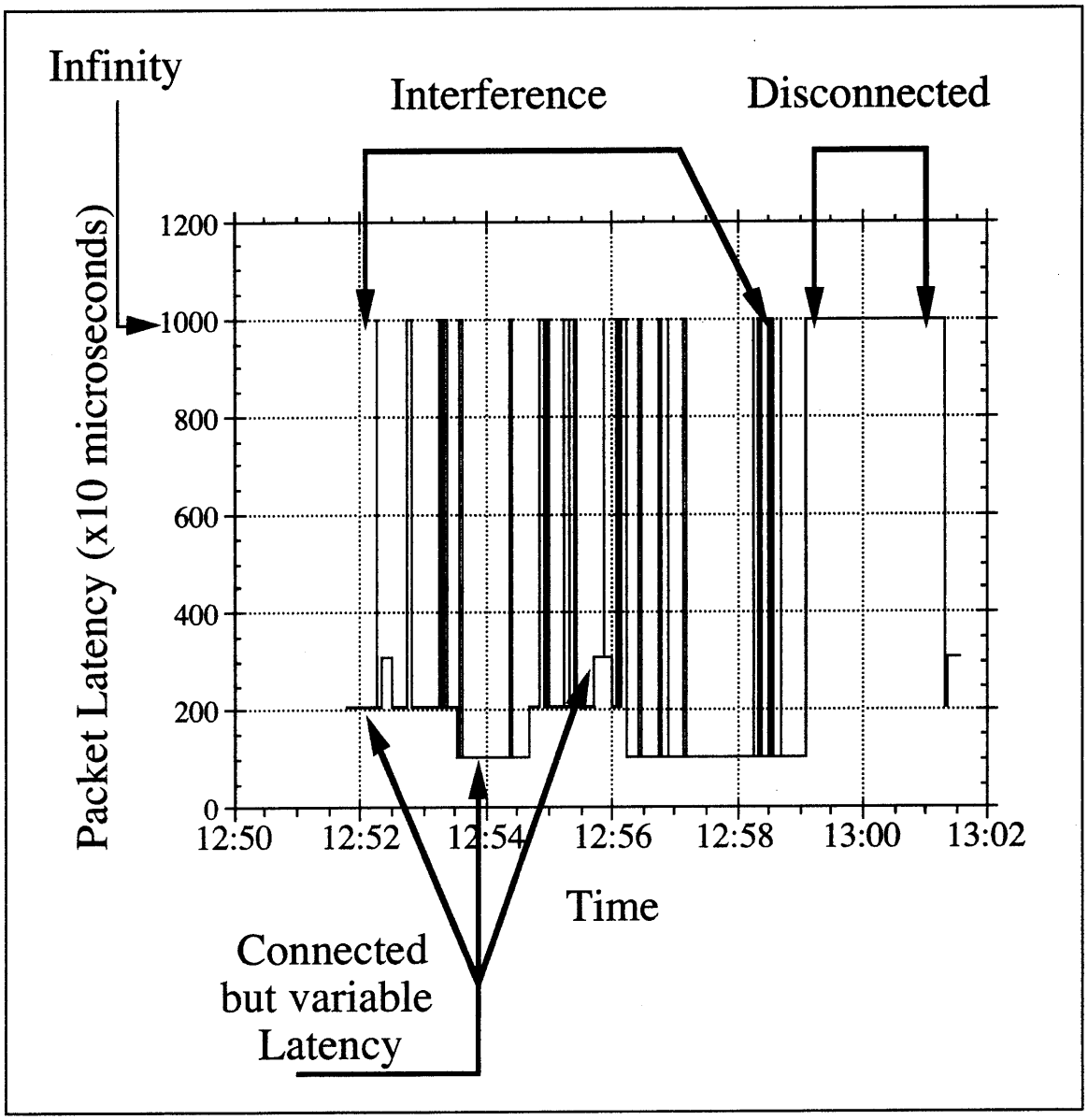

Figure 5.4 - Typical Variations in Latency as given by LinkSim

5.1.5 Implementing the Network Statistics Monitor

Our implementation for NSM is an event generator which monitors change in the network latency given by LinkSim and sends `LatencyChanged` events onto applications which have registered interest. If the underlying network interface is a real

network instead of LinkSim, our NSM will monitor statistics collected by programs similar to TCP [Postel]. For example, the TCP retransmission mechanism has an efficient and effective algorithm [Jacobson] for estimating the mean and variance of the round-trip packet latency. Therefore implementing the NSM in real networks should be quite trivial.

5.2 Using Property Specifications for Application Programming

5.2.1 Approach and Choice of Application

Our primary goal was to gain experience in implementing and using new application features that support mobile computing. We were faced with the following possibilities: implement a new application from scratch, modify an existing application, or build an application independent tool as we discussed in 4.1.3 but make no change to existing applications. We would liked to have tried all three options, but time limitations forced us to choose only one. Our decision to modify an existing application was mainly due to our desire to learn about the effectiveness of application specific features as well as the implementation overhead needed to acquire them. We did not implement the application independent tool because we believe that the scope of support it can provide is limited and does not fully exploit the power of Property Specifications. Building an application from scratch would allow us the freedom of exploring new application features but it would be difficult for us to gauge the implementation overhead caused by these features.

We chose to modify xmh because it was widely used, had a graphical user interface, used the file system extensively, and because we had access to the source code. In addition, email was of particular interest to us because it was being hailed by industrial sources as

the "killer" application for wireless mobile computing [Loudermilk]. Xmh's popularity was an advantage because our users were familiar with the user interface and basic functionalities, allowing them to focus on exploring the effectiveness of our new features. Some of the other applications we considered were: xrn[14], cm[15], and xedit[16], because we felt they would be useful applications for mobile computing. We did not choose xrn because it used NNTP[17] to access its articles instead of the file system. Therefore, unlike the file system, any solution we provided for NNTP would not be applicable to any other application. We chose xmh over cm and xedit because it was more challenging. Xmh worked with a much larger data set and had many more interesting features.

5.2.2 Wc-xmh: Weakly Connected xmh

First we partitioned all the files wc-xmh used into two categories: resource and data. Resource files are needed for xmh's features, e.g. context, .mh_profile, mtstailor, and all the .xmhcache files. Mail messages are data files. We then had to understand wc-xmh's internal dependencies, i.e. the resources needed by each of its features. In designing the user interface techniques to handle graceful disconnected operation, we tried to be consistent with original xmh conventions and being as unobtrusive as possible. We were delighted to find that our users often did not even notice the new features at work until we pointed out to them.

The majority of the implementation was fairly mechanical. An excerpt from wc-xmh's startup sequence is shown in Figure 5.5. Note mutual exclusion is not necessary because wc-xmh is single threaded. At startup, we register callback procedures to all the resource

[14]X News Reader, a NetNews browser with a graphical user interface.
[15]Calendar Manager from Sun's SparcStation DeskSet.
[16]Simple text editor/browser built with the Xt toolkit and Xaw widget set.
[17]Net News Transfer Protocol, xrn uses it to access news articles stored on a NetNews server.

files, and ask the cache to try to make them available. We then query the cache and enable or disable all the menus and buttons according to the availability of the resource files. The same callback procedure, `EnableProperButtons()`, is registered with both the cache and the NSM. It is invoked whenever a resource file is paged in or out and when the network connectivity changes. `EnableProperButtons()` encapsulates all of xmh's internal dependencies. It enables or disables buttons depending on the state of the network and the availability of wc-xmh's resource files. Callbacks from NSM updates the network latency thermometer shown in Figure 2.3 whenever the network latency changes substantially. Code excerpts from `EnableProperButtons()` is shown in Figure 5.6 and Figure 5.7 shows the callback procedure for monitoring the network latency.

```
static char *resources[] = {
     "replcomps", ".mh_profile", "context", "mtstailor",
     "forwcomps", "components", "MailAliases" /* etc. */};

struct latency latency_range;  /* the latency range which
                                  defines what it means to be
                                  connected for xmh */

void InitializeWorld()
  {
    MonitorFiles(resources, EnableProperButtons(), NULL);
    MonitorLatency(latency_range, EnableProperButtons(),
                   NULL);
    ...
    (void) MakeAvailable(resources, AppResources);
    EnableProperButtons(FilesAvailable(resources));
    ...
  }
```

Figure 5.5 - wc-xmh startup sequence

The availability indicators described in Section 2.1.1 are implemented by registering callback procedures for messages, and adding or removing the asterisk next to the message header when the corresponding file becomes available or unavailable. Since wc-xmh typically manages thousands of old messages, we only registered callback procedures for messages in folders which the user had opened.

```
Boolean CompAvailable()
{ /* Dependencies of the "compose" feature */
  return (BareEssentialsCached()
     /* resource files required by all features */
    && EssentialFilesMonitor[COMPONENTS].cached
     /* the "Components" template is needed for compose */
    && TocGetScanfileCached(DraftsFolder))
     /* .xmhcache file for drafts folder is needed */
    || NetworkGood());
     /* these files are all available if we are connected */
}

void EnableProperButtons()
{
  ...
  ...
  SendMenuEntryEnableMsg(Message_Menu, "compose",
          CompAvailable());
  ...
}
```

Figure 5.6 - EnableProperButtons() code excerpts

```
void NetLatencyCallbackProc(event, latency, client_data)
     EnvEvent event;
     struct timeval latency;
     XtPointer client_data;  /* unused */
{
  switch (event) {
    case LatencyChanged:
     NetLatency = latency;  /* NetLatency is global */
     UpdateNetworkIndicator();
     EnableProperButtons();
     if ((PendingJobs > 0) && NetworkGood(latency))
          DoPendingJobs();  /* send any pending mail */
    default: break;
  }
}
```

Figure 5.7 - Callback procedure for network latency changes

Implementing the Smart Availability Management and Dependable Future Availability features as described in Sections 2.2.3 and 2.2.4 were relatively simple using Hinting. Resource files critical to wc-xmh's features were distinguished by the "`AppResource`" hint, given by the `GiveHint()` call. Any user request to make files available through "Enable" or "BringOver" commands translated into `MakeAvailable()` calls with the "`UserRequest`" hint. These calls are answered synchronously, providing the application and the user with immediate feedback on whether the request was granted or not. Wc-xmh also registers callback procedures on any files explicitly made available due to user request, so that if these files later become unavailable, wc-xmh will either try to make them available or notify the user. This way, after the user has explicitly made some data and features available, he can continue working without worrying about unknowingly causing some of those data and features to become unavailable again. Thus the user does not have to request his set of data and features just prior to disconnection. He can request whenever he wants and unless otherwise notified, he can depend on those features and data to be available when he disconnects.

5.3 Challenging Aspects

Since wc-xmh was written in C, it was not multi-threaded. This made asynchronous callbacks difficult to implement. Fortunately, wc-xmh used Xt toolkit's event loop, and we were able to simulate an X event by writing the event record into a UNIX FIFO file, and mounting the FIFO as an event source for Xt. However, if wc-xmh is busy, the FIFO may not be read for a long time. We found that during long running wc-xmh operations such as generating the header summary for a folder, events were lost because UNIX only buffers 2 KB of data for each FIFO file. Our solution to this problem was for the NSM and FPI to each maintain a queue for all the events destined for a particular FIFO, and

monitor the FIFO with a background thread. Whenever the FIFO is empty or near empty, the background thread takes one event record off the queue and writes it to the FIFO.

There are a number of difficulties associated with implementing the user level file system on top of the kernel file system. First, the user process only gets a maximum of 256 open descriptors, which must be shared by all of its client applications. This was not a problem in our prototype system because we rarely ran more than one or two applications simultaneously as FPI clients. Second, we could not arbitrarily assign "pseudo" descriptors for the files managed by the FPI because they might conflict with descriptors the kernel gives to things other than files, e.g. sockets. We generated non-conflicting "pseudo" descriptors by forcing the kernel to assign a descriptor to "/dev/null" every time we needed a new descriptor. The third difficulty with our user level file system was preserving the application library's state, e.g. our "pseudo" descriptor table, in child processes created by the `exec()` system call. The problem exists because our descriptor table is in the application's address space which is not inherited by the child process created by `exec()`. Our solution was to write the library's state to the /tmp directory before the `exec()` call and reading and restoring the library state in the child process when the library initializes itself.

Xt, Xaw and the mh library programs consist a large amount of fairly sophisticated code. Our reluctance to change this body of code had two effects. First, we did not get to test out our ideas for "tri-state" buttons and "color-coded" buttons we introduced in Section 2.1.4. Second, some of wc-xmh's new features were hard to implement because these library programs did not provide adequate error prevention and handling. For example, if the Xaw text widget cannot open the file containing the text it needs to display, it calls Xt's "quit application" procedure. Thus if we try to read an uncached message and the network disconnects while Xaw is trying to open the file containing the message, wc-xmh

will crash. Similarly, some of the mh library programs exit when they encounter errors like "network timed out". It is difficult for wc-xmh, which forks these programs as child processes, to detect and report such errors in a meaningful way to the user.

Another problem we confronted was the need for atomicity and recoverability. This problem was amplified by the mh library programs: when they crashed, they often left the wc-xmh's file and directory structure in an inconsistent state. There are two complementary approaches to address this problem. One option is to provide application independent support, such as a file system level transaction mechanism which allows groups of operations to be executed atomically or use a programming language which provides transactions as primitive operations [Liskov87]. These solutions have very nice semantics but implementing them efficiently for production systems is a major challenge. Our other option is to always proceed optimistically, detect errors and restore the application's external state using application specific methods. We implemented error recovery for wc-xmh's "pack" feature using this technique. "Pack" uses the mh program `pack()` to consolidate the message numbers of all the messages in a folder by renaming the files containing those messages. If the network disconnects while `pack()` is running, `pack()` immediately stops and returns with an error. But it is impossible for wc-xmh to know exactly which message caused the error and how much `pack()` was able to accomplish. Thus the table of contents for the folder (the .xmhcache file) becomes out of date, e.g. the header for message 5 may no longer refer to the same message. Instead of executing "pack" as an atomic operation to ensure consistency, we chose to mark the effected folder as "out of date" and use wc-xmh's built-in feature, "rescan", to bring the table of contents up to date once connectivity is restored. "Rescan" generates new table of content files. This is an example where error recovery is greatly simplified by using application specific information and tools. We think it may be a good idea for system services to allow applications to override system level error recovery

methods with application specific ones. We believe that support for transactions will still be necessary for application operations where error recovery is difficult. Transactions may also be useful for application programmers who do not wish to deal with the overhead of writing application specific error recovery routines.

We considered letting the cache manager allow applications to "pin" down files in the cache temporarily. The idea of "pinning" is not new, virtual memory systems [Young92] [McNamee] [Cheriton] [Harty] often provide this feature. In our file system, pinning would be useful to prevent errors caused by disconnections: an application can bring all the files it needs for a particular operation into the cache by calling `MakeAvailable()`, and pin them in the cache for the duration of the operation. Of course this only prevents those errors caused by using unavailable files when disconnected. We did not implement this feature because wc-xmh did not need it. We also had no desire to complicate the semantics of FPI or to distract any attention from clearly illustrating the idea of Property Specifications.

5.4 Ideas for Future Work

5.4.1 Verifiability of Property Specifications

Formal specifications [Wing] allow us to reason about the correctness of programs. Although formal specifications are a promising area of research in programming methods, their utility across a wide range of software development projects has yet to be demonstrated [Liskov90]. But this is likely to change as we build larger and more complex systems. The use of formal methods is especially important for understanding

the behavior of mobile computing environments because they are massively distributed, highly heterogeneous, dynamically configured, and evolve over time.

There exist languages and tools [Guttag85] [Guttag90] that reason about the correctness of programs based on their Functional Specifications. While Functional Specifications describe the behavior of the program, Property Specifications describe the effects of the computing environment on the program. We believe that it should be possible to reason about Property Specifications just as we reason about Functional Specifications. Auxiliary specifications are needed to model aspects of the computing environment, such as network latency and disconnections. Although it appears that modeling the environment in real time is difficult, e.g. distinguishing sluggish networks from disconnected ones, the fact that we were able to implement Property Specifications successfully gives us confidence that we can reason about them. We propose the verification of Property Specifications and formal specifications of environmental constraints as future work.

5.4.2 The "cause/effect" Problem

One problem in a multiprogramming environment is the "cause/effect" problem: when something is paged out, how does the user know what caused it to be paged out? When disk space is limited, the user needs a way to tailor the availability of features, which requires knowledge of the effect of enabling one feature on the availability of others. This is particularly difficult when the operating system is multitasking: background jobs may be running and may wake up to run periodically (e.g. cron jobs[18]), causing files to be paged out of the cache.

[18]Background UNIX tasks which are scheduled to run periodically.

We are optimistic that this problem can be solved for two reasons. First, inside the cache manager, there is definitely sufficient information to know at least which application caused paging activities. The cache manager may allow some files to be marked "super critical", and when they must be paged out, it will lock the cache and notify the application before proceeding. The application can tell the user the consequences of this file being paged out as well as the name of the other process which is causing this file to be paged out. At this point, the user has the option to directly influence the cache manager's decision.

The second reason for our optimism is that disk capacity will continue to become less critical in the future. Portable computers today often have hundreds of megabytes of disk space[19]. Increased disk capacity means we will not need to deal with application features and data on as fine a granularity as we did with wc-xmh, where messages and features were managed individually. Future applications might only allow the user to manipulate "working sets" consisting of large chunks of features and data, and either all or none of the features and data in a group are made available. This level of granularity would make user level negotiations much simpler. Another consequence of the increase in disk capacity is that applications can afford to keep resources like icons and fonts with the executable like Macintosh applications, rather than like UNIX applications which separate application resources and binary. If we assume that the application's binary and resources are local on the portable computer, we can achieve a high level of availability without using the "`AppResource`" hint, and not have to monitor every feature with `EnableProperButtons()`.

[19]For example, I recently acquire 210 MB Macintosh Powerbook internal disk for $575.

5.4.3 Supporting Atomicity

Another interesting area for future research is in investigating data access interfaces which would provide better support for data consistency and atomicity. It might be interesting to try organizing the file system as a database and to think about operations on files as database transactions rather than as operations on a collection of bytes. This could make error prevention and recovery easier for the application. More attention should be paid to understanding the tradeoff between the semantics of the data access interface and its efficiency, and the role of application specific error recovery in allowing more optimism but using weaker semantics.

5.4.4 Remote Evaluation

Another interesting area of research is in providing support for remote evaluation. Remote evaluation would be particularly useful for implementing features like wc-xmh's "rescan". "Rescan" generates a table consisting of message headers by examining every message file belonging to the same folder. "Rescan" is expensive because it causes the cache manager to page in all the message files which will be used only once. Remote evaluation can prevent "rescan" from thrashing the cache by executing the code to generate the message headers on a backend machine. It might be challenging and expensive to implement general remote evaluation where an application can execute any arbitrary program remotely. An alternative approach is to provide a toolkit of popular remote procedures, such as searching. The toolkit procedures are implemented by a proxy process running on the backend which communicates with the toolkit on the frontend via RPC. For example, the toolkit might provide `remote-grep()` for searching. "Rescan" would call the `remote-grep()` wrapper in the toolkit, which

sends the arguments to the backend server. The server executes `grep()`, accessing files over a high speed network, and returns the result to the toolkit. Then wc-xmh would complete the "rescan" operation by formatting the headers returned by the toolkit into a table of contents file.

5.4.5 Loose RPC

We also feel that traditional RPC semantics are too strong for an intermittent environment. It might be interesting to explore a "loose" RPC mechanism which allows the application to make a call, disconnect, and then asynchronously reconnect in the future to collect the result. For example, an application may send a database query to a backend database server, disconnect, and eventually reconnect to retrieve the results of the query. On the client side, we need to put calling threads to sleep during the call and waking them when the result becomes available. On the server side, we need to collect and buffer results for future retrieval. The key difference between "loose" RPC and traditional RPC is that the call is not completed by the server returning the result as soon as it is produced, but rather by the client who eventually reconnects to get the result.

Chapter 6

Experience and Evaluation

We have learned a great deal from building and using our system, even though our experience is limited to one programmer and five users, four of whom read email with the prototype in a simulated environment for about a week each, and the other (biased) user used it for several months. In this chapter, we first highlight some of the interesting feedback from our users, and then conclude by evaluating the effectiveness of wc-xmh and Property Specifications in meeting our goal of supporting graceful disconnected operation in an intermittent environment.

6.1 User Experience

6.1.1 A Furious User

One day a furious user walked into my office. He demanded to end the experiment early because "[he does] not know what is going on with the simulation and wc-xmh is

unusable". "But you had no complaints during the first two days. What specifically do you want to know?" I asked. He then told me that he had clicked a button to open a folder and waited for a long time and he was not sure if wc-xmh was hanging, so he ended up killing the process. He found wc-xmh to be no longer usable because he felt uncomfortable waiting for any slow operation to complete. After examining the code, I realized that his problem was caused by a bug in `EnableProperButtons()` which overlooked some of the folder buttons. After I fixed the bug and explained to him that wc-xmh was designed specifically to prevent problems like what he had experienced, he continued with the experiment.

The user's fury caused by this bug immediately underscores the problem this thesis addresses: unpredictable failures are intolerable! Although LinkSim produced unplanned disconnections quite frequently, none of our other users found using wc-xmh to be very different from using xmh. This is an encouraging sign that wc-xmh was effective in allowing users get work done despite the intermittent network.

6.1.2 Obtrusive User Interface Techniques

The first user of the wc-xmh complained that after he had left wc-xmh running overnight, upon his return, he found his screen covered by about a dozen pop up notices telling him that various features were no longer available. We learned two lessons from his experience. First, it is very important for an adaptive user interface to behave unobtrusively. The user should be warned with pop up notices only when absolutely necessary. Second, pop up notices about features being unavailable are a lot more useful if the user can find out what was the *cause*. This problem raises some subtle issues about the role of cause and effect as discussed in Section 5.4.2. When some features of

application A are paged out, the user needs to know whether they were paged out by some unimportant background job or by application B when he asked it to make some features available. If he knew the cause, he could kill the background job or ask B to make fewer of its features available so he could retain A's features.

6.1.3 Voluntary Disconnection

One user simulated voluntary disconnection by manually switching LinkSim between connected and disconnected modes, pretending that he was moving in and out of active areas for his portable computer's radio. The surprise came when we examined his activities log. He was connected only a few times each day, and each time for only a few minutes. Apparently the confidence he has gained in wc-xmh's ability to operate disconnected allowed him to dramatically lower his connection time. It appears that if the application does not handle disconnections gracefully, the user would remain connected for much longer than necessary *just in case* he might do something which will hang or crash the application.

We feel that the applications' connectivity requirements can be reduced even further by generalizing Notification. For an application which is voluntarily disconnected because of network cost or congestion concerns, it is very useful to have backend services which will notify the application when something of interest happens. For example, wc-xmh would like to be notified if new mail arrived in the user's mail box. Another example: when I am away from my office and another user tries to schedule a meeting with the calendar manager on my workstation, it should try to confirm the appointment by paging the palmtop computer I carry with me. Notification is not only useful for managing

applications in an intermittent environment, but more importantly, it can reduce the applications' connectivity requirements.

6.2 Five Conclusions From Our Experiences

First, intermittent connectivity is a good model for mobile computing environments. For most mobile computing applications such as mail or news browsers, editors, calendar managers, and database browsers, strong connectivity is not a strong requirement because we can exploit the locality in their data sets. We believe the predominant mode of operation will be autonomous applications which occasionally connect to backend storage or retrieval systems, burst or trickle some data, and then disconnect. Intermittent connectivity is also cost effective if users are charged for network services, e.g. cellular modem users are charged based on connection time. Finally, the reliability of wireless networking will always be constrained by cost and the physical environment. An intermittent model for connectivity is practical because ordinary radios will always be cheaper and smaller than radios optimized to work near refrigerators.

Second, Adaptive user interfaces are an effective way for providing fine grained graceful disconnected operation. By fine grained we mean that we can manage the availability of a partially functional application at the level of individual features or data objects. Adaptability is a powerful way of influencing a user's expectations in the capability of his application and computing environment. The user is hidden from the intricacies and dependencies in the system and interacts with high level entities like application features and data objects, e.g. appointments in calendar. Applications which are informed about their environment and can adapt to changes in it will thrive in mobile computing environments which are highly heterogeneous and dynamic.

Third, autonomy and predictable performance are key requirements for mobile computing. Autonomy and predictable performance are the prevailing reasons why we are willing to tolerate noisy and bulky workstations or personal computers on our desks. Graceful disconnected operation provides both autonomy and predictable performance. Caching enables autonomy, and user friendly management of partially functional applications provides predictable performance. Predictability means there is a close correlation between user expectation and reality. Our users were highly irritated when their expectations were not met, e.g. when pressing an enabled button caused the application to hang. Predictable performance was enough of an incentive for our users to become a little more knowledgeable about the environment and to cooperate with the application in managing availability. Adaptive applications are important because they greatly reduce the amount of knowledge the user needs to have about the computing environment in order to work effectively.

Fourth, Property Specifications reduce the programming overhead for application features supporting graceful disconnected operation. As illustrated in Section 5.2.2, well designed Property Interfaces made most of our modifications to xmh fairly trivial. We only began to fully appreciate our ideas when we had to implement some new features on top of these libraries which do not export Property Specifications, such as Xt and Xaw. For example, Xaw causes the application to exit when it encounters network timeout errors and we had no clean way to catch those errors and continue without leaving the toolkit and application in a slightly confused state. We experienced first hand the effect of programming with the wrong abstractions in producing ad hoc code and frustrated programmers.

Fifth, application specific information can greatly simplify error recovery. Even though this thesis does not address cache consistency as a research topic, we had to provide practical consistency mechanisms to entice our users to trust their mail to our system. A errors occurs in our system when a cached file becomes stale because the original was modified by a third party, e.g. the .xmhcache file changed because the user incorporated new mail from another instance of xmh while wc-xmh was disconnected. The system detects the inconsistency, but can do little to rectify it because .xmhcache looks like any other sequence of bytes in the file system. Traditionally, the system alerts the user who must then manually sort out the problem. Based on the observation that our attempt to hide such errors from the application burdened the user, we allowed wc-xmh to override the system's default error handler on .xmhcache with its method. The error handler supplied by wc-xmh simply called an mh library routine to regenerate a most up to date .xmhcache. Resolving the inconsistency was a trivial task for wc-xmh because it understands the exact semantics of the .xmhcache file.

6.3 Evaluating Property Specifications

The strengths of our approach lies in the effectiveness in supporting both planned and unplanned disconnected operation. We demonstrated the usefulness of adaptive applications and how user expectations can be changed while still keeping the user hidden from the intricacies of the underlying system. Property Specification is a powerful abstraction for providing system level support for achieving autonomy and predictable performance. Query and Notification allow centralized and efficient monitoring of environmental events, and ensures that applications are notified of environmental changes in a timely manner. Hinting provides a general way for applications to influence and customize system level entities without having to deal with

unnecessary details of the implementation. Although applications must be modified or rewritten in order to take advantage of Property Specifications, we take great comfort in knowing that the new interfaces are backward compatible and existing applications can be ported incrementally.

Although Property Specifications greatly simplify the implementation of some application features, one disadvantage is that it requires extra understanding from the application programmer and the operating system programmer. The application programmer must understand the internal dependencies of the application, and the operating system designer must decide which properties and tools to include in the Property Specification. We believe that there is a lot more to learn about building systems and applications for mobile computing. Wc-xmh is evidence that there is fruitful research in this area.

Critics may argue against Property Specifications because we give up transparency at both the system and the user level. We believe that complete transparency at the user level is impossible in an intermittent environment. The fact is that if the network fails frequently, then the only choice we have is whether to deal with it in the system, in the application, or leave it to the user. It is not surprising that traditional system interfaces are inadequate for supporting mobile computing because they were designed for stationary workstations connected with network cables which fail very occasionally; and when they do fail, we reboot our machines and go for coffee. Mobile computing radically changes our assumptions about the computing environment and require new abstractions and tools to be developed. Property Specifications is a step in this direction. By specifying the properties of the environment and the functions provided by the implementation in separate interfaces, we give power and flexibility to sophisticated applications while maintaining transparency for ordinary applications.

Bibliography

[Adams]
N. Adams, R. Gold, B. Schilit, M. Tso, and R. Want. The ParcTab Mobile Computing System. Submitted to HICSS-27. 1993.

[Asente]
P. J. Asente and R. R Swick with J. McCormack. X Window System Toolkit, The Complete Programmer's Guid and Specification. Digital Press. 1990.

[Birrell]
A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. In ACM Transactions on Computer Systems, 2(1). February, 1984.

[Cheriton]
D. R. Cheriton. The V Distributed System. Communications of the ACM, 31(3). March, 1988.

[Cornelius92]
D. Cornelius, XRemote™: A Serial Line Protocol for X. Sixth Annual X Technical Conference, Boston MA. 1992.

[Douglis]
F. Douglis and J. K. Ousterhout. Transparent Process Migration Design Alternatives and the Sprite Implementation. Software - Practice & Experience 21(8). August, 1991.

[Dylan]
Apple Computer, Eastern Research and Technology. Dylan: An object oriented dynamic language. Apple Computer, November 1992.

[Falcone]
J. R. Falcone. A Programmable Interface Language for Heterogeneous Distributed Systems. In ACM Transactions on Computer Systems, 5(4). November, 1987.

[Fulton]
J. Fulton and C. K. Kantarjiev. An Update on Low Bandwidth X (LBX), A Standard for X and Serial Lines. Technical Report P93-00001, Xerox Palo Alto Research Center. February, 1993.

[Gosling]
J. Gosling, D. S. H. Rosenthal, and M. J. Arden. The NeWS Book. Springer-Verlag, 1989.

[Gray]
C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In Proceedings of the Twelfth ACM Symposium on Operating Systems Principles. Litchfield park, Arizona. December, 1989.

[Guttag85]
J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of Specification Languages. IEEE Software, 2(5). 1985.

[Guttag90]
J. V. Guttag, J. J. Horning, and A. Modet. Report on the Larch Shared Language, version 2.3. Research Report 58, DEC Systems Research Center. 1990.

[Harty]
K. Harty and D. Cheriton. Application Controlled Physical Memory Using External Page-Cache Management. Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V) Proceedings. October, 1992.

[Heller91]
D. Heller. XView Programming Manual. O'Reilly & Associates, Inc. September, 1991.

[Heller92]
D. Heller. Motif Programming Manual For OSF/Motif Version 1.1. O'Reilly & Associates, Inc. July, 1992.

[Herbert]
K. P. Herbert. XRemote and Terminal Services. In Proceedings of the Silicon Valley Networking Conference. April, 1991.

[Jacobson]
Van Jacobson. Congestion Avoidance and Control. In ACM SIGCOMM Symposium on Communications Architectures & Protocols. August, 1988.

[Jamsa]
K. Jamsa. DOS - The Complete Reference, Fourth Edition. Osborne McGraw-Hill. 1993.

[Jul]
E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine Grained Mobility in the Emerald System. ACM Transactions on Computer Systems 6(1). February, 1987.

[Kazar]
M. L. Kazar. Synchronization and Caching issues in the Andrew file system. Technical Report CMU-ITC-058, Information Technology Center, Carnegie Mellon University. June, 1987.

[Kiczales]
G. Kiczales, J. des Rivieres, and D. G. Bobrow. The Art of the Metaobject Protocol. The MIT Press. 1991.

[Kistler]
J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In Proceedings of the 13th ACM Symposium on Operating Systems. October, 1992.

[Liskov87]
B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In Proceedings of the 11th ACM Symposium on Operating Systems Principles. November, 1987.

[Liskov90]
B. Liskov and J. V. Guttag. Abstraction and Specification in Program Development. The MIT Press. 1990.

[Loudermilk]
S. Loudermilk and S. Higgins. E-mail: the 'killer' wireless application. Supplement on Mobile Computing, PC Week, April 1993.

[MacOS]
Apple Computer. Inside Macintosh: Overview. Addison-Wesley. December, 1992.

[McKusick]
M. McKusick, W. Joy, S. Leiffler, R. Fabry. A Fast File System for UNIX. ACM Transactions on Computer Systems, 2(3). August, 1984.

[McNamee]
D. McNamee and K. Armstrong. Extending the Mach External Pager Interface to Allow User-Level Page Replacement Policies. Technical Report 90-09-05, University of Washington. September, 1990.

[Nelson88]
M. M. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. In ACM Transaction on Computer Systems. February, 1988.

[Nelson91]
G. Nelson, Editor. Systems Programming with Modula-3. Prentice Hall, 1991.

[Nye90a]
A. Nye, Editor. Xlib Reference Manual for Version 11 of the X Window System. O'Reilly & Associates, Inc. October, 1990.

[Nye90b]
A. Nye, Editor. X Protocol Reference Manual for Version 11 of the X Window System. O'Reilly & Associates, Inc. May, 1990.

[Novobilski]
A. Novobilski. Penpoint Programming. Addison-Wesley. August, 1992.

[Ousterhout]
J. K. Ousterhout. TCL: An Embeddable Command Language. In Winter Conference Proceedings, USENIX Association. 1990.

[ParcRPC]
Xerox PARC Modula-3 RPC. Available via anonymous ftp from parcftp.parc.xerox.com or gatekeeper.dec.com. 1992.

[Peek]
J. D. Peek. MH & xmh: e-mail for users and programmers. O'Reilly & Associates, Inc. January, 1991.

[Petzold]
C. Petzold. Programming Windows™ 3.1, Third Edition. Microsoft Press. 1992.

[Postel]
J. Postel, Editor. Transmission Control Protocol Specification. ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA. September, 1981.

[Rodriguez]
L. H. Rodriguez Jr. Coarse-Grained Parallelism Using Metaobject Protocols. Technical Report P91-00130, Xerox Palo Alto Research Center. September, 1991.

[Rosenblum]
M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. Operating Systems Review, 25(5). October, 1991.

[Sandberg]
R. Sandberg, D. Goldberg, S. Cleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In Summer Conference Proceedings, USENIX Association. 1985.

[Satya85]
M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The ITC distributed file system: principles and design. In Proceedings of the 100th ACM Symposium on Operating System Principles. Orcas Island, 1985.

[Satya90]
M. Satyanarayanan, J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siefel, and D. C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. In IEEE Transactions on Computers, 39(4). April, 1990.

[Scheifler]
R. W. Scheifler and J. Gettys. X Window System, Second Edition. Digital Press, 1990.

[Schilit]
B. Schilit. Dynamic Software Customization Supporting Mobile Computing. Thesis Proposal, Columbia University. February, 1992.

[Stamos]
J. W. Stamos and D. K. Gifford. Implementing Remote Evaluation. IEEE Transactions on Software Engineering, 16(7). July, 1990.

[SunNeFS]
Sun Microsystems. The Network Extensible File System Protocol Specification. Unpublished draft, available by email: nfs3@sun.com. February, 1990.

[SunNeWS]
Sun Microsystems. NeWS 3.0 Programmer's Guide, Revision A. Sun Microsystems document number 800-6736-11. December, 1991.

[SunRPC]
Sun Microsystems. Network Programming, Revision A. Manual document number 800-3750-10. March, 1990.

[Tait]
C. D. Tait and D. Duchamp. Service Interface and Replica Management Algorithms for Mobile File System Clients. In IEEE Conference on Parallel and Distributed Information Systems. December, 1991.

[XtIntrinsics]
O'Reilly and Associates, Inc. X Toolkit Intrinsics Reference Manual. O'Reilly and Associates, Inc. January, 1990.

[Welch]
B. B. Welch. Naming, State Management, and User-Level Extensions in the Sprite Distributed File System. Ph.D. thesis and Technical Report UCB/CSD 90/567, Department of Computer Science, University of California at Berkeley. April, 1990.

[Wing]
J. M. Wing. A Specifier's Introduction to Formal Methods. IEEE Computer. September, 1990.

[Ward]
T. A Ward and S. M. Liffick with Editors B. Holmes and D. Paul. Microsoft Windows™ for Pen Computing Programmer's Reference. Microsoft Press. 1992.

[Young87]
Michael W. Young et al. The Duality of Memory and Communication in the implementation of a Multiprocessor Operating System. In Proceedings of 11th ACM Symposium on Operating System Principles, Austin, Texas. November, 1987.

[Young89]
Michael W. Young. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Ph.D. thesis and Technical Report CMU-CS-89-202, Department of Computer Science, Carnegie Mellon University. November, 1989.

Using Property Specifications to Acheive Graceful Disconnected Operation in an Intermittent Mobile Computing Environment

Michael Tso

CSL-93-8