Mobile Computing with the Rover Toolkit

by

Anthony Douglas Joseph

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

November 1997

© Massachusetts Institute of Technology 1997. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
November 26, 1997

Certified by . . . .
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Mobile Computing with the Rover Toolkit

by

Anthony Douglas Joseph

Submitted to the Department of Electrical Engineering and Computer Science
on November 26, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Rover is one of the first architectures that supports the construction of both *mobile-adaptive* applications and *mobile-adaptive* proxies for *mobile-transparent* applications. To excel in the harsh conditions of a mobile environment, mobile-adaptive applications are aware of and take an active part in adapting to those conditions. The mobile-transparent approach is appealing because it allows applications to be run without alteration. The contributions of this thesis include the Rover architecture and a reference implementation, the Rover Toolkit. Together, the architecture and toolkit support a set of programming and communication abstractions that enable and simplify the construction of both mobile-adaptive proxies for mobile-transparent applications and new mobile-adaptive applications. The programming abstractions include *Relocatable Dynamic Objects*, a form of code- and data-shipping, along with remote invocation and method logging, that allows mobile-adaptive applications to dynamically adapt to environmental changes, and programming language extensions for reliable execution of applications at servers. The communication abstractions are based upon *Queued Remote Procedure Call*, a form of remote invocation that provides asynchronous, reliable delivery of requests and results. Using the Rover abstractions, applications obtain increased availability, concurrency, resource allocation efficiency, code reuse, fault tolerance, data consistency, application adaptation to environmental changes, and more efficient scheduling and utilization of communication channels. Experimental evaluation of a suite of mobile applications built with the toolkit demonstrates that such application-level control can be obtained with relatively little programming overhead and allows correct operation, increases interactive performance, and reduces network utilization under intermittently connected conditions.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Mobile Computing with the Rover Toolkit

by

Anthony Douglas Joseph

Submitted to the Department of Electrical Engineering and Computer Science
on November 26, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Rover is one of the first architectures that supports the construction of both *mobile-adaptive* applications and *mobile-adaptive* proxies for *mobile-transparent* applications. To excel in the harsh conditions of a mobile environment, mobile-adaptive applications are aware of and take an active part in adapting to those conditions. The mobile-transparent approach is appealing because it allows applications to be run without alteration. The contributions of this thesis include the Rover architecture and a reference implementation, the Rover Toolkit. Together, the architecture and toolkit support a set of programming and communication abstractions that enable and simplify the construction of both mobile-adaptive proxies for mobile-transparent applications and new mobile-adaptive applications. The programming abstractions include *Relocatable Dynamic Objects*, a form of code- and data-shipping, along with remote invocation and method logging, that allows mobile-adaptive applications to dynamically adapt to environmental changes, and programming language extensions for reliable execution of applications at servers. The communication abstractions are based upon *Queued Remote Procedure Call*, a form of remote invocation that provides asynchronous, reliable delivery of requests and results. Using the Rover abstractions, applications obtain increased availability, concurrency, resource allocation efficiency, code reuse, fault tolerance, data consistency, application adaptation to environmental changes, and more efficient scheduling and utilization of communication channels. Experimental evaluation of a suite of mobile applications built with the toolkit demonstrates that such application-level control can be obtained with relatively little programming overhead and allows correct operation, increases interactive performance, and reduces network utilization under intermittently connected conditions.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Acknowledgments

This thesis is the result of a long arduous trip through graduate school. Along the way, many people have made the trip more enjoyable and pleasant. I owe a special debt of gratitude to Kim Nelson for helping me get through graduate school and for putting up with all the nights and weekends I spent in the lab.

Special thanks to Frans Kaashoek for providing me with guidance, support, and encouragement. Frans is a great advisor and a fun, exciting person to work with.

John Guttag also deserves thanks for mentoring me over the years. I thank him for the guidance he has given me, especially when I was changing research groups and during my search for a job.

I thank David Gifford for his efforts during the design stages of the Rover toolkit, in particular his ideas for the semantics of QRPC.

I thank my parents for the motivation they instilled in me, for all the support they have provided, and for their faith in me.

I owe a lot of thanks to the rest of the Rover project team: Joshua Tauber, George Candea, Constantine Cristakos, Alan F. deLespinasse, and Michael Shurpik. Special thanks go to Josh for all the great design arguments/discussions we shared that helped shape the Rover architecture and for his help in building and debugging applications.

I thank the Parallel and Distributed Operating Systems group for reading many drafts of the papers behind this thesis (thanks Massimiliano Poletto, Eddie Kohler, and Greg Ganger) and for providing me with a fun, exciting, and sometimes loud (thanks David Mazieres and Emmett Witchel), research environment. Thanks also to Neena Lyall for making the administrative side of the project much more pleasant.

Thanks go to my roommates (Dave Gardner, Sanjay Ghemawat, and Parry Husbands) for tolerating my equipment (and other) messes and being great roommates.

Finally, I thank my sponsors for providing the resources used by the Rover project. This work was supported in part by the Defense Advanced Research Projects Agency under contract DABT63-95-C-005, by an NSF National Young Investigator Award, by an Intel Graduate Fellowship, and by grants from Intel, IBM and AT&T.

Contents

1 Introduction **13**

1.1 Mobile versus Stationary Environment 15

1.2 Goals 18

1.3 Rover: A Toolkit-based Approach 19

1.4 Building and Using Mobile Applications 21

1.5 Arguments for Mobile-Adaptive Computing 23

1.6 Contributions and Results 24

1.7 Outline of the thesis 26

2 Related Work **28**

2.1 Relocatable Dynamic Objects 28

2.2 Queued Remote Procedure Call 30

2.3 Mobile-Transparent File Systems 31

2.4 Mobile-Adaptive Applications 32

2.5 Reliable Execution 34

3 Design of the Rover Toolkit **36**

3.1 Design Overview 36

3.1.1 Why Use a Toolkit? 37

3.1.2 Failures 38

3.1.3 Mobile Environment Problems Addressed by Rover 39

3.1.4 The Synergy Between the Solutions 43

3.2 Relocatable Dynamic Objects 43

3.2.1 RDO . . . 43
3.2.2 Using RDOs . . . 45
3.2.3 The Reliable Execution of RDOs at Servers . . . 51
3.2.4 Using RDOs for Computation Relocation . . . 53
3.2.5 The Safe Execution of RDOs . . . 53
3.3 Queued Remote Procedure Call . . . 55
3.3.1 QRPC . . . 55
3.3.2 Using QRPC . . . 55
3.3.3 QRPC Failure Recovery . . . 58
3.3.4 Client Communication Scheduling . . . 61
3.4 RDO Replication and Consistency . . . 65
3.4.1 Replication . . . 65
3.4.2 Consistency . . . 67
3.5 Mobile-Adaptive Application Support . . . 71
3.6 Design Summary . . . 72

4 Implementation of the Rover Toolkit 74
4.1 Implementation Overview . . . 74
4.2 Client-side Implementation . . . 76
4.2.1 Access Manager . . . 76
4.2.2 Persistent RDO Cache . . . 77
4.2.3 Stable QRPC Log . . . 78
4.2.4 Network Scheduler . . . 79
4.3 Server-side Implementation . . . 81
4.3.1 Access Manager . . . 82
4.3.2 Network Scheduler . . . 83
4.3.3 Stable QRPC Log . . . 84
4.4 Implementation Summary . . . 88

5 Applications Using Rover 89
5.1 Using Objects Instead of Files . . . 89

5.2 Toolkit Programming Interface 92
5.3 Rover Application Suite 93
5.3.1 Proxies for Mobile-Transparent Applications 95
5.3.2 Mobile-Adaptive Applications 99
5.3.3 Developing Reliable Applications 100
5.3.4 Stock Market Tracker 101
5.3.5 Rover Web Browser proxy 102
5.3.6 Text file search 102
5.4 Discussion 104

6 Experiments 106

6.1 Experimental Environment and Methodology 107
6.2 Null QRPC Performance 108
6.2.1 QRPC Client Costs 111
6.2.2 QRPC Compression Costs 111
6.2.3 QRPC Transport Cost 111
6.2.4 QRPC Stable Logging Cost 113
6.2.5 QRPC Server Costs 114
6.3 Discussion of QRPC Costs 115
6.4 QRPC Batching and Compression 116
6.5 Mobile-Transparent Application Performance 121
6.5.1 Rover File System Proxy 124
6.6 Mobile-Adaptive Application Performance 125
6.7 Fault-Tolerant Applications 127
6.7.1 Stable Variables 128
6.7.2 Rover Web Browser proxy 128
6.7.3 Text file search 129
6.8 Discussion 133

7 Future Work and Conclusion 134

7.1 Future Work 134

7.2 Conclusion 135

List of Figures

1-1 Rover offers applications a client/server distributed object system with client caching and optimistic concurrency control. 18

1-2 Inconsistent replicas in a replicated file system. 22

3-1 The steps in client-side import and export operations for an RDO. . . 46

3-2 The steps in server-side import and export operations for an RDO. . 47

3-3 The components of Queued Remote Procedure Call. 55

4-1 The Rover toolkit reference implementation. 75

4-2 Volatile and stable Tcl counters. 85

5-1 Server-side code for a reliable file search application (part one). . . . 103

5-2 Server-side code for a reliable file search application (part two). . . . 104

6-1 Average time to perform a QRPC. When batching is used, the times t_{connect}, t_{cCompOvr}, and $t_{\mathrm{sDecompOvr}}$ are amortized over the number of requests in the batch. 112

6-2 Time in seconds to execute 50 null QRPCs with asynchronous log record flushing. 117

6-3 Time to complete the first request in a batch of QRPCs. 119

6-4 Time to complete subsequent requests in a batch of QRPCs. 119

6-5 Time in seconds to fetch/display 10 WWW pages using Netscape alone and with the Rover HTTP proxy. 123

6-6 Speedup (or slowdown) of the Rover File System (RFS) Proxy over the Network File System (NFS). 123

6-7 Speedup (or slowdown) of Rover mobile-adaptive versions of applications over the original X11-based applications when performing common tasks. 125
6-8 Time to complete the server-side portion of the text search application. 130

List of Tables

1.1 Mobile versus stationary environment. 14

2.1 Benefits and limitations of the Odyssey project. 33

3.1 Server responses to import/export operations and client toolkit actions. 48

3.2 Cache tags options for RDO revalidation. 66

5.1 Implementation choices for the initial application set built using the Rover toolkit. 90

5.2 Lines of code changed or added in porting or implementing *Rover Exmh*, *Rover Irolo*, *Rover Stock Watcher*, and *Rover Webcal* and implementing *Rover File System Proxy*, *Rover HTTP Proxy*, and *Rover NNTP Proxy*. 94

6.1 The Rover experimental environment. Measurements include 90% confidence intervals. 109

6.2 Costs for using Rover compression and decompression algorithms. Measurements include 90% confidence intervals. 110

6.3 Attributes of the media used for stable logging (from manufacturer's specification sheets). 113

6.4 Comparison between experimental and model results. Results are in milliseconds and include 90% confidence intervals. 115

6.5 Comparison between experimental and model batch results. Results are in milliseconds. 121

6.6 Approximate times in milliseconds to execute each iteration of the example counter code in Figure 4-2 at the server. 127
6.7 Time in milliseconds (with 90% confidence intervals) to execute the server portion of the Rover Web Browser proxy while fetching the Rover project home page and its two inlined images. 129
6.8 Times in seconds to execute the server-side portion of the file search application. For the single failure case, the fault was injected after 75% of the files were searched. 130
6.9 Actual and model times in seconds to execute the server-side portion of the file search application. 131
6.10 The effects of doubling and quadrupling the amount of data logged by the text search application. 132

Chapter 1

Introduction

The mobile computing environment presents application developers with a unique set of communication and data integrity constraints that are absent in traditional distributed computing settings. These constraints make it difficult to use existing applications and complicate the development of new mobile-adaptive applications. A *mobile-adaptive* application adapts its behavior as it discovers the performance and reliability characteristics of its mobile environment. Adaptive applications to offer users better interactive performance and better utilization of network bandwidth. For example, although mobile communication infrastructures are becoming more common, network bandwidth in mobile environments is often severely limited, unavailable, or expensive. The two traditional solutions are either to create single-use, special-purpose software support for an application or to provide special-purpose support in the operating system. Unfortunately, the former approach does not lend itself to code reuse or system resource sharing by multiple applications, while the latter offers limited portability to other operating systems and limited support for conflicting changes (update-update conflicts) by remote users.

The hypothesis in this thesis is that running mobile applications efficiently and correctly often requires making applications and users aware of the environment in which they are running. This thesis posits that support for mobile applications should be provided as system facilities in a layer below applications, but above the operating system. Such facilities should: allow mobile application developers to minimize appli-

Issue	Stationary	Mobile
Network resources	High bandwidth / Low latency	Limited bandwidth and latency latency / Intermittent connectivity
Computational resources	Significant resource	Limited resource
Power	Unlimited availability	Limited availability
Storage	Unlimited storage	Limited storage
Reliability	Very reliable	Fragile, limited reliability

Table 1.1: Mobile versus stationary environment.

cations' dependence upon continuous network connectivity; provide tools to optimize the utilization of available network bandwidth; reduce battery consumption; minimize dependence on data stored on remote servers; and allow for dynamic division of work between clients and servers. The ideas behind such system facilities are discussed in this thesis and encapsulated in a proof of concept implementation, called the *Rover toolkit*. The toolkit is one of the first software tools that both supports applications that operate obliviously to the underlying environment, and enables the construction of applications that use awareness of the mobile environment to isolate themselves from its limitations. Furthermore, the toolkit allows application developers to make the transition from existing applications to mobile-adaptive applications in a gradual and incremental fashion.

This thesis presents a detailed discussion of the issues associated with the mobile environment and of the choices that were made in designing and implementing the toolkit. To illustrate the effectiveness of the toolkit, several distributed applications were implemented. Experimental results show that these applications perform well while executing over several wired and wireless networks that differ by nearly three orders of magnitude in bandwidth and latency.

1.1 Mobile versus Stationary Environment

This section discusses the differences in the resources that are available in stationary and mobile environments, the variability of those resources, and the significant problems that these issues cause for mobile computers. The important issues are listed in Table 1.1.

The *stationary environment* usually consists of some number of computers that are interconnected via a high-bandwidth, low-latency network. The computers are relatively unconstrained with respect to all of the following resources: network, computation, or power. Developers creating applications for stationary applications do not usually consider power as a resource. However, regardless of available network or computation resources, a lack of power means neither resource can be used.

The *mobile environment* usually consists of some number of portable computers that may be disconnected or interconnected with stationary environment computers via a variety of wired and wireless networks. In comparison to the computers in the stationary environment, these computers are relatively constrained with respect to one or more of the following resources: network, computation, or power.

Developers of applications for mobile environments must address several differences between the mobile environment and the stationary environment. Relative to most stationary computers, a single mobile computer has fewer computational resources available and limited electrical power; this difference makes computation migration from clients to servers an important consideration for mobile environments. However, taken together, the total sum of the computational power of the mobile computer clients may be greater than that of a stationary server, making computation migration from servers to clients to offload work an important consideration. Stationary computers may have large amounts of storage; thus, they can store very large amounts of information. Mobile computers, however, have limited amounts of storage and may not be able to store complete replicas of the data stored at servers. However, the available resources may change dynamically (*e.g.,* a "docked" mobile computer that has access to a unlimited electrical power, larger display, graphic or

math coprocessor, additional stationary storage, and high-bandwidth, low-latency networks).

In addition to the limitations that the mobile environment places upon resources, the environment also adds *volatility*. The mobile environment is significantly more volatile than the stationary environment. Network connectivity, bandwidth, latency, and cost all can vary while an application is running. In addition, computational and power resources can vary. These are changes that rarely occur in a stationary environment; as such, they are not addressed by applications in stationary environments.

Computers in a stationary environment are usually very reliable. Relative to their stationary counterparts, mobile computers are quite fragile: a mobile computer may run out of battery power, be damaged in a fall, be lost, or be stolen [47, 71]. Given these threats and the limited amount of storage available to mobile computers, primary ownership of data should reside with stationary computers, not mobile computers. Furthermore, application developers should take special precautions to enhance the resilience of the data stored on mobile computers (*e.g.,* storing changes to application data using stable storage techniques).

A stationary environment can distribute an application's components and rely upon the use of high-bandwidth, low-latency networks to provide good interactive application performance. Mobile computers operate primarily in a limited bandwidth, high-latency, and intermittently-connected environment; nevertheless, users want the same degree of responsiveness and performance as in a fully-connected environment.

Network partitions are an infrequent occurrence in stationary networks; therefore, most applications consider them to be major failures that are exposed to users. In the mobile environment, applications will face frequent, long-duration network partitions. Some of the partitions will be involuntary (*e.g.,* due to a lack of network coverage or radio shadows), while others will be voluntary (*e.g.,* due to high monetary cost or limited battery power). Applications should gracefully, and as transparently as possible, handle such partitions. In addition, users should be able to continue working (*i.e.,* accessing and modifying locally cached data) as if the network was still available, albeit with some limitations.

When users on different sides of a network partition modify local copies of global data, consistency is an issue. In a mobile environment, optimistic concurrency control [49] is useful because pessimistic methods are inappropriate (*e.g.,* a disconnected user cannot acquire or release locks), as pointed out by the designers of the Coda file system for disconnected operation [47]. However, using an optimistic approach has its costs: associated with long duration partitions, will be a greater incidence of update-update conflicts than in stationary environments. It is therefore important to use application-specific semantic information both to detect when such conflicts are false positives and can be avoided and, when possible, to help resolve such conflicts.

While network bandwidth in the stationary environment is usually symmetric, in the mobile environment it may be asymmetric. Also, wireless network bandwidth is a limited, shared resource and currently, is less than wired bandwidth. Stationary servers have access to unlimited power for transmitting and to larger, better antennas for receiving. Mobile computers, however, have limited power available for transmission and smaller, more portable antennas. Thus, servers may be able to transmit to mobile computers at a higher data rate than mobile computers can transmit to servers.

As discussed earlier, the differences in resources and their volatility represent significant problems for mobile computers. Unfortunately, these are not problems that technological improvements will fix. Size, weight, and cooling constraints for mobile computers mean that computational resources and battery power will always be constrained. The only solution to these problems is to redesign the classical distributed systems techniques normally used in stationary environments.

In summary, the mobile environment represents an extreme case of distributed computing. Certain of the algorithms used by distributed algorithms are based upon assumptions that are invalid in the mobile environment.

In contrast, the algorithms used by the Rover toolkit are based upon distributed algorithms that have been adapted and optimized for the mobile environment. Placing these algorithms in the toolkit provides application developers with the reusable building blocks that they need to cope with environmental volatility. In addition,

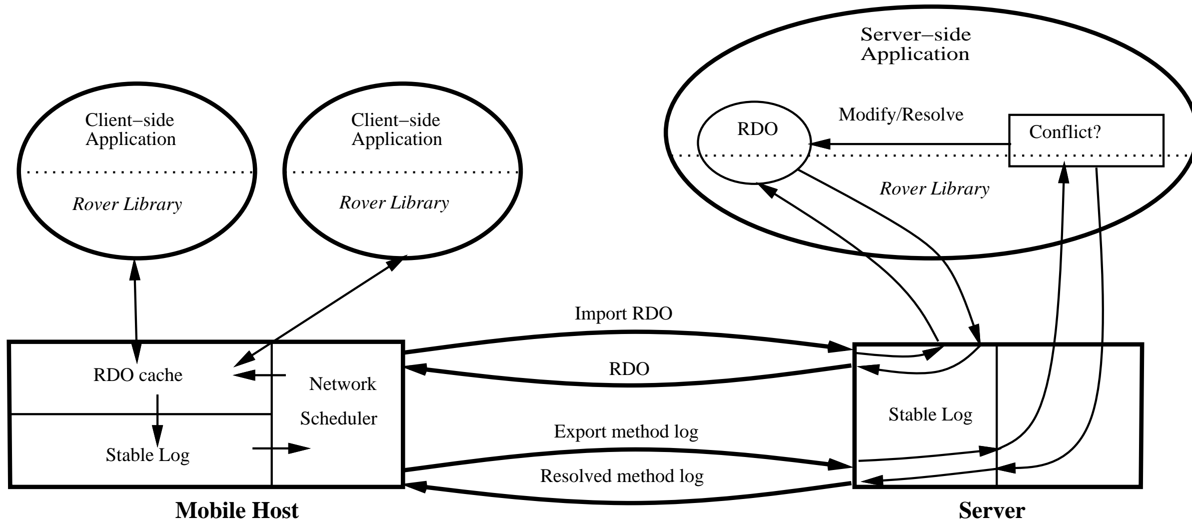


Figure 1-1: Rover offers applications a client/server distributed object system with client caching and optimistic concurrency control.

using a toolkit simplifies the development of mobile applications.

1.2 Goals

This thesis has several goals that directed the design and architecture of the ideas in the Rover toolkit:

1. Users want applications that provide user interfaces that allow users to continue working while disconnected. Applications should be able to enqueue non-blocking requests that will be delivered when connectivity becomes available. In addition the applications should be able to cache data, and modify that cached data, while intermittently connected or disconnected.

2. Network connectivity in the wide-area is limited and costly both in terms of dollars and transmission power; so every effort should be made to efficiently use available connectivity. In addition, there is a significant amount of existing wide-area network infrastructure; every effort should be made to reuse it (*e.g.,* World-Wide Web, E-mail, etc).

3. Clients may be temporarily computationally underpowered, or servers may be

temporarily overloaded; as such, applications should be able to dynamically shift work between clients and servers.

4. The application development environment should isolate developers from the mobile environment's problems and help them cope with the mobile environment's volatility. At the same time, the application development environment should expose environmental (*e.g.,* connectivity) information to applications, if the applications desire the information.

5. Application developers should be provided with an efficient application development environment; one that allows them to incorporate existing code and applications, quickly create new mobile-adaptive applications, and provides developers with an incremental transition from mobile-transparent proxies to mobile-adaptive applications.

1.3 Rover: A Toolkit-based Approach

The Rover toolkit's design and architecture address the goals listed in the previous section by providing applications with a general-purpose distributed object system based on a client/server architecture [16, 38, 41, 40, 74] (see Figure 1-1). Clients are Rover applications that typically run on mobile hosts, but could run on stationary hosts as well. Servers, typically run on stationary hosts and hold the long term state of the system. Rover applications employ a primary-copy, tentative-update, optimistic concurrency-based distributed object model of data sharing: they call into the Rover library to *import* objects, and to *export* logs of methods that mutate objects. Client-side applications *invoke* operations directly on locally cached objects. Server-side applications are responsible for resolving conflicts and notifying clients of resolutions. A network scheduler drains the stable queued communications log, which contains the requests and methods that are to be performed at the server.

Communication between clients is limited to mobile host–server interactions and peer-to-peer interactions between clients sharing the same local object cache. Such

clients need not be collocated on the same mobile host. There is, however, no support for remote peer-to-peer (clients using different local object caches) or mobile host–mobile host (object cache—object cache) interactions.

The Rover toolkit provides mobile communication support based on the novel combination of two ideas: *relocatable dynamic objects* (RDOs) and *queued remote procedure call* (QRPC). A relocatable dynamic object is an object (code and data) with a well-defined interface that can be dynamically loaded into a client computer from a server computer, or vice versa, to migrate computation or data and reduce client-server communication requirements. Queued remote procedure call is a communication system that permits applications to continue to make non-blocking or asynchronous remote procedure calls [8] even when a host is disconnected — requests and responses are exchanged upon network reconnection. The synergy between QRPC and RDOs helps applications deal with volatility in the mobile environment.

The key task of the programmer when using the Rover toolkit to build a mobile-adaptive application or a proxy for mobile-transparent applications is to define RDOs for the data types manipulated by the application or proxy, and for data transported between client and server. The programmer then divides the application or proxy into portions that run on the client and portions that run on the server; these parts communicate by means of QRPC. The programmer then defines methods that update objects, including code for conflict detection and resolution.

To use the Rover toolkit, a programmer links the modules that compose the client and server portions of an application with the Rover toolkit. The application can then actively cooperate with the Rover runtime system to *import* objects onto the local machine, *invoke* well-defined methods on those objects, *export* logs of method invocations on those objects to servers, and *reconcile* the client's copies of the objects with the server's.

1.4 Building and Using Mobile Applications

The Rover toolkit provides a novel approach to building and using mobile applications. This section discusses two other categories of classical distributed computing techniques for building client-server applications for mobile environments. The development of both approaches has been guided by the attributes of the typical stationary environment and they fall into one of two categories: those approaches that are special-purpose, single-use implementations and those approaches that are unaware of the environment.

Special-purpose applications use a special-purpose implementation of the network transport and data management protocols [33, 47, 68, 71]. Such applications are able to address all of the differences between stationary and mobile environments. However, because the network and data infrastructure is very specialized, it may be difficult to implement, extend, upgrade, or reuse for other applications. In addition, there is no coordination of resources between multiple applications executing on the same mobile host. This lack of coordination on a mobile host results in all of the applications competing in an uncoordinated fashion for the same limited network and computational resources. An example of the problems that this lack of coordination introduces is provided in Section 6.4.

Mobile-transparent applications, assume that they are operating under the conditions of a stationary environment; therefore, they make certain, potentially incorrect, assumptions about the location and availability of resources. However, mobile-transparent applications can be used unmodified in mobile environments by having the operating system shield or hide the differences between the stationary and mobile environments from the applications. Coda [47] and Little Work [31] used this approach by providing a file system-based interface to applications. These systems consist of a local *proxy* for some service, in this case the file system, running on the mobile host. The proxies are themselves mobile-adaptive applications and they provide the standard service interface to the application, while attempting to mitigate any adverse effects of the mobile environment and cooperating with remote proxies

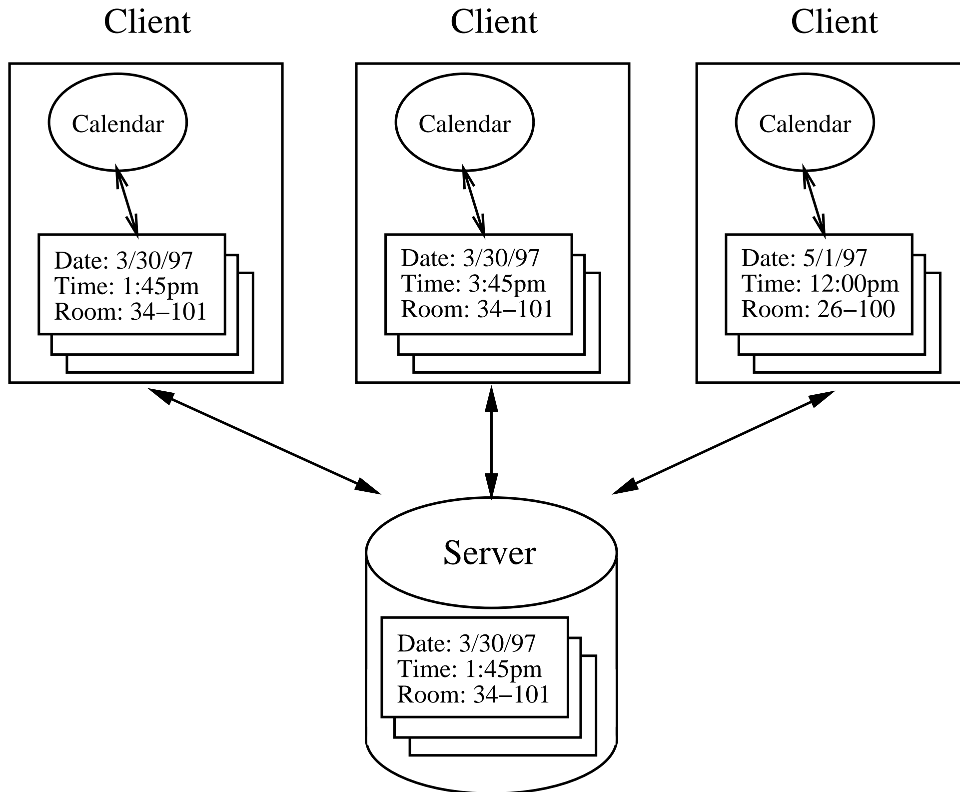


Figure 1-2: Inconsistent replicas in a replicated file system.

on well-connected, stationary hosts.

Unfortunately, the mobile-transparent approach often sacrifices functionality and performance. While proxies hide mobility issues from applications, they usually require manual intervention by users (*e.g.,* having a user indicate which data to prefetch onto the user's computer). The limitations and semantics of a mobile-transparent interface may make it more difficult to provide a good user interface. In some situations, it may not be possible to decouple user actions from the network (*e.g.,* most file system proxies block when a user requests an uncached file and network connectivity is not available).

Similarly, the resolution of update-update conflicts (conflict resolution) is complicated because the interface between the application and its data was designed for a stationary environment. Consider an application writing records into a file shared among stationary and mobile hosts. While disconnected, the application on the mobile host inserts a new record into the file. The local file system proxy records the

write in a log. Meanwhile, an application on a stationary host alters another record in the same file. Upon reconnection, the file system can detect that conflicting updates have occurred. However, without semantic information about the application or its files, the file system cannot resolve the conflict.

Coda recognizes this limitation and provides for the use of *application-specific resolvers* (ASRs) [48]. However, ASRs alone are insufficient. In the above example, there is no way for the ASR to use the file system interface to determine whether the mobile host inserted a new record or the stationary host deleted an old one. The cause of this confusion is that Coda changes the contract between the application and the file system in order to hide the condition of the underlying network. As Figure 1-2 shows, the read/write interface no longer applies to a single file, but to possibly inconsistent replicas of the file. Therefore, any applications that depend on the standard read/write interface for synchronization and ordering may fail.

1.5 Arguments for Mobile-Adaptive Computing

Although the mobile-transparent approach is appealing because it offers to run existing applications without alteration, it is fundamentally limited. The functionality needed to create correct, well-performing applications in an intermittently-connected environment often requires the cooperation of both application and user. The alternative to hiding environmental information from applications is to expose the information to the applications and involve them in decision-making. Exposing environmental information yields the class of *mobile-adaptive* applications.

A mobile-adaptive application can store not only the *value* of a write, but also the *operation* associated with the write. Knowing the operation adds a significant amount of application-specific semantic information; for example, it allows for "on-the-fly" dynamic construction of conflict resolution procedures.

Unlike previous systems, the Rover toolkit is designed to support both mobile-adaptive and mobile-transparent approaches. It also is designed to help application developers to gradually and incrementally port existing applications to a mobile-

adaptive environment. For mobile-transparent applications, the Rover toolkit provides the tools to build local proxies for remote services. For mobile-adaptive applications, the Rover toolkit provides the components and architecture necessary for operation in a mobile environment.

The mobile-adaptive argument can be viewed as applying the end-to-end argument from [70] to mobile applications: "Communication functionality can be implemented only with the knowledge and help of the application standing at the endpoints of the communications system." The file system example described above illustrates that applications need to be aware of intermittent network connectivity to achieve consistency. Similar arguments can be made with respect to performance, reliability, low-power operation, etc.

The special-purpose, single-use applications discussed in the previous section use the mobile-adaptive argument. However, the mobile-adaptive argument does not require that every application use its own, *ad hoc* approach to mobile computing. On the contrary, it allows the underlying communication and programming systems to define an application programming interface that optimizes common cases and supports the transfer of appropriate information between the layers. Since mobile-adaptive applications share common design goals, they will share design features and techniques. The Rover toolkit provides exactly such a mobile-adaptive application programming interface.

1.6 Contributions and Results

In addition to achieving the goals listed earlier in Section 1.2, this thesis offers several key contributions and results:

1. QRPC meshes well with intermittently connected environments. QRPC performance is acceptable even if every RPC is stored in stable logs at clients before being transmitted and at servers before being processed. For lower-bandwidth and higher-latency networks, the overhead of using stable logs is dwarfed by the underlying communication costs.

2. Using stable message queues enables QRPCs to be scheduled, yielding more efficient use of communication channels and increased network performance. The network scheduler uses batching and compression to reduce the number of roundtrip messages and network connections used by QRPCs. This decrease in messages reduces the average latency for a request and minimizes the total amount of data sent over communication channels.

3. The reliable delivery transport offered by QRPC greatly simplifies the development of mobile applications by hiding communication failures from applications.

4. Use of RDOs allows mobile-adaptive applications to migrate functionality dynamically to either side of a slow network connection to minimize the amount of data transiting the network. Caching RDOs reduces latency and bandwidth consumption. Interface functionality can run at full speed on a mobile host, while large data manipulations may be performed on the well-connected server.

5. RDOs also allow computation migration decision based based upon available computation resources. Computation can be offloaded from a temporarily computationally underpowered client or a temporarily overloaded server. Thus, computation migration aids in making Rover scalable to large numbers of clients.

6. Long-running applications are usually the most likely to be affected by software and hardware faults. Rover supports the development of reliable applications by providing reliable server-side RDOs based upon QRPC's reliable delivery, several simple language extensions, and failure recovery functions.

7. Experiences with building several mobile-adaptive applications shows that porting existing applications or building new applications using Rover generally requires relatively little change to the original application. In addition, the new applications perform significantly better than the original applications.

8. Experiences with building several proxies to support mobile-transparent applications shows that building Rover proxies is fairly easy. In addition, unmodified,

mobile-transparent applications show significant performance benefits from using Rover proxies.

9. One of the proxies constructed with the Rover toolkit is an World-Wide Web browser proxy. The proxy supports offline or intermittently connected web browsing using unmodified web browsers. Several of the techniques used by the proxy have been adopted by commercial applications or have been proposed as components of World-Wide Web (WWW) standards.

The thesis includes several software components, a publicly available reference implementation of the Rover Toolkit along with several example applications (E-mail browser, graphical rolodex, and HTTP proxy), and on-line documentation of the toolkit. There also will be internally available prototypes of an NNTP reader proxy [13], distributed calendar tool, and the Rover File System proxy. Additional information about the software components is available from the Rover project home page: http://www.rover.lcs.mit.edu/ .

The software components are provided as examples of how the ideas presented in this thesis can be used; other implementations of the Rover toolkit and applications may make different implementation choices. The components are also prototypes; as such, not all of the features and functionality discussed in this thesis are implemented, including: several of the client cache consistency options; messages are sent unencrypted; limitations on the sharing of data between client applications on the same machine; limited support for handling errors. Additional details can be found in Chapter 4.

1.7 Outline of the thesis

The remainder of this thesis places the research in the context of related work (Chapter 2), presents the design and architecture of the Rover toolkit (Chapter 3), describes the reference implementation of the Rover toolkit (Chapter 4), discusses the implementation of several mobile-adaptive applications and proxies for mobile-transparent

applications (Chapter 5), presents experimental results from micro benchmark and system level measurements (Chapter 6), and finally, offers observations on future work and the benefits and limitations of the Rover approach (Chapter 7).

Chapter 2

Related Work

To the best of my knowledge, no one has studied an architecture like Rover's, which provides both queued RPC and relocatable dynamic objects. Queued RPC is unique in that it provides support for asynchronous fetching of information, as well as for lazily queuing updates. The use of relocatable dynamic objects for dealing with the constraints of mobile computing — intermittent communication, varying bandwidth, and resource poor clients — is also unique to the Rover architecture. There are, however, a number of research and commercial projects that are related to the Rover Toolkit. This chapter discusses these related projects in detail.

2.1 Relocatable Dynamic Objects

The Java [3, 73] language provides a limited framework for code-shipping and remote invocation and could have been used for the language component of RDOs. However, the Tcl and Tk languages [66] were chosen for the language component for RDOs because Java was not yet available during the early development of Rover. In addition, Java has only recently become as portable as Tcl. Future Rover implementations may be based upon Java or other byte-compiled or interpreted languages.

The important consideration here is that the choice of which code-shipping language to use for RDOs is immaterial because the particular form of code shipping is orthogonal to the Rover architecture. The key difference between Rover and other code

shipping systems is that Rover provides RDOs with a well-defined object-based execution environment that provides code- and data-shipping, a uniform naming scheme, an application-specific replication model, reliable execution, and QRPC.

The Bayou project [17, 77] defines a mobile-adaptive, peer-to-peer, database architecture for sharing data among mobile users, where mobile hosts store updates locally in a stable log and communicate with other mobile hosts to propagate the changes. Conflicts are resolved using log replay. Bayou addresses the issues of tentative data values [78] and session guarantees for weakly-consistent replicated data [75]. To illustrate these concepts, the authors have ported calendar and E-mail applications and built a bibliographic database tool. Rover borrows the notions of tentative data and the calendar tool example from the Bayou project. Rover extends this work with RDOs and QRPC to deal with intermittent communication, limited bandwidth, and resource poor clients.

An alternative to the Rover object model is the Thor object model [54]. In Thor, objects are updated within transactions that execute entirely within a client cache. However, Thor does not support disconnected operation: clients have to be connected to the server before they can commit. An extension for disconnected operation in Thor has been proposed by Gruber and others [30], but it has not been implemented. Furthermore, it does not provide a mechanism for non-blocking communication, and their proposed object model does not support method execution at the servers.

RDOs can be viewed as simple Agents [69] or as a light-weight form of process migration [22, 67, 72, 79]. Other forms of code shipping include Display Postscript [1], Safe-Tcl [9], Active Pages [32], Dynamic Documents [42], and LISP Hypermedia [56]. RDOs are probably closest to Telescript [87], and Ousterhout's Tcl agents [66]. As discussed above, most differences between RDOs and these other forms of code shipping are immaterial because of the environment provided by the Rover architecture.

My research borrows from early work on replication for non-mobile distributed systems. In particular, Rover borrows from Locus [82] (type-specific conflict resolving) and Cedar [26] (check-in, check-out model of data sharing).

The Rover architecture supports data replication using a primary copy replication

with tentative updates model. Gray *et. al.* [27] perform a thorough theoretical analysis of the options for database replication in a mobile environment and conclude that primary copy replication with tentative updates is the most appropriate approach for mobile environments.

2.2 Queued Remote Procedure Call

A number of proposals have been made for dealing with the limited communication environments for mobile computers. Katz surveys many of the challenges [43]. Baker describes MosquitoNet, which shares similar goals with Rover, but has not been implemented yet [5]. Several commercial systems for mobile environments, including Telescript [87] and Oracle Mobile Agents [12], offer asynchronous communication and reliable message delivery.

Several systems use E-mail messages as a transport medium, and obtain benefits similar to those obtained using QRPC. The Active Message Processing project [81] has developed various applications, including a distributed calendar, that use E-mail messages as a transport medium. In another project, researchers at Digital Equipment Corporation's Systems Research Center (DEC SRC) used E-mail messages as the transport layer of a project that coordinated more than a thousand independently administered and geographically dispersed nodes to factor integers of more than 100 digits [52]. The application is a centralized client-server system with one server at DEC SRC that automatically dispatches tasks and collects results.

There are several research and commercial research projects that use queued communication to support intermittently connected and/or offline World Wide Web browsing. The DeckScape WWW browser [10] is a "click-ahead" browser that was developed simultaneously with the Rover web browser proxy. However, their approach was to implement a browser from scratch; as such, their approach is not compatible with existing browsers. There currently are a number of commercial offline web browser proxies available (*e.g.,* IBM's ARTour Web Express [11]) that are compatible with existing browsers, however all are based upon special-purpose implementations.

IBM's proxy is particularly interesting because it optimizes the communications link between the local proxy and a remote IBM proxy using compression and differencing techniques.

2.3 Mobile-Transparent File Systems

Several previous projects have studied building mobile-transparent services for mobile clients; most have been centered around file systems. The Coda project pioneered the provision of distributed services for mobile clients. In particular, it investigated how to build a mobile-transparent file system proxy for mobile computers by using optimistic concurrency control and prefetching [47, 71]. Coda logs all updates to the file system during disconnection and replays the log upon reconnection. Coda provides automatic conflict resolution mechanisms for directories and files, and uses Unix file naming semantics to invoke application-specific conflict resolution programs at the file system level [48]. A manual repair tool is provided for conflicts of either type that cannot be resolved automatically. A newer version of Coda also supports low bandwidth networks, as well as intermittent communication [58].

The Ficus file system is also a mobile-transparent file system supporting disconnected operation, but it relies on version vectors to detect conflicts [68]. The Little Work project caches files to smooth disconnection from an AFS file system [34]. Conflicts are detected and reported to the user. Little Work is also able to use low-bandwidth networks [33].

A follow-on to the Coda project, the Odyssey project retains a file system-based interface, but modifies the operating system by adding an application programming interface for application awareness of the mobile environment and application adaptation to environmental changes [64, 65]. Odyssey is discussed in more detail in the next section.

2.4 Mobile-Adaptive Applications

The need for mobile-adaptive applications and complimentary system services to expose mobility to applications was identified concurrently by several groups. Katz noted the need for adaptation of mobile systems to a variety of networking environments [43]. Davies *et. al.* cited the need for protocols to provide feedback about the network to applications in a vertically integrated application environment [15]. Similarly, Kaashoek *et. al.* created a Web browser which exposed the mobile environment to mobile code that implemented mobile-adaptive Web pages [42]. Baker identified the dichotomy between mobile-adaptive and mobile-transparent issues in general application and system design [5]. Rover is the first implemented general application architecture to support both mobile-transparent system service proxies and mobile-adaptive applications.

The BNU project implements a framework for executing RPC-driven mobile-transparent applications on mobile computers. It allows for function shipping by downloading Scheme functions for interpretation [85]. The BNU environment includes mobile-transparent proxies on stationary hosts for hiding the mobility of the system. BNU applications do not dynamically adjust to the environment; they do not have a concept of tentative or stale data; and there is no support for disconnected operation, such as Rover's queued RPC. Application designers for BNU noted that the workload characterizing mobile platforms is different from workstation environments and will entail distinct approaches to user interfaces [50]. A follow-up project, Wit, addresses some of these shortcomings and shares many of the goals of Rover, but employs different solutions [84].

A number of proposals have been made for various degrees of mobile-adaptive support in operating system services and application. The InfoPad [51], Daedalus [59], GloMop [24] and W4 [6] projects focus on mobile-adaptive wireless information access. The InfoPad project employs a dumb terminal and offloads all functionality from the client to the server. Daedalus and GloMop use dynamic "transcoding" or "distillation" to reduce the bandwidth consumed by data transmitted to a mobile host.

Odyssey Benefit	**Odyssey Limitation**
Adaptation of multimedia applications	Read-only support, no support for write operations
Centralized resource management	No support for disconnected or intermittently connected operation
Model for data fidelity	Single dimension of adaptation
Server to client data migration	No support for code migration

Table 2.1: Benefits and limitations of the Odyssey project.

Their transcoding technology is completely compatible with Rover's architecture. Applications on the mobile host cooperate with mobile-adaptive proxies on a stationary host to define the characteristics of the desired network connections. Similarly, W4 applies the technique of dividing application functionality between a small Portable Data Assistant (PDA) and a powerful, stationary host to Web browsing. Rover is designed for more flexible, dynamic divisions. Depending on the power of the mobile host and available bandwidth, Rover allows mobile-adaptive browsers to dynamically move functionality between the client and the server.

The BARWAN [44] project supports mobile, "data type aware" applications. The approach relies on strongly typed transmissions. A dynamically extensible type system enables type-specific compression levels and abstraction mechanisms to conserve network usage. User code is itself a transmission type allowing computation relocation. Davies' Adaptive Services [15] similarly takes a protocol-centric approach for exposing information about the mobile environment to the application.

A similar approach is taken by the Odyssey project. Odyssey focuses on operating system support to enable "agile" mobile-adaptive applications to use "data fidelity" to control resource utilization. Data fidelity is defined as the degree to which a copy of data matches the original [64, 65]. Odyssey provides centralized resource management and uses application-specific procedures to deal with changes in available resources. Odyssey has been used to implement a variety of read-only mobile information access applications. Table 2.1 lists Odyssey's advantages and disadvantages.

Rover, however, in addition to supporting dynamic adaptation of program func-

tionality and data types, also supports application-specific conflict detection and resolution.

A number of successful commercial mobile-adaptive applications have been developed for mobile hosts and limited-bandwidth channels. For example, Qualcomm's Eudora is an E-mail browser that allows efficient remote access over low-bandwidth links. Lotus Notes [45] is a groupware application that allows users to share data in a weakly-connected environment. Notes supports two forms of update operations: append and time-stamped. Conflicts are referred to the user. TimeVision and Meeting Maker are group calendar tools that allow a mobile user to download portions of a calendar for off-line use. The Rover toolkit and its applications provide functionality that is similar to these proprietary approaches, however it does so in an application-independent manner. Using the Rover toolkit, standard workstation applications, such as the *Exmh* E-mail browser and the *Ical* distributed calendar, can be easily turned into "roving" mobile applications.

2.5 Reliable Execution

Some of the ideas behind the Rover toolkit's reliable execution model were drawn from existing work in distributed computing. Applying these ideas to the Rover toolkit moves them to a domain where failures are frequent and network transports offer intermittent connectivity, high latency, and low bandwidth.

There is a large body of research on logging and distributed fault-tolerant transactions; for an excellent discussion of some of the issues, see [28] and [29].

Other systems have addressed some of the problems relating to reliable communications. The Tacoma project explored the use of rear guard agents to guarantee agent delivery and execution [37]. More recent work on Tacoma relies upon Horus for fault-tolerant communication and execution [80]. ISIS defined an environment for fault-tolerant computing based on group communication [7]. The failure model used by ISIS is a fail-stop model, which requires recovering processes to recover their state from other active processes instead of a log. Rover's failure model supports recovery

based upon either client computer retransmissions (active processes) or server stable logs.

The state of guardians in the transaction-based Argus system is split into stable and volatile variables [53]. Recovery relies upon replay of a local stable log. Likewise, the state of objects in the Clouds distributed operating system project [2] was split into permanent and volatile data. Clouds also provided computation fault-tolerance support for mobile objects by using primary and backup schedulers. Rover also splits server application state into stable and volatile variables.

Chapter 3

Design of the Rover Toolkit

The Rover toolkit is designed to support the construction of mobile-adaptive applications and proxies for mobile-transparent applications. This chapter discusses the design and architectural choices that were made during the development of the Rover toolkit. The first section provides an overview of the ideas in the toolkit followed by sections discussing Relocatable Dynamic Objects, Queued Remote Procedure Call, RDO replication and consistency issues, and mobile-adaptive applications.

3.1 Design Overview

This section provides an argument for using a toolkit, discusses the fault model for the Rover toolkit, and details the problems addressed by the toolkit and the four solutions offered by it:

1. Relocatable Dynamic Objects (RDO)
2. Queued Remote Procedure Call (QRPC)
3. Replication and application-specific consistency control
4. Mobile-adaptive applications

3.1.1 Why Use a Toolkit?

One of the key contributions of this thesis is a toolkit for transferring existing, mobile-transparent applications into a mobile environment and for constructing new, mobile-adaptive applications. An alternative approach is single-use, special-purpose implementations. However, toolkits offer code reuse, a distinct advantage over special-purpose implementations. Code reuse means that the functionality offered by the toolkit only needs to be implemented, debugged, and optimized once — code reuse simplifies application porting, development, and correctness. Another, sometimes complementary, approach is operating system support. Toolkits offer the advantage over operating system implementations of portability to other operating systems and preservation of application data type semantics.

One can ask the converse question of why a toolkit should not be used. Some of the potential reasons why one would not use a toolkit are: efficiency of the toolkit, flexibility of Application Programming Interfaces (APIs), and customizability of the APIs.

Rover provides developers with a spectrum of customizability. Developers may choose to use Rover-supplied functionality and components, or they may choose to replace components with custom functionality. Obviously, there are situations where a custom implementation will offer better performance than a toolkit. However, a properly designed and implemented toolkit will often offer sufficient efficiency with the benefit of reusability. For example, the implementation of the client-side Rover toolkit uses a multi-level cache implementation to optimize client application performance (see Section 4.2.2 for more details). In addition, a toolkit is designed to be reused; thus, it is reasonable to expect that more time will be spent on optimizing the toolkit's performance, than would be spent on a single application.

Flexibility and customizability of a toolkit are important considerations as they determine the toolkit's applicability to a given application domain. A significant amount of effort has gone into the design of the application programming interfaces for the Rover toolkit. There are, however, application domains that are not addressed by the

Rover toolkit (*e.g.,* real-time and multimedia applications).

Although a toolkit is located above the operating system, it can be viewed as serving a similar purpose: providing an abstract interface to the underlying environment. Thus, just as application developers do not typically implement custom operating systems for each application that they develop, mobile application developers need not implement special-purpose functionality for their applications.

3.1.2 Failures

The Rover toolkit is designed to handle several types of transient failures, some of which are common in the mobile environment. The most commonly encountered failures in the mobile environment are transient communication link failures (*e.g.,* a dropped dialup link, a packet lost due to a wireless link error, or a packet dropped by a congested router) and power losses (*e.g.,* an exhausted battery).

Other less common failures that are addressed by the toolkit are transient software failures, or *Heisenbugs*. Software faults occur mostly in the form of benign, transient failures caused by programming errors in rarely executed code paths (*e.g.,* race conditions), by resource exhaustion, or by transient hardware errors. The best recovery mechanism for transient failures is to restart the application or system [29].

One of the important guarantees offered by Rover is a delivery guarantee: changes to application data will be delivered from client-side applications to servers and the results delivered back to the client machines, regardless of any *transient* faults. Applications can trust that once a request is inserted into the stable log, it will be delivered to the server. This is an important guarantee considering the potential problems that may be encountered in the mobile environment. By providing application developers with this guarantee, Rover greatly simplifies the application developers' problem of delivering user's modifications to servers.

The delivery of results is only guaranteed to reach the client machine and not the original client application. Rover provides support for failure recovery only for client-side Rover toolkit failures and not for preserving the internal state of client-side applications across client failures. A client application that fails will not be automati-

cally restarted and all application state, except for that stored in the persistent RDO cache or stable log, is lost.

To help protect against client failures, the data stored at clients is considered *tentative* and, unless it was locally created, is a copy of data stored at a server. Additional client-side application support for failure recovery is an area for future research. However, regardless of transient failures of the client machine or application, communication links, or server machine, the user's modifications will be delivered to the server and the results returned.

For server-side applications, Rover provides extensive support for preserving their internal state across server failures. The toolkit support for reliable server-side applications is discussed in more detail in Sections 3.2.2 and 3.3.2.

The Rover toolkit does not address persistent software, hardware, or communication link failures. Examples of persistent hardware faults are failed components, in particular, hard disk drives, and total loss situations (theft, fire, etc.). Persistent software failures, or *Bohrbugs*, are faults that are not transient; they recur when the operation is reexecuted (*e.g.,* they are due to programming error). In addition, they may corrupt system state, leaving the system unrecoverable even after a restart.

There are many research techniques that can be applied to persistent faults [4], however, for most applications, providing a system that can tolerate persistent failures would impose an unnecessary burden on programmers, performance, and hardware. In addition, past studies have shown that most hardware and software errors are transient, recoverable failures [29]. As such, the Rover toolkit model balances the need to hide hardware and software faults with the need to avoid overly burdening programmers and lowering performance in the normal case.

3.1.3 Mobile Environment Problems Addressed by Rover

The Rover toolkit is designed to address the problems and challenges associated with mobile environments, as described in Section 1.1:

1. The volatile nature of the mobile environment — Use QRPC, replication, and

application-awareness to dynamically migrate RDOs. These Rover solutions allow applications to adapt to changes in available computational and communication resources. Use application-specific consistency control for the problems related to the high latency for propagating changes to data in the mobile environment.

2. Unpredictable remote access times in the mobile environment — Use QRPC to isolate applications from unpredictable latency and bandwidth. Use RDOs and replications to move application data and functionality closer to the resources that they need (*e.g.,* move GUIs to users, move functions to the data that they use). Use application-specific consistency control to handle changes to data during periods of disconnected or intermittently connected operation.

3. Limited computational and power resources — Use QRPC's split-phase operation (discussed in Section 3.3.2) for more power-efficient communication. Migrate RDOs to minimize bandwidth requirements (transmitter time), to hide latency from applications and users, and to handle constrained computational resources at clients or servers.

4. Users' desire for good interactive performance and application availability, regardless of available network connectivity — Replicate and migrate application data and GUI RDOs to give users good interactive performance by hiding network latency. Use application-specific consistency control to allow users to make immediate changes and propagate results in the background, while handling potential conflicts.

Rover Solution #1: Relocatable Dynamic Objects

Rover provides RDOs to help applications adapt to volatility in the computational, network, and power resources that are available in the mobile environment. RDOs also help applications use computation relocation to decrease both application dependence upon continuous network connectivity and the amount of data sent between clients and servers.

Some of the alternatives to the dynamic object model offered by RDOs are a static object model (*e.g.,* a fixed client-server division) or a file system-based model. Each of these alternatives has some deficiencies relative to RDOs. For example, a static object model cannot adapt to changes in the mobile environment (computational, network, and power resources). A file system-based model (even a mobile-adaptive one, such as Odyssey [65]) uses objects, files, that are coarse-grain and hides application-specific semantic information that is needed during conflict detection and resolution.

Rover Solution #2: Queued Remote Procedure Call

Rover provides QRPC — asynchronous, queued communication with optional *call-backs* — to help isolate applications from both the limitations of networks (bandwidth, latency, and cost) in the mobile environment and the unreliability of the mobile environment (*e.g.,* transient hardware and software failures and limited battery power).

By using QRPC with its logging of requests to stable storage, Rover is able to offer client applications the delivery guarantee discussed earlier: *Once a QRPC is received by the toolkit, it will be delivered to the intended recipient and the results will be delivered back to the client's machine, regardless of any transient faults*. Rover does not guarantee delivery in the presence of persistent faults and that the delivery of results back to the caller is not guaranteed, since the caller application may have failed in the meantime.

In addition, as described in more detail in Section 3.5, applications can modify and delete QRPCs that they have issued, but that have not already been sent to a server.

Two alternatives to using QRPC for communication are operating system-based communication and application-specific communication. In the former case, the common communication support usually offered by operating systems is not well suited to the mobile environment, while special operating system support is usually not portable, is based upon a file system interface that hides application-specific semantics that may be useful in detecting and resolving conflicts, and does not offer the delivery guarantees offered by Rover. The Rover toolkit is designed to be portable

across a variety of operating systems and to preserve application-specific semantics.

The other alternative, application-specific communication, makes code reuse difficult, an important consideration given the amount of effort required to implement the functionality incorporated in the Rover toolkit. Application-specific communication also prevents the coordination and management of the communication by multiple client applications with remote servers (*e.g.,* using a single network connection to send data from separate client applications to a common server machine). The Rover toolkit is designed for easy code reuse and QRPC supports several features for optimizing communication between separate client applications and servers.

Rover Solution #3: Replication and Application-Specific Consistency Control

The Rover toolkit provides users with the capability to continuously access and modify locally cached data; this support is provided in the form of application-specific conflict detection and resolution. The toolkit gives client and server applications the application-specific tools they need to control both the replication of application data and the degree of consistency between client and server replicas. Rover provides a significant amount of support for optimistic concurrency, but does not preclude the use of other concurrency models.

Rover Solution #4: Mobile-Adaptive Applications

Since the mobile environment is dynamic, it is important to present users and applications with information about the current environment. The Rover toolkit provides applications with environmental information (*e.g.,* network connectivity options, maximum batching delay, compression level, and the contents of the persistent RDO cache and stable log) for use in dynamic decision making or for presentation to the user. Applications may use either polling or callback models to determine the state of the mobile environment.

Applications can choose to forward notifications to users or use them for silent policy changes. For example, in the Rover calendar application (see Section 3.5),

appointments that have been modified but not propagated to the server are displayed in a distinctive color (a technique that was borrowed from the Bayou room scheduling tool [17]). The color informs users that the appointment is tentative and might be canceled due to a conflict.

3.1.4 The Synergy Between the Solutions

There is a synergy between RDOs and QRPC — QRPC provides the transport layer for moving RDOs and conveying remote invocations upon RDOs. Rover clients and proxies use QRPC to lazily **import** RDOs from servers (see Figure 1-1). When a client application or proxy issues an import or export request for a remote RDO, Rover automatically turns it into a QRPC containing the request. By using QRPC as a transport mechanism, applications and their actions are automatically provided with isolation from the volatile and fragile nature of the mobile environment.

Likewise, RDOs rely on the toolkit's replication and application-specific consistency control to provide users with continuous access and modification rights to their data along with coherent views of that data.

Finally, RDOs, QRPC, replication, and application-specific consistency control are all linked via mobile-adaptive application support. Mobile-adaptive support gives applications control over the other three solutions.

3.2 Relocatable Dynamic Objects

Relocatable Dynamic Objects (RDO) are the central data structures in the Rover toolkit. This section details the components of an RDO, discusses the lifetime of and uses for an RDO, and provides the security model used for RDOs.

3.2.1 RDO

An RDO is an object that consists of: mobile code, encapsulated data, a well-defined interface that specifies the methods provided by the RDO, and outcalls — the ability

to make invocations from one RDO to another.

Associated with an RDO is a method log. At the client, an RDO's method log contains any *tentative* mutating methods (methods with side effects) that have been applied to the RDO, but which have not been declared as committed by the server. Included with the mutating methods is any data created for or used by the method. Client RDOs exist in one of two states:

1. *Committed*. This state indicates that the RDO is a locally unmodified copy.

2. *Tentatively committed*. This state indicates that the RDO has been locally modified, but the changes have not yet been committed by the server. When a mutating method is invoked upon an RDO, the toolkit automatically clones the committed version of an RDO and applies the method to one of the copies; this action creates a tentatively committed RDO. Mutating methods are discussed in more detail in Section 3.2.2.

At the server, an RDO's method log contains a record of the methods that have been applied to the RDO; this log represents the RDO's history.

RDOs may vary in complexity from simple objects with a small set of methods (*e.g.,* calendar items) to modules that encapsulate a significant part of an application (*e.g.,* the graphical user interface for an E-mail browser). Complex RDOs may create a new thread of control when they are imported.

Servers have multiple options for transmitting RDOs to clients. If the client does not have a local replica of an RDO, the server marshalls the entire RDO and sends it. Otherwise the server uses one of two formats, either the entire RDO is marshelled and sent, or a *log suffix* message is sent. A log suffix is a suffix of the RDO's method log, specifically it contains those mutating methods, including methods from other clients, that have been applied to the RDO since the last time it was imported by the client — these are the methods that when applied to the client's copy of the RDO will transform the RDO to reflect the current canonical state of the RDO at the server.

The Rover toolkit provides functions for estimating the marshalled sizes of RDOs and log suffixes. The server-side application uses these functions to choose between

sending a new RDO and a log suffix based upon which one will require the least transmission space. For example, if the client has a very old copy of the RDO (*i.e.,* a large number of mutating methods have been applied to the RDO since the client imported it), then the more space-efficient choice is to send a new copy of the RDO.

3.2.2 Using RDOs

All mobile-adaptive application code and all application-touched data are written as RDOs. All RDOs have a "home" server that maintains the primary, canonical copy. Clients import secondary copies of RDOs into their local persistent caches and export tentatively updated RDOs back to their home servers. In addition, clients may create new RDOs and export the RDOs to servers for execution or for safe-keeping. Since, RDOs may execute at either clients or servers, they can be used by clients and servers to ship data and functions from one to the other or vice versa.

At the level of RDO design, application builders have semantic knowledge that is extremely useful in attaining the goals of mobile computing. By tightly coupling data with program code, applications can manage resource utilization more carefully than is possible with a replication system that handles only generic data or files. For example, an RDO can include compression and decompression methods along with compressed data in order to obtain application-specific and situation-specific compression, reducing or optimizing both network and storage utilization.

As discussed in Section 3.1.2, the Rover toolkit provides protection against faults for client applications' methods and results, but not for client applications. However, the toolkit provides extensive support for long-running RDOs at servers in the form of stable variables and recovery procedures. Stable variables are an easy way for applications to store and use information that is expensive, in terms of time, dollars, etc., to recreate after a failure. Recovery procedures are procedures that are executed during server failure recovery to restore an RDO's state, usually from data saved using stable variables. Together, these two simple mechanisms allow long-running RDOs at servers to reliably store information that is potentially expensive to reconstruct.

Figures 3-1 and 3-2 summarize the client- and server-side import and export op-

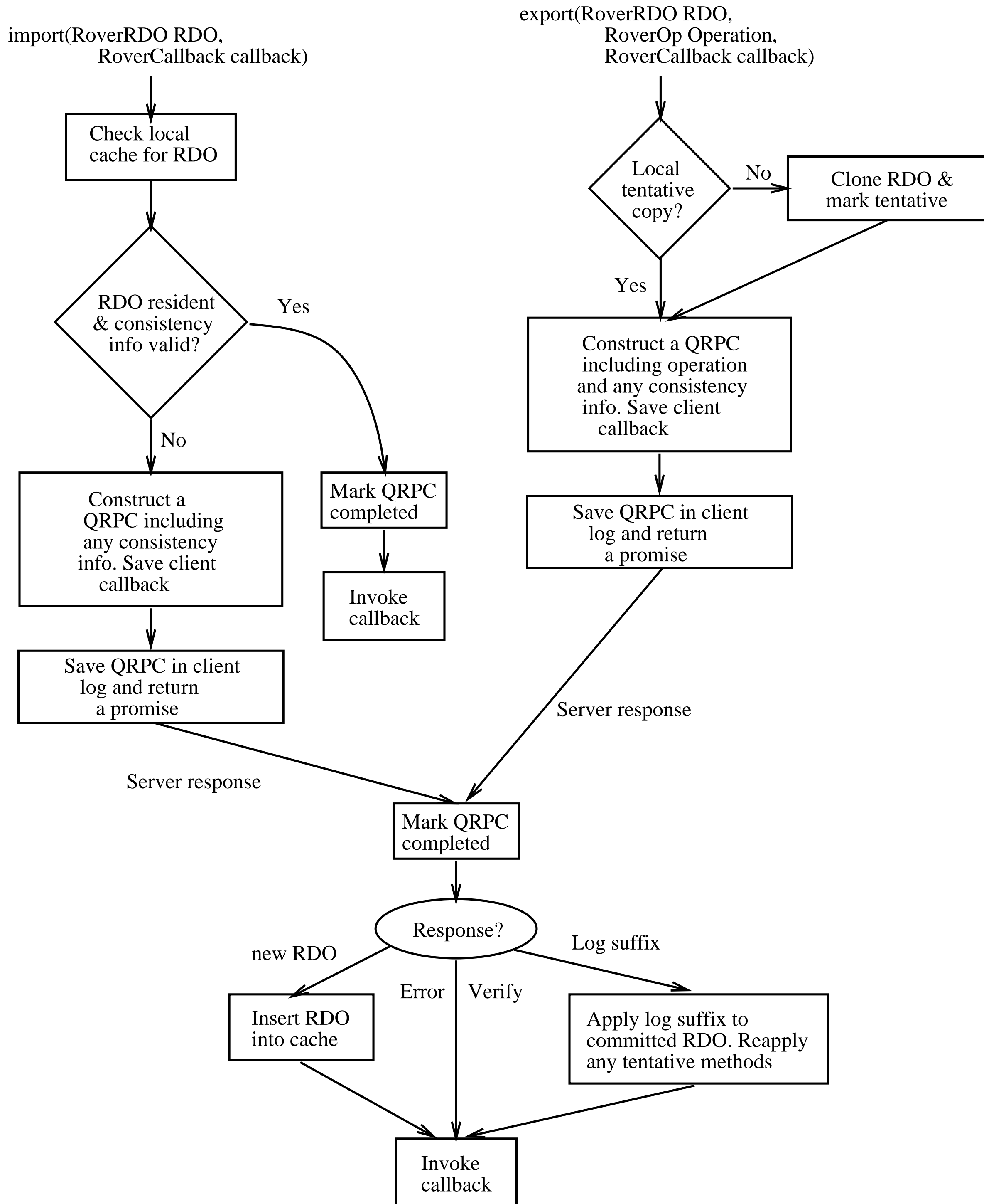


Figure 3-1: The steps in client-side import and export operations for an RDO.

erations for RDOs. The following sections discuss these operations in more detail.

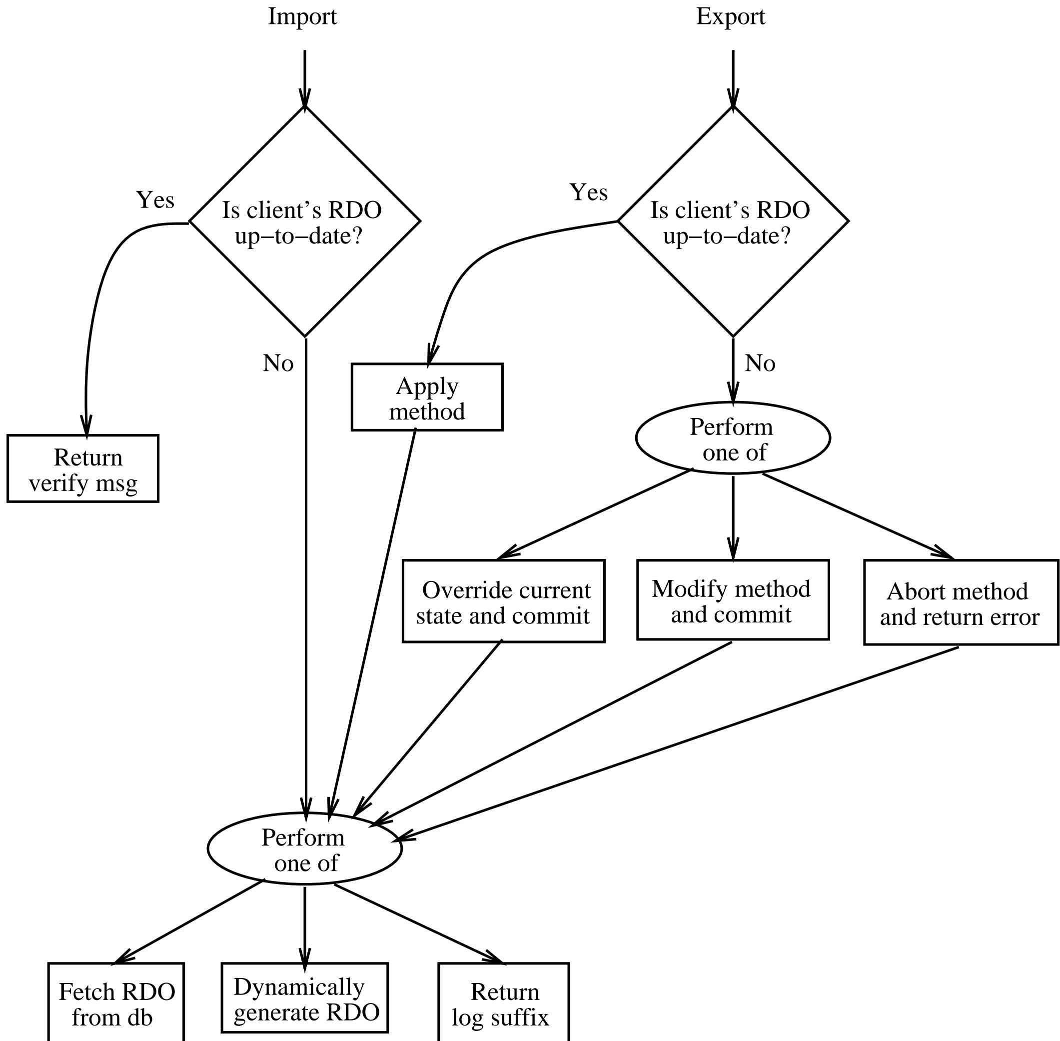


Figure 3-2: The steps in server-side import and export operations for an RDO.

Client Operation: Importing RDOs

When a client-side application issues an import request for an RDO, the toolkit first checks the local persistent RDO cache to see if the RDO is already resident in the persistent cache. Import requests, like QRPC, are non-blocking and they include a priority and an optional *callback* procedure.

If the RDO is already resident in the persistent cache and depending upon the RDO's consistency options (see Section 3.4.2), then Rover invokes the optional callback procedure to notify the application that the RDO is present.

Otherwise, if the RDO's consistency options require contacting the server or the

Server Response	Action Taken by Client Toolkit
Verify	Verification that the existing RDO is up-to-date. Rover invokes the optional callback procedure to notify the application.
New RDO	Message containing a new RDO. Rover stores the RDO in the local persistent RDO cache, marks the RDO as *committed*, and invokes the optional callback procedure to notify the application.
Log suffix	Message containing a *log suffix*. Rover retrieves the committed copy of the RDO from the persistent cache, applies the methods from the log suffix, stores the resulting new committed state of the RDO in the persistent cache, and then invokes the optional callback procedure to notify the application.
Error	Error message (*e.g.,* the RDO wasn't found or the client had invalid or insufficient access permissions). Rover invokes the optional callback procedure to notify the application of the error.

Table 3.1: Server responses to import/export operations and client toolkit actions.

RDO is not resident in the persistent cache, then the toolkit turns the request into a QRPC. Rover adds the import's priority and optional callback to the QRPC. Also, if the RDO is already resident in the persistent cache and has any application-specific consistency information, Rover includes the information in the QRPC.

Rover then immediately returns a *promise* [55] to the application, leaving the application free to continue execution. Applications may use the promise to query the status of the import operation or to block and wait for the operation to complete (blocking is an undesirable action, especially when the mobile host is disconnected).

The response from the server will be one of the four messages listed in Table 3.1: verify, new RDO, log suffix, or error.

After the response message is processed, if there are any outstanding tentative methods stored in the RDO's method log, then Rover applies the methods as described in Section 3.2.2 creating new tentatively committed copy of the RDO.

One significant issue for small mobile devices is the size of the persistent RDO

cache. The Rover toolkit does not specify a cache eviction policy, rather it provides applications with several cache management tools. Applications, and not the Rover toolkit, are responsible for ensuring that the client does not run out of cache space. Rover maintains RDO reference timestamps and application-specified priorities. Using this information and the application-awareness methods specified in Section 3.5, applications can manage their use of the persistent RDO cache. Additional cache management tools and policies are an area for future research.

Client Operation: Exporting Changes to RDOs

Once the RDO is local, the client application is free to **invoke** the methods the RDO provides. When an application invokes a mutating method on a locally cached RDO, it directly invokes the RDO's method. The implementation of the RDO's method uses toolkit provided functions to **export** the mutating method to the server and invoke the same mutating method there. Thus, the method is first performed on the local copy of the RDO and then on the canonical copy of the RDO. To perform the method locally, Rover first clones the cached copy of the RDO and then applies the mutating method to one of the copies. Next, Rover marks the modified copy as *tentatively committed* (the original, unmodified copy is still marked as *committed*). Finally, Rover saves the mutating method in the RDO's method log and commits the method by using QRPC to lazily propagate the method to the Rover server. Included with the QRPC is any application-specific consistency information for the RDO.

In the meantime, the application and other local applications may choose to use committed or tentatively committed RDOs (this choice is explained in more detail in Section 3.4). This choice allows applications to continue execution even if the mobile host is disconnected.

The results of applying the mutating method at the server are returned to the client with a completion message and a results message (more information on server-side processing can be found in the next section). The completion message indicates whether the method originally submitted by the client completed successfully. The results message is in one of two formats:

1. A new RDO. The client-side toolkit replaces the tentative and committed versions of the RDO in the persistent cache with the new RDO.

2. A log suffix. Rover retrieves the committed copy of the RDO from the persistent cache, applies the methods from the log suffix, and then stores the resulting new committed state of the RDO in the persistent cache.

   If the completion method indicates that the client's method failed, then the log suffix may contain a substitute method. The server's substitute method can be used by the client application to involve the user in conflict resolution. For example, the information could be used to provide the user with an RDO's old value, current value, and the method that failed.

As with imports, the server chooses the format that will require the least transmission space.

After the response message is processed, if there are any other outstanding tentative methods stored in the RDO's method log, then Rover clones the RDO as before and applies the tentative methods to one of the copies, creating a new tentatively committed copy of the RDO.

After all local processing is complete, Rover invokes the method's optional callback procedure.

Server Operation: Importing RDOs

When a QRPC containing an import request arrives at the server, the server routes the request to the appropriate server-side application. The application examines the consistency information included with the import request, if any was sent, and takes one of the following actions:

1. The server returns a message verifying that the client's copy of an RDO is up-to-date.

2. The server fetches the requested RDO from a object repository or database.

3. The server dynamically generates an RDO.

4. The server returns a log suffix.

For RDOs, the server sets the RDO's consistency options (see Section 3.4.2) and returns it to the client.

Server Operation: Exporting Changes to RDOs

For export requests, the server routes the request to the appropriate server-side application. The application invokes the requested method on the canonical copy. Typically, before modifying the RDO, the method first checks whether the RDO has changed since it was imported by the client or the client last received an update for the RDO.

Rover does not try to detect conflicts directly. However, to aid applications in detecting conflicts, the client-side toolkit maintains application-provided consistency information for each RDO and automatically sends the information with all import and export methods. The consistency information allows server-side applications to easily detect when the client has an old version of an RDO. Complete details of application-specific conflict detection and resolution are provided in Section 3.4.2.

If the server-side application determines that there is no conflict, then the application invokes the mutating method to modify the canonical copy and appends the results to the RDO's method log at the sever.

As outlined in the above, the results of the mutating method are a completion message, indicating whether the method completed successfully or encountered an unresolvable conflict, and a results message containing either a new RDO or a log suffix. The server returns the completion and results messages to the client.

3.2.3 The Reliable Execution of RDOs at Servers

To provide reliable execution of long-running RDOs at servers, the Rover toolkit provides special application support. Support for reliable server applications reduces the amount of work that is redone after a failure. The support is based on using programming language support to record intermediate values in the stable server log and providing failure recovery procedures to retrieve those intermediate values. The

Rover toolkit's support consists of two fault-tolerant features: stable variables and per-application and per-RDO failure recovery procedures.

Stable variables provide a simple mechanism for long-running applications or RDOs at servers to reliably store information that is potentially expensive to reconstruct. Stable variables are based upon transparent intermediate value logging. Programmers can declare stable variables anywhere within an application or RDO by simply notifying the toolkit that the variable is a stable variable. This notification can be done either at the time the variable is declared or at a later time. Stable variables are identical to ordinary global variables. The difference is that, for stable variables, the Rover toolkit records in the stable server log the stable variable's existence and initial value, if supplied. The toolkit also traces writes to the variables and then records the changes in the stable server log. The recorded values are used during failure recovery as a form of REDO log [28].

An alternative approach to providing stable variables would be to provide support for periodically checkpoint the application's or RDO's state. Providing control over the frequency of checkpointing would limit the amount of work lost after a failure, but would not provide application developers with fine grain control over which application or RDO data is stably logged and when that data is stably logged. Stable variables provide application developers with fine-grain control.

An application can specify a per-application failure recovery procedure. Likewise, an RDO can specify a per-RDO failure recovery procedure. The toolkit saves the procedures in the stable server log. During server failure recovery, the toolkit first invokes the application-specific failure recovery procedure to perform any actions that are necessary to restore the application's execution environment. Then, the toolkit invokes the RDO-specific failure recovery procedure for each QRPC that failed to complete execution and needs to be executed. The per-RDO failure recovery procedure uses any defined stable variables to restore the RDO's state and then resumes the execution of the interrupted QRPC.

Each application or RDO is responsible for using these techniques to determine the state of an RDO at the time of a failure. Rover provides support for using stable

variables to record incremental changes that are made to or by an RDO. Some applications and RDOs may choose to use their own mechanisms (*e.g.,* recording changes in a private stable log). In keeping with the Rover design philosophy, the choice is left to the application developer.

3.2.4 Using RDOs for Computation Relocation

Rover gives applications control over the location where RDOs execute. In a mobile environment, the network often separates an application from the data upon which it is dependent. By moving RDOs across the network, applications can move data and/or computation from client to server and vice-versa. Computation relocation is useful when a large body of data can be distilled down to a small amount of data or code that actually transits the network or when remote functionality is needed during periods of disconnection.

For example, migrating a GUI to the client serves both these purposes. The size of the code to implement a GUI is usually much smaller than the traffic generated by the user keyboard and mouse events the code receives and the graphical display updates the code generates. Experimental results in Section 6.6 demonstrate this difference.

At the same time, the GUI together with the application's RDOs can locally process user actions, avoiding additional network traffic and enabling disconnected operation.

Clients also can use RDOs to export computation to servers. Such RDOs are particularly useful for two operations: performing filtering actions against a dynamic data stream and performing complex actions against a large amount of data. With RDOs, the desired processing can be performed at the server, with only the processed results returned to the client.

3.2.5 The Safe Execution of RDOs

The Rover toolkit supports the migration of arbitrary code: code that was generated by a client can execute at a server, while code that was generated by a server or other

clients can execute at clients. To protect clients and servers from faulty or malicious code, Rover provides support for the safe execution of RDOs.

There are several primary issues regarding safe execution:

1. Authentication. Cryptographic authentication of clients, servers, and code. Authentication in conjunction with access control provides a means for controlling what data an RDO can access and what operations it may perform on that data.

2. Access control. Allows application developers to place limitations on the methods exported by an RDO. These limitations can be used to restrict changes to the RDO to authorized, authenticated users. Access control can also be used to restrict access to environmental resources: persistent storage, network interfaces, etc.

3. Denial of services. Faulty or malicious code may enter infinite loops, reference non-existent memory locations, allocate excessive amounts of memory or other resources, or perform other actions that deny other clients or RDOs of access to services.

Current Rover architectural support for the safe execution of RDOs is limited: RDOs sent from clients to servers include cryptographically-protected user identification information; RDOs are executed in a restricted interpreter (one that has been stripped of hazardous functions, as in [9]); at clients, separate address spaces are used for each application's RDOs; and, at servers, RDOs from clients are restricted to only invoking those functions exported by server-side applications.

These safety measures are appropriate for the sharing of RDOs between mobile hosts and servers in the framework of specific applications and environments. However, there are several safety issues relating to the general use of mobile code that are not addressed by the current architecture. These issues represent an area of active research [83] beyond the scope of this thesis.

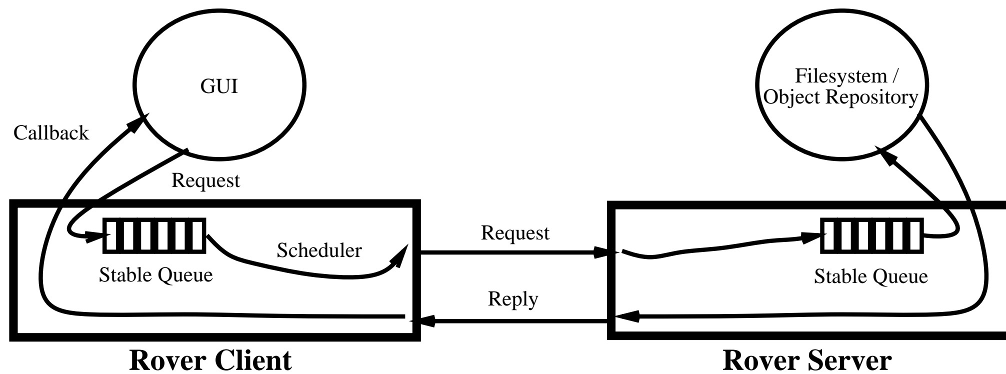


Figure 3-3: The components of Queued Remote Procedure Call.

3.3 Queued Remote Procedure Call

Queued Remote Procedure Call (QRPC) is the transport mechanism for the Rover toolkit. This section details the components of QRPC, outlines the lifetime of and uses for a QRPC, and discusses the issues relating to communication scheduling and failure recovery.

3.3.1 QRPC

As shown in Figure 3-3, QRPC consists of: Remote Procedure Call (RPC) [8], split-phase operation, a queue located on stable storage, and a network scheduler. Split-phase operation means two things: QRPC is asynchronous and it can use separate network connections or links for requests and responses. Network scheduling offers several benefits, including reduced communication latency and more efficient use of available bandwidth, as discussed in Section 3.3.4.

3.3.2 Using QRPC

QRPCs are issued when the toolkit converts an import or export request into a QRPC or a client application directly issues a QRPC. QRPC is asynchronous transport which means that control is returned to the application while the QRPC is being transmitted or is waiting for transmission. Applications may choose to include an optional callback procedure when invoking a QRPC, they may choose to block and wait, or they may

periodically poll the QRPC's status. The optional callback is invoked when the results of the QRPC are received by Rover or if the QRPC encounters an unrecoverable error. Also associated with the client request is a priority for the request, which is added to the QRPC. The priority is used by the network scheduler, discussed in Section 3.3.4.

The destination host of a QRPC is specified using a *session*. A session is a mapping between a *service* and a set of servers. Applications offer clients services (*e.g.,* the server-side portion of the E-mail browser offers E-mail service, the server-side web browser proxy offers HTTP service, etc.). Sessions are a precursor to supporting replicated servers, however, Rover does not provide any additional support for applications (*e.g.,* session guarantees across different servers [76]).

When a session contains only a single session to server mapping, Rover provides ordering guarantees for clients and sessions: the QRPCs sent by a client using a particular session will be processed in the same order that they were sent. This guarantee is necessary because some of the communication networks used by Rover do not guarantee message ordering.

Client Operation

The first step taken by the client-side toolkit when it receives a QRPC from an application is to store a *new QRPC* record containing the QRPC and its optional callback in a local log located on a stable storage device; this step makes the QRPC durable. If there is insufficient space on the stable storage device, an error is returned to the application; when such an error occurs, the application may be able to use some of the techniques described in Section 3.5 to release some of the log space.

If there are no problems, then in the next step, the client-side toolkit inserts the QRPC into the appropriate priority queue for transmission based upon the QRPC's client-specified priority. Finally, the toolkit returns a promise to the application. Since each QRPC must be logged before it is sent, the logging action is on the critical path of QRPC execution.

When the mobile host is connected, the Rover network scheduler drains the priority queues in the background, forwarding any queued QRPCs to the server. For each

QRPC that the the network scheduler is successful in sending, it appends a *QRPC sent* record to the stable log.

When the network scheduler fails to send a QRPC due to a delivery failure (*e.g.,* as a result of a communications problem, link failure, or server failure), it first reevaluates the condition of available communications links. If no links are available, the scheduler leaves the QRPC in the queue. Otherwise, the network scheduler marks the QRPC as a retry and immediately attempts to resend the QRPC. If this second attempt fails, the server is marked as down. The network scheduler will then periodically reevaluate the server's condition and attempt to resend the QRPC.

If the client-side toolkit does not receive a server response within a preset amount of time, the toolkit will poll the server for the QRPC's status. If no response is received, the toolkit retries the QRPC.

As the client-side toolkit receives the results of each QRPC, the toolkit inserts a *QRPC completed* record into the log indicating that the QRPC has completed. The toolkit then invokes the QRPC's callback, if one was specified.

Server Operation

When the server receives a QRPC from a client, it immediately appends a *QRPC received* record containing the QRPC to its stable log. This logging action is important because the QRPC may not be executed immediately (*e.g.,* the messages containing multiple QRPCs may have been arbitrarily reordered by the communications channel). If the server receives a QRPC out of order, it saves the QRPC in its stable server log and places it in an incoming QRPC queue. After processing each QRPC, the server checks the queue to see if any of the messages in it are now acceptable (*i.e.,* all preceding messages from the client host have already been received and processed). The message reception records in the stable log provide a total ordering over messages received from multiple clients.

Before starting execution of the QRPC, the server also appends a *QRPC start-of-execution* record for the QRPC to its stable log; thus, these two logging actions are on the critical path of QRPC execution. The start record is not mandatory, however,

it greatly simplifies the failure recovery processes described in the next section by allowing Rover to determine whether the QRPC started execution before the failure.

After the server has finished processing the QRPC and delivered the results back to the client, it appends a *QRPC completed* record for the QRPC to its stable log.

By default, results are stored in volatile memory. However, application developers have the option of specifying that the server should append a record containing the application's outgoing response to the stable server log. Response logging is very useful when the generated response is expensive to recreate after a failure.

The results of expensive methods (in terms of CPU usage or some other metric) or results that are not reproducible (*e.g.,* results that are dependent upon a dynamic data stream) are good candidates for response logging, as long as the size of the results is sufficiently small.

Very large or easily regenerated responses, however, are not good choices for response logging, since response logging is a synchronous action on the critical path for processing a client request.

3.3.3 QRPC Failure Recovery

This section discusses the mechanisms provided by the Rover toolkit to handle the transient software and hardware failures at clients and servers.

Client Failure Recovery

Failure recovery from client software and hardware failures is handled by the client-side Rover toolkit. This task is simplified somewhat by Rover's use of a stable log. During failure recovery, the new, sent, and completed records are used to determine the status of and action to be taken for each QRPC in the log:

1. QRPC never sent or incomplete send (new record only). The QRPC is marked as a retry and inserted into the appropriate priority queue. The retry tag is added to the QRPC because the server may have received the request header, but not the entire message.

2. QRPC sent, but incomplete (new and sent records). The toolkit probes the QRPC's recipient for a status update. If no update is received for the QRPC, the toolkit marks the QRPC as a retry and inserts it into the appropriate priority queue.

3. QRPC sent and completed (new, sent, and completed records). The QRPC completed successfully, so no action needs to be taken.

The stable client log and unique identifiers are used to guarantee at-least-once delivery of QRPCs to servers in the presence of transient client software or hardware failures. The retry tags and unique identifiers are used by servers to detect duplicate QRPCs.

One issue that remains an open question is how to handle error responses from resent QRPCs for client-side applications that no longer are running. The current design ignores such responses.

Server Failure Recovery

The server stable log is not necessary to meet the delivery guarantee specified in Section 3.1.3. However, a client stable log alone is insufficient to guarantee the efficient handling of server-side failures. For example, if the QRPCs received at servers (and the responses sent by those servers) are not stored in a stable log, then the infrequent occurrence of a server software or hardware failure requires a retransmission from the client's stable log. Such retransmissions might be significantly delayed if the client computer is not connected (*e.g.,* a client computer that is disconnected during a long plane flight). The retransmissions might also incur a significant time and dollar cost depending upon the connectivity options available to the client.

In addition to the retransmission delay, if the RDO running at the server is long-running, there might be a significant amount of lost work after a server failure. If the RDO was using real-time data, it might not be possible to completely recover the lost information.

During recovery from a server failure, the server's stable log is used as a REDO

log. The received, start, and completed records are used to determine the status of and action to be taken for each QRPC in the log:

1. Never executed (received, but no start or completed records). Rover treats the QRPC as a newly received QRPC and queues it for execution after all preceding QRPCs have been received and processed.

2. Executed and incomplete (received and start records only). The server takes one of the following actions:

   (a) If the stable server log contains a record of the QRPC's output response, the server repeatedly attempts to retransmit it to the client.

   (b) If the stable server log contains an application-provided failure recovery procedure (see Section 3.2.3), the server invokes the failure recovery procedure.

   (c) Otherwise, the server simply reexecutes the QRPC using the record of the original QRPC stored in the log.

3. Executed and completed (received, start, and completed records). No action needs to be taken.

There are two situations during server failure recovery of interrupted requests — either an import request or an exported method is being repeated. Import requests are effectively idempotent; so there are no problems with repeating them. Exported methods, however, usually have side effects and so must be handled differently.

If a failure recovery procedure is present, the procedure can use Rover-supplied information to determine whether the exported method started execution before the failure. If the method did not start execution, it is reexecuted. Otherwise, the procedure uses the RDO's method log to determine the QRPC's status and whether the exported method completed (in which case, its results are in the RDO's method log and can be sent to the client) or the method did not complete (in which case the procedure resumes the execution).

If no failure recovery procedure is present, then the RDO's mutating methods can use Rover-supplied information to determine whether the a method started execution before the failure.

To reduce the logging overhead, the server can operate without logging QRPC start records. Without QRPC start records, the failure recovery process cannot differentiate between requests that were never executed and requests that were interrupted (executed and incomplete). As such, every request without a completed record will be processed as an interrupted (executed and incomplete) request. The disadvantage to this approach is that the RDO's recovery procedure and mutating methods have no Rover-supplied information about whether the QRPC started execution or not. Instead, they must both be written so that they check the method log to determine whether a method is being reexecuted. However, this approach may be costly, since it usually involves checking persistent storage, and it complicates application development.

3.3.4 Client Communication Scheduling

As mentioned previously, each QRPC has an associated priority. There are multiple priority queues for sending QRPCs and client applications (and users through those applications) can specify the priority of a QRPC.

The Rover network scheduler may deliver QRPCs out of order (*i.e.,* non-FIFO), depending upon any associated, application-specified priorities and the dollar costs for using the available networks. It also may reorder logged requests based on consistency requirements and application-specified priorities. Reordering is important to usability in an environment with intermittent connectivity, as it allows the user through applications to identify the important methods. For example, a user may choose to send urgent updates as soon as possible while delaying other sends until inexpensive communication is available. Alternatively, the user may choose to cancel or abort the request using the mobile-adaptive application support discussed in Section 3.5.

One of the advantages of queued communication is that there may be queues of requests that are waiting to be sent as a result of intermittent connectivity or

disconnected operation. The network scheduler can take advantage of such a backlog in two ways: by batching or grouping together requests that are destined for the same server, and by transmitting duplicate requests once.

Batching of Requests for the Same Server

The original network scheduler sent a request as soon as it was received from a client application — the goal was to minimize delay for every request. However, it quickly became apparent that for high-latency networks, sending a single request per-connection resulted in very high per-message overhead. For example, experimental results show that a request sent over a cellular network completes in an average of 640 milliseconds.

As a result of these observations, the network scheduler uses the following heuristic to batch requests that are destined for the same server. Before the network scheduler sends a request to a server it performs two checks:

1. The scheduler checks the priority queues for additional requests that are destined for the same server.

2. The scheduler uses the access manager to check all the client applications (including the one that sent the original request) to see if any are in the process of sending a request.

Rover automatically merges any requests that are found into a single batch of requests. After each request is found, the scheduler repeats the checks. To prevent a request from being excessively delayed by batching, the scheduler places a bound on the amount of the time spent checking for additional requests.

When the time bound is reached or no additional requests are found, the scheduler batches the requests and sends them on the same connection; the results are also received on the same connection. Thus, an application that issues several requests in a series will have the requests automatically batched and sent to the server using a single connection.

This batching heuristic imposes a small, bounded delay on requests (larger for early requests in the batch, smaller for later requests in the batch): the time for the access manager to check each client application for pending requests and to receive the requests. This delay is a small penalty to pay relative to the three significant benefits that batching offers: reduced average request latency, better compression ratios using automatic Rover-supplied compression, and better utilization of bulk transport networks. For example, batching imposes an 1,800 millisecond delay on the first request of a batch. However, subsequent requests complete on average once every 17 milliseconds.

Batching is important in situations where the round-trip latency for establishing communication connections is high. By sending multiple requests in a single connection, Rover significantly reduces the average connection delay seen by a request when using high-latency networks. For high-latency, bulk transport networks (*e.g.,* SMTP), batching allows the high per-message overhead usually associated with such networks to be amortized across multiple requests.

The network scheduler also provides automatic compression of requests; applying compression across the multiple requests within a batch usually yields significantly better compression ratios. Many network interconnection devices provide data compression. However, several studies have shown that processor-based software compression yields more efficient compression than network infrastructure-based compression [57, 63]. The most likely reason for the difference is the availability on hosts of more memory for compression dictionaries and faster host processors.

One of the alternatives to the network scheduler's batching heuristic is to rely upon applications to specify the set of requests that should be batched together (*e.g.,* by using startBatch and endBatch commands).

There are two advantages to the batching heuristic over application-specified batching. The first advantage is that the heuristic naturally adapts the size of batches to the available bandwidth and latency. When the network is low-latency and high-bandwidth, transmission queues will be short and batches will be small. However, when the network is latency higher and bandwidth is lower, transmission queues will

be longer and batches will be larger. This effect is confirmed at the network transport level in Section 6.2.3.

The second advantage is that the heuristic is transparent to applications. To adapt to rapidly changing network conditions, application-specified batching requires that applications be constantly aware of the conditions. However, one of the goals of the Rover toolkit is to decouple applications from environmental volatility.

The disadvantage to the heuristic over application-specified batching is that the heuristic delays requests while it performs its checks — checks that represent wasted time when there are no pending messages. Thus, for future work, it would be interesting to investigate having the network scheduler use application-specified batching commands as informational suggestions or overrides for the batching heuristic.

Merging of Duplicate Requests

In similar fashion to batching, when a client issues an import request for an RDO, Rover checks to see if there are any duplicate requests for the RDO (*i.e.,* duplicate QRPCs). Rover automatically merges the duplicate requests into a single request, at a priority equal to the highest priority of any of the individual requests and with multiple callbacks, one for each individual request. By merging requests, Rover reduces latency and bandwidth.

Split-Phase Operation of QRPC

QRPC supports split-phase operation; thus, if a mobile host is disconnected between sending the request and receiving the reply, a Rover server will periodically attempt to contact the mobile host and deliver the reply. To aid the server in contacting the client host, the QRPC can include several possible contact addresses for the mobile host.

One advantage of the split-phase communication model is that it enables Rover to use different communication channels for the request and the response and to close channels during the intervening period. Closing the channel while waiting is particularly useful when the waiting period is long and the client must pay for connection

time in dollars or in reduced battery power.

Another advantage of QRPC's split-phase communication model is its support for asymmetric communication. This support is an ideal match for the typical application's traffic pattern, where the majority of client-server data traffic is from servers to clients. Several wireless technologies offer asymmetric communication options, such as receive-only pagers. By splitting the request and response pair, communication can be directed over the most efficient, available channel.

The combination of the split-phase and stable nature of QRPCs allows a mobile host to be completely powered-down while waiting for pending request. When the mobile host resumes normal operation, the results of the QRPC will be relayed reliably from the server. Thus, long-lived computation can occur at the server while the mobile host conserves power.

3.4 RDO Replication and Consistency

An essential component to accomplishing useful work while disconnected is having the necessary information locally available [46]. RDO replica caching is the chief technique available in Rover to achieve high availability, concurrency, and reliability. However, allowing local or remote modifications to shared application data while disconnected introduces the problem of data consistency. The Rover toolkit addresses this problem with a flexible approach: application-specific consistency control. This section discusses strategies for replicating RDOs and for reducing consistency-related costs.

3.4.1 Replication

Rover relies on the replication of RDOs to reduce client applications' dependence upon network resources (*e.g.,* so clients can continue working while disconnected). One of the issues associated with replication is ensuring that cached client copies are consistent with the server's canonical copy. Rover supports several options for validating copies of RDOs. The validation option is stored as a cache tag with the

Cache Tag	**Function**
Uncacheable	An uncacheable RDO will not be stored in the client's persistent cache.
Immutable	An immutable RDO cannot be modified by client applications and is never revalidated.
Verify before use	A verify before use RDO is always validated before it is returned to a client application; this action may result in significant delays if the client is disconnected.
Best effort verify before use	A best effort verify before use RDO is validated before it is returned to the client, if network connectivity to the server is available. Otherwise, the cached RDO is returned immediately to the client and a QRPC request to validate the RDO is sent to the server.
Server callback	A server callback RDO is not validated by the client, instead the server notifies the client of any changes to the RDO.
Lease	A lease RDO is not revalidated until the lease from the server expires or is revoked.
Application-specific	The application registers a function that Rover can invoke. The function returns the appropriate action that the toolkit should take in handling the cache element.

Table 3.2: Cache tags options for RDO revalidation.

RDO in the client's persistent cache. RDOs are revalidated when a client application imports an RDO that is already in the client's persistent cache. Rover also performs periodic background revalidations.

Server applications can mark RDOs with one of seven tags (see Table 3.2). For each RDO in the persistent cache, the toolkit performs different actions depending upon the specified tag

As mentioned previously, for small mobile devices the size of the persistent RDO cache must be considered. The Rover toolkit provides applications with several cache management tools and information about cache contents, but does not specify a cache eviction policy.

RDO replication is accomplished during periods of network connectivity by filling the mobile host's persistent cache with useful RDOs. Applications should decide which RDOs to prefetch. The usability of applications will be critically dependent upon simple user interface metaphors for indicating collections of RDOs to be prefetched. Requiring users to directly list the names of RDOs that they wish to prefetch is inherently confusing and error-prone. Instead, Rover applications can provide prioritized prefetch lists based upon high-level user actions. For example, the Rover E-mail browser, *Rover Exmh*, automatically generates prefetch operations for the user's inbox folder, recently received messages, as well as folders the user visits or selects.

While replication can bring great benefits, application developers must be careful to avoid unnecessary communication, increased latencies, and dead-lock. Applications should not replicate any more data than absolutely necessary and should strive to keep update messages small.

3.4.2 Consistency

When clients are allowed to perform concurrent updates on shared RDOs, most applications require consistency control to resolve these uncoordinated updates.

In Rover, the server-side application is responsible for maintaining consistent views of an application's data. Update conflicts are detected and resolved, when possible, by the server-side application, and the results of reconciliation are always treated by the

client-side of applications as overriding the tentative state stored at the client. Thus, the client-side applications only needs to submit tentative methods to the server to reconcile the system state and to assure that any updates are durable.

As discussed earlier, the Rover toolkit provides extensive support for client and server application handling of mutating RDO method invocations. Client-side application support consists of automatic mutating method logging, rollback, and replay. Server-side application support consists of RDO method log manipulation functions.

The Rover toolkit only addresses consistency schemes and not policies. Thus, applications are fully responsible for implementing appropriate consistency policies using either the consistency schemes provided by the toolkit or their own schemes (*e.g.,* application-level locking or application-specific algorithms). This decision is based on the idea that no single consistency scheme or policy is appropriate for all applications.

Optimistic Concurrency Control

There are only a limited number of schemes that lend themselves naturally to mobile environments. The most appropriate scheme is primary-copy, tentative-update optimistic concurrency control; Rover provides substantial support for this scheme. However, application developers are not precluded from using other schemes.

Optimistic concurrency control schemes will be widely used because they allow updates by any host on any local data regardless of the availability of network connectivity. All Rover applications built to date use optimistic concurrency control. However, many applications will continue to use a variety of schemes, including *ad hoc* approaches such as hand editing or requiring all data replicas to converge to the same values. Certain applications will be structured as a collection of independent atomic actions [25], where the importing action uses application-level locks, version vectors, or dependency-set checks to implement fully-serializable transactions within Rover method calls. Of course, pessimistic concurrency control may cause long blocking periods in the mobile environment.

Together mutating method logging and optimistic concurrency control provide

applications with powerful tools for conflict avoidance. By logging method invocations at clients, rather than only new data values, Rover increases application flexibility in resolving conflicts.

For example, a financial account RDO with debit, credit, and balance methods provides significantly more semantic information to the application than a simple account file containing only the balance. Debit and credit methods from multiple clients can be arbitrarily interleaved as long as the balance never becomes negative. In contrast, consistently updating a balance value by overwriting the old value requires the use of an exclusive lock on the global balance.

Application-Specific Consistency Information

The downside to optimistic concurrency control is the potential for update-update conflicts. Conflict detection, avoidance, and resolution all may depend not only on the application, but on the data or even the method involved. Since Rover can employ type-specific concurrency control [86], many potential conflicts may be avoided.

The Rover toolkit provides clients with infrastructure to maintain the RDO consistency information necessary to detect *potential update-update conflicts* and leaves it up to application developers to provide the policies necessary to verify conflicts and to resolve *true update-update conflicts*.

When a server-side application returns an RDO or log suffix, it can include consistency information (*e.g.,* version vectors). The client-side toolkit maintains the information with the RDO and sends the information with any import or export requests. Thus, the server-side application can easily compare the consistency information provided by the client with the RDO's current consistency information. If the two do not match, the client has a stale replica of the RDO. For an import request, the server-side application returns a current replica. Otherwise, for an export request, a potential update-update conflict exists: the client invoked the mutating method against a stale replica.

Application-Specific Conflict Detection and Reconciliation

When a potential update-update conflict is identified, it is up to the application to determine whether a true update-update conflict exists. The definition of conflicting modifications is strongly application- and data-specific. Since the method being performed is available to the server-side application, it can determine whether an update-update conflict really exists.

The first step for the server-side application is to determine whether earlier changes to the RDO actually conflict with the new method. If there is a true update-update conflict, the next step is to *reconcile* the server's copy of the RDO with the method sent by the client.

The conflict reconciliation process is dependent upon application-specified policy, discussed in more detail in the next session. Based upon that policy, the server-side application takes one of the following three actions:

1. The application commits the method by overriding the current state of the RDO. The application then returns a log suffix, which includes the newly committed method.

2. The application modifies the method (resolving the conflict) and commits the new method. The application then returns a log suffix, which includes the newly committed modified method.

3. The application aborts the method by returning a method aborted completion message and a log suffix.

Since the submitted method is tentative and may have been originally performed at the client on tentative data, the result of performing the method at the server may not be exactly what the client expected.

Application Policy

As discussed earlier, Rover leaves consistency policy decisions up to application developers. Application policy determines whether two update conflict with each other

and how the application handles conflicts.

As an example of one application's choices, the Rover distributed calendar tool exploits semantic knowledge about calendars, appointments, and notices to determine whether a change violates consistency. For example, concurrently deleting two different appointments in the same calendar does not result in a conflict. Each client is informed of the other concurrent delete, so that its copy of the calendar will reflect the second delete. For a conflict that cannot be reconciled, the server returns an error that is reflected to the user so that he or she can resolve the conflict.

Another application policy issue that application developers must address is the potential for cascading aborts. Cascading aborts can occur when the client sends a batch of mutating methods to the server and one fails. If one of the methods encounters a conflict and is aborted, subsequent methods also may encounter the same conflict and also abort. Depending upon the number of aborted methods, the client application may be forced to present the user with a large number of conflicts to resolve. Thus, the potential for cascading aborts should be minimized as much as possible by structuring methods so that they are minimally interdependent upon one another or are commutative.

The interval between the time an RDO is imported by an application and the time an method is exported back to the server represents the time window during which conflicting updates may occur. Applications can reduce this or eliminate this window by using a variety of techniques: periodic revalidation of RDOs using polling, application-specific locks, or server callbacks.

3.5 Mobile-Adaptive Application Support

The Rover toolkit provides client-side applications with access to a variety of information about the mobile environment and the internal state of the toolkit. Rover client applications can query the status and availability of the network connectivity and bandwidth, computational resources, electrical power, etc. The toolkit allows applications to poll for status changes or register callback functions.

The toolkit also provides client applications with the ability to access and modify internal toolkit state. For example, client applications can access the persistent RDO cache with the ability to view all entries and delete committed RDOs.

Client applications can view all of the entries in the QRPC priority queues and the stable client QRPC log. Applications can also modify or delete entries that they created, with the caveat that they are responsible for ensuring that their modifications do not break their own application semantics. Note that in some instances, Rover may not be able to change or modify a QRPC that has already been sent to a server.

One issue Rover addresses with an application-specific approach is method log growth during disconnected operation. The ability to convey application-level semantics directly to servers is an important functional advantage, especially in the presence of intermittent connectivity. However, it may lead to a method log that grows in size at a rate exceeding that of a simple write-ahead log. The traditional approach is log compaction [46]. Rover takes a different approach by directly involving applications in reducing the size of the log. Applications can download procedures into the access manager to manipulate their log records. For example, an application can filter out duplicate requests (*e.g.,* duplicate QRPCs to verify that an object is up-to-date can be reduced to a single QRPC). In addition, applications can apply their own notion of "overwriting" to their methods in the log with the aforementioned caveat about application semantics.

3.6 Design Summary

To summarize, the design and architecture of the Rover toolkit provides four solutions to challenges posed by the mobile environment.

Relocatable Dynamic Objects allow computation and data to be dynamically migrated closer to where it is needed. By moving computation from clients to servers, operations that require high-bandwidth, low-latency network support can execute on a well-connected server. Likewise, Rover allows applications to use computation relocation to help improve interactive performance for users by moving GUI functions

from servers to clients. This relocation allows user actions to be serviced locally and immediately, instead of requiring the actions and their results to traverse the network.

Queued Remote Procedure Call simplifies application development by relieving application developers of the need to handle transient communication, hardware, and software faults. QRPC also allows communications to be more efficiently scheduled using batching and compression.

Replication and application-specific consistency control control allow users to continue accessing and modifying locally cached data regardless of the availability of network connectivity.

Mobile-adaptive application support allows applications to query and interact with the mobile environment. Rather than hide environmental information, it fully exposes it to applications and, through the applications, it also exposes it to users.

Chapter 4

Implementation of the Rover Toolkit

4.1 Implementation Overview

This chapter provides specific details about the reference implementation of the Rover toolkit. The reference implementation provides a fixed reference point by making many of the design and architecture details concrete. Other implementations of the Rover toolkit may make different implementation choices.

The reference implementation is a prototype; as listed below, not all of the features listed in the preceding chapter are implemented. Some of the features that were not considered essential in demonstrating the usefulness of the Rover toolkit have not been implemented. The unimplemented features include: the server callback, lease, and application-specific cache tag options; encrypted messages; sharing by local applications of modified objects through the client's cache; multiple contact addresses for a client machine in a QRPC; server-side RDO size and log suffix size estimation functions; complete handling of errors.

As shown in Figure 4-1, the reference implementation of the Rover toolkit consists of four key components: the access manager, the RDO cache (client-side only), the RDO database (server-side only), the stable log, and the network scheduler; this chapter discusses each component in turn.

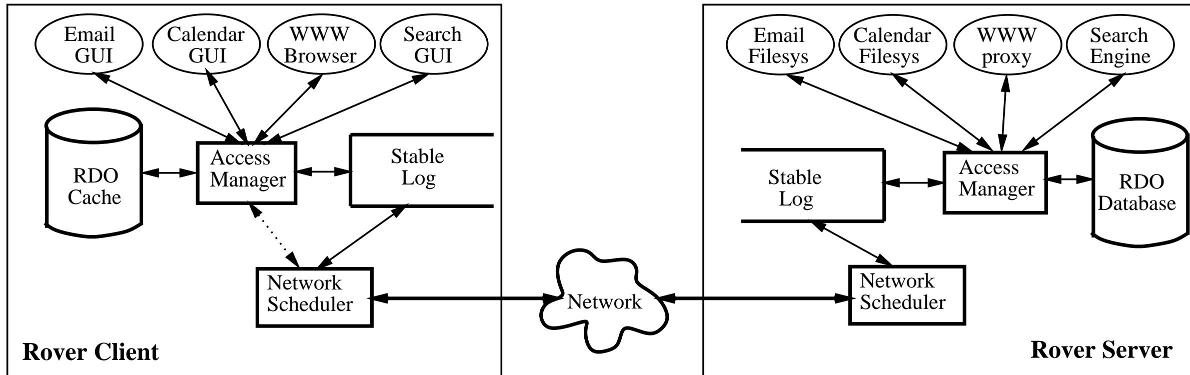


Figure 4-1: The Rover toolkit reference implementation.

Each server or client machine has a local Rover *access manager*, which is responsible for handling all interactions between client-side and server-side applications. The access manager services requests for RDOs, mediates network access, logs mutating method invocations on RDOs, and manages the persistent RDO cache and persistent RDO database. Client-side applications communicate with the client-side access manager to import RDOs from servers, to cache them locally, to invoke the methods provided by RDOs and to make changes globally visible by exporting them back to the servers. Server-side applications are invoked by the server-side access manager to handle requests from client-side applications.

The reference implementation of RDOs uses the Tcl and Tk languages [66]. Since the interface is designed to be language-independent, it will be easy to explore the use of other interpreted or byte-compiled languages (*e.g.,* Java [3]). RDOs use interpreted or byte-compiled languages because they languages greatly simplify computation migration across different architectures.

Rover is designed to be portable and relatively independent of operating systems. As such, Rover has been implemented on several platforms:

- Several types of IBM ThinkPad laptops running Linux, including ThinkPad 560 (133Mhz Pentium) laptops running Linux 2.0.30.
- Intel 200 Mhz Pentium Pro workstations running OpenBSD 2.1.
- Intel Advanced/EV (120 Mhz Pentium) workstations running Linux 1.3.74.

- DECstation 5000 workstations running Ultrix 4.3.

- SPARCstation 5 and 10 workstations running SunOS 4.1.3_U1.

The primary mode of operation is to use the laptops as clients of the workstations. However, workstations can also be used as clients of other workstations.

Rover supports a variety of network transports, including wired and wireless local-area and wide-area network technologies (for more information, see Section 6.1).

4.2 Client-side Implementation

Rover clients use a local client-server model: each client application executes in a separate address space from one another and from the local access manager, persistent RDO cache, QRPC log, and network scheduler. By using separate address spaces, the failure of an individual client has no effect on other clients or the access manager and its components.

The Rover toolkit on the client helps applications to be minimalistic in their operation. Client applications start with a minimal "kernel" of functionality and import additional functionality on demand. This feature is particularly important for mobile hosts with limited resources: small memory or small screen versions of applications may be loaded by default. If the application finds more hardware and network resources available (*e.g.,* if the mobile host is docked), additional RDOs may be loaded to handle these cases [42].

The following sections detail the implementation of the client-side Rover toolkit, including the access manager, persistent RDO cache, QRPC log, and network scheduler.

4.2.1 Access Manager

Each Rover client application is linked with the Rover toolkit client library and usually executes as a child processes of the access manager. Client applications use the library to communicate with the access manger through POSIX pipes or sockets so they can

import RDOs or *export* logs of methods that mutate RDOs. The applications can also run as independent processes on the local machine or a remote machine and communicate with the access manager using TCP sockets. Remote client support is useful when dealing with severely resource limited mobile hosts (*e.g.,* PDAs) that have high-bandwidth, low-latency connectivity to a local mobile host.

The Rover library linked with the client application manages communication to the access manager using a single-threaded, event driven finite state machine. If the client application is a Tk script, the state machine also interacts with the Tk event loop.

The access manager is also single-threaded, and it uses an event driven finite state machine and non-blocking operations to provide efficient support for servicing multiple clients. Within the access manager, RDOs are imported into the *persistent RDO cache*, while QRPCs are exported to the *QRPC log*. The access manager routes invocations and responses between applications, the cache, and the method log. The QRPC log is drained by the *network scheduler*, which mediates between the various communication protocols and network interfaces.

4.2.2 Persistent RDO Cache

The *persistent RDO cache* provides stable storage for local copies of imported RDOs. The RDO cache model is a three-level cache, consisting of local in-memory caches that are private to each application, and global in-memory and stable storage-based caches managed by the access manager.

To improve the efficiency of the local client-server model and reduce communication between client applications and the access manager, each application has a local private cache located within the application's private address space and managed by the Rover toolkit client library. Copies of each RDO imported by the application are cached within the private cache. These copies are unavailable to other applications on the mobile host, but, if desired by the application, they will be kept consistent with the access manager's RDO cache.

The global in-memory cache is located within the access manager's address space,

while the global stable storage-based cache is stored on a local stable storage device. For both caches, the access manager maintains time of last reference and application-specified priority information. The information can be used by client-side applications when RDOs need to be evicted from the cache to provide space for new RDOs . The access manager also ensures that all changes to the in-memory cache are propagated to the stable storage-based cache. In addition, the access manager limits the size of the in-memory cache and transparently moves RDOs from the in-memory cache to the stable storage-based cache and vice versa using a Least Recently Used (LRU) algorithm.

Client-side applications do not usually interact directly with the global cache and the cache hierarchy is not visible to client applications. When a client-side application issues an *import* or *export* operation, the toolkit and access manager satisfy the locally request based upon whether the RDO is found in the RDO cache hierarchy and the replication and consistency options specified for the RDO (see Section 3.4).

The reference implementation limits client-side applications to sharing committed data through the cache. Future implementations will remove this restriction.

4.2.3 Stable QRPC Log

QRPCs are issued when the toolkit converts an import or export request into a QRPC or a client application directly issues a QRPC. The insertion of a QRPC into the stable QRPC log is a synchronous action and consists of saving the QRPC contents, the QRPC's session identifier, an optional callback, and the QRPC's priority to the stable QRPC log.

The stable client QRPC log is managed by the access manager and the network scheduler. The log is implemented as an ordinary POSIX file located on: a hard disk, a battery-backed static RAM PCMCIA card, or a flash RAM PCMCIA card. The Rover toolkit only addresses transient hardware failures, however media failures for these devices are sufficiently rare that ignoring them does not represent a cause for concern.

After the access manager appends a QRPC to the log, it performs both a file flush

and, optionally, a file synchronize operation. These actions flush both the application and operating system file buffers and force the QRPC to the stable storage device. Thus, the log append, file flush, and file synchronize operations are on the critical path for request transmission.

The file synchronize operation is optional because of the cost that it imposes on QRPCs (see Section 6.2). Making the operation asynchronous only opens a small window of a few seconds of vulnerability. To increase the robustness of the log when the file synchronize operation is not immediately performed, the access manager installs signal handlers for several types of faults, including segmentation faults and bus errors. Should an unexpected signal occur, the access manager performs flush and synchronize operations on the log file before aborting its operation.

Support for intermittent network connectivity is accomplished by allowing the log to be incrementally flushed back to the server. Thus, as network connectivity comes and goes, the client will make progress towards reaching a consistent state.

As detailed in Section 3.3.2, the stable QRPC log is used during failure recovery. The new QRPC, QRPC sent, and QRPC completed records are used to determine the status of and action to be taken for each QRPC in the log. QRPCs that were never sent or incompletely sent are marked as retries and inserted into the appropriate priority queue. For QRPCs that were sent, but did not complete, the toolkit sends a status request message to the server. If no response is received for the QRPC, the toolkit marks the QRPC as a retry and inserts it into the appropriate priority queue. The toolkit takes no action for those QRPCs that successfully completed.

The reference implementation provides limited client application access to the status of outstanding QRPCs. Future implementations will provide client applications with a richer interface and allow the applications to register QRPC status callback procedures.

4.2.4 Network Scheduler

The *network scheduler* manages the available network transports and handles communication between the client-side and server-side toolkits. When communication

channels are available, the network scheduler drains the QRPC priority queues (and the stable log) and forwards QRPCs to the appropriate servers.

The reference implementation implements request priority using a simple multiple queue mechanism. Future implementations will explore the use of lottery scheduling and other priority mechanisms.

To determine the destination server for a request, the network scheduler uses the QRPC's session identifier. The client-side toolkit maintains session-to-server mappings for client applications. Sessions provide a level of indirection for referencing servers and are a precursor to supporting replicated servers. Each mapping indicates the current destination server for requests on that session. The toolkit changes the mapping if the current server is down or the toolkit is unable to send requests to the current server. The reference implementation currently does not provide any additional support for replicated servers.

Once the destination server has been identified, the *network scheduler* batches and compresses requests for that server and selects the appropriate transport protocol and medium over which to send them. Rover is capable of using a variety of network transports. Rover supports both connection-based protocols (*e.g.,* HTTP over TCP/IP networks) and connectionless protocols (*e.g.,* SMTP over IP or non-IP networks) [14, 35]. Different protocols have different strengths. For example, while SMTP has extremely high latency, it is fundamentally a queued background process; it is more appropriate than more interactive protocols for fetching extremely large documents, such as stored video, which require large amounts of time regardless of the protocol. Another advantage is that the IP networks required for HTTP or TCP are not always available, whereas SMTP often reaches even the most obscure locations.

By batching requests, the network scheduler gains transmission efficiency by leveraging the queuing of QRPCs performed by the log. The result is a potentially significant reduction in per-request transmission overhead and an increase in connection efficiency through amortization of connection setup and teardown across multiple requests and responses. This amortization is especially important when connection

setup is expensive (either in terms of added latency or dollar cost). For example, the latency for a null RPC over a 9.6 Kbit/s Cellular CSLIP link is 640 milliseconds; batching offers a substantial performance benefit: 1,800 milliseconds for the first request and 17 milliseconds for subsequent requests.

The network scheduler uses the Zlib compression library [18, 19, 20] to apply compression to the headers associated with requests and, in the absence of application-specified compression, applies compression to application data. Compression offers significant performance advantages, especially when combined with batching. Typical compression ratios for the applications studied in this thesis are 1.5 – 9.7 to one. The combination of batching and compression yields, on average, a two- to four-fold reduction in execution times. These applications were communication-bound and not computation-bound, however, these are the attributes of most of the applications that are used in the mobile environment.

The reference implementation does not provide support for encrypted communications, however, future implementations will provide support for a variety of encryption technologies.

4.3 Server-side Implementation

The Rover server-side toolkit is a privileged application that authenticates requests from client applications, mediates access to RDOs, and provides a reliable Tcl execution environment for RDOs from client applications.

For simplicity, the current server implementation is single-threaded. The server can simultaneously receive multiple requests, but only one request is processed at a time.

As described in the preceding chapter (Chapter 3), the server-side toolkit provides server-side applications with: functions for accessing and modifying an RDO database or repository, stable variable and response logging functions, RDO method log maintenance and manipulation functions, and support for maintaining application-provided consistency information and detecting potential update-update conflicts.

The following sections detail the implementation of the server-side Rover toolkit, including the access manager, network scheduler, stable QRPC log, and persistent RDO repositories.

4.3.1 Access Manager

Each Rover server application is linked into the server using a modular, well-defined *plug-in* interface. The Rover toolkit and the server applications all execute in a single address space.

Within the access manager, incoming QRPCs are placed into receive queues for processing. The queues are used to hide any network reordering of messages. The access manager routes incoming requests and responses between client machines and server applications.

There are two implementations of Rover servers. One is compatible with the Common Gateway Interface (CGI) [60] of standard, unmodified HTTP-compliant servers (*e.g.,* Apache, CERN, NCSA, and other httpd servers). A new copy of the CGI-based server is forked and executed for each connection from a client; these fork and exec operations make it expensive to maintain shared persistent state.

The other implementation is a standalone TCP/IP server that provides a very restricted subset of HTTP/1.0. A single standalone server provides service to multiple clients. The standalone server yields significant performance advantages over the CGI version, as it avoids the fork and exec overheads incurred on each invocation of the CGI version. In addition, because a new copy of the CGI server is started to satisfy each incoming request, any state that is intended to be persistent across connections must be saved using the file system and re-read for each connection. For these reasons, the CGI-based server is only intended to be a technology demonstrator that shows interoperability between Rover and ordinary httpd servers; the rest of this chapter only discusses the standalone server.

Reliable Server Execution

To protect against transient software and hardware server failures, the standalone Rover server operates as a pair of processes. The server is automatically started at system startup time and immediately forks a copy of itself. It then performs a wait operation on the child process.

Meanwhile, the child server starts execution by: initializing its state; performing failure recovery, if necessary; and preparing to receive and process requests. During initialization, the child server installs signal handlers for all the POSIX signals that can be caught by a process.

If the child server encounters an unrecoverable error or a signal that cannot be handled, it terminates operation. The parent process detects the end of the child server and starts a new child server. Because the parent server's operation is extremely simple (wait and restart a process), it should not be vulnerable to many software faults. The parent server will automatically recover from transient server hardware failures since it is automatically started at system startup time.

To improve the efficiency of the server's recovery from transient failures, the Rover toolkit includes several server-side features for reliable operation as described in Section 3.2.2 and 3.3.2. The reliability features are a stable log for incoming QRPCs, stable variables for intermediate value logging, and logging of server responses to stable storage.

4.3.2 Network Scheduler

The network scheduler handles the reception of incoming requests from clients and the transmission of results back to clients. The server executes as a single process; so to avoid blocking other parts of the server, the network scheduler uses an event driven finite state machine and non-blocking operations to receive requests and send results.

For efficiency reasons and because of the latency associated with opening a network connection back to a client, the toolkit usually sends the response to a client on the

same connection as the request. However, the connection may be shutdown by the user (to save money or battery power) or lost as a result of communication failures. As mentioned earlier, QRPC supports split-phase communications, which allows the server to send results back to the client using a new connection.

The client-side toolkit support for split-phase communication reuses functionality added to the client-side access manager to support the Rover HTTP proxy [41] (see Section 5.3.1). The access manager provides an httpd-like interface port that allows unmodified client WWW browsers to operate in an offline or intermittently connected manner by submitting HTTP requests directly to the access manager, instead of a remote WWW server.

To aid the server in contacting the client host, each QRPC from the client includes several contact addresses for the client access manager (*e.g.,* IP and SMTP addresses) and the client access manager's httpd port number.

Thus, when a new connection is needed, the server-side toolkit will use the contact list to attempt to open a new connection to the client-side access manager.

4.3.3 Stable QRPC Log

Crucial to providing the server with efficient reliable operation is a stable log of incoming QRPCs, stable variables, failure recovery procedures, and logging of server responses as described in Sections 3.2.3, 3.3.2, and 3.3.3. Incoming QRPCs are saved in the stable log to make failure recovery more efficient. During QRPC execution, server-side applications may use stable variables and response logging to save intermediate values and final results to the stable server log. Together, these techniques enable applications to quickly recover from transient failures, reduce the delay in recovering from failures, and reduce the amount of work that is reexecuted after a failure.

The following sections detail the implementation of the stable log and server-side support for reliable execution.

```
set counter 0
while {$counter < 10000} {
    incr counter
}
```

(a). Volatile Tcl counter

```
global counter
Rover_stable counter 0
while {$counter < 10000} {
    incr counter
}
```

(b). Stable Tcl counter

Figure 4-2: Volatile and stable Tcl counters.

Implementation of the Stable QRPC Log

Like Rover clients, the server stable QRPC log is implemented using ordinary POSIX files located on a hard disk, on a battery-backed static RAM PCMCIA card, or on a flash RAM PCMCIA card. As each record containing an incoming QRPC, stable variable, server response, etc. is appended to the log, the server-side toolkit performs both a file flush and, optionally, a file synchronize operation. Thus, log flushing is on the critical path for message reception, stable variable processing, and the processing of other reliable execution operations.

As described in Section 3.3.2, all incoming QRPCs are saved in the stable log before being executed. In addition, before the server begins executing a particular QRPC, it appends a start record for the QRPC to the stable log.

During idle periods, the server-side toolkit cleans the stable QRPC log by deleting information for QRPCs whose results that have been successfully transmitted back to client machines.

Stable Variables

Rover's server-side support for reliable execution of long-running RDOs is based upon extending the programming language used to construct RDOs, Tcl. Application developers declare stable or non-volatile variables using the `Rover_stable` *variableName [initialValue]* declaration to specify the name and, optionally, the initial value for a stable variable. The server appends the declaration in the stable server log file and sets a Tcl variable trace on the variable. The Tcl variable trace procedure registers

a procedure that will be invoked when the specified variable is modified. Rover uses this procedure to detect changes to stable variables; the changes are then appended to the stable server log.

Programmers are required to explicitly declare stable variables because of the high overhead associated with both tracing writes to a stable variable (approximately 2 times the cost of a write to a volatile variable) and recording changes in a disk-based stable server log (approximately 100 times the cost of a write to a volatile variable). Section 6.7 provides a detailed discussion of the costs associated with using stable variables.

Failure Recovery Procedures

Each RDO can use the Tcl function `Rover_setRecoveryProc` to specify the name of a special Tcl procedure to be used during recovery; the server saves the procedure name in the stable server log. In addition, each server application can specify a C recovery procedure that will be invoked during the recovery process. The toolkit provides routines that application developers can use to perform the necessary failure recovery steps.

During failure recovery, the Rover server scans its stable server log looking for any QRPCs that were received but not processed or that were processed but whose results were not delivered to the client. Unprocessed QRPCs are handled as detailed in Section 3.3.2. For incomplete QRPCs, the server invokes the application's and RDO's recovery procedures. If the C failure recovery procedure is specified, it is responsible for performing the following actions:

1. Restoring the application's or RDO's C environment. This action consists of initializing any local or static variables used by the request.

2. Restoring the application's or RDO's Tcl environment. This action consists of loading any Tcl libraries and RDOs used by the request.

3. Invoking a toolkit-provided function, `restoreStableVars`, to restore the RDO's stable variables. During failure recovery, the values of stable variables in the sta-

ble server log take precedence over the initial values specified by `Rover_stable` declarations. The reason for this precedence order is that the logged values represent later states of the variable.

4. Invoking the RDO's Tcl recovery procedure, if one was specified. When the Tcl recovery procedure is invoked, it should recreate any volatile data that was being used at the time of the fault and then resume the computation that was in progress.

5. If no Tcl recovery procedure is specified, the C recovery procedure should reexecute the QRPC.

Part (a) of Figure 4-2 provides an example of a simple Tcl counter that can be created at a client and shipped to a server for execution. The corresponding version using stable variables is provided in part (b). As can be seen from the figure, it is very simple to declare and use a stable variable. Once declared, a stable variable can be treated as any other global variable. No Tcl recovery procedure is needed for the counter code. During failure recovery, the C recovery procedure restores the latest saved value of the counter and resumes execution of the code.

Reliable Application Development

Creating new reliable applications and proxies or modifying existing ones involved two steps: identifying the objects that should be made stable and providing the necessary recovery procedures.

The first step is usually simple. The best candidates for stable variables are objects that are expensive to recreate (computationally or otherwise); or represent nonrecoverable data, such as a dynamic data stream. Objects that are logged should be small. Otherwise, the cost of logging each object may outweigh the costs of recreating it.

For the second step, the Rover toolkit provides application developers with help in creating the necessary recovery procedures. As outlined in the previous section,

the toolkit provides procedures for performing several of the recovery steps, including procedures for restoring stable variables and invoking the Tcl recovery procedure.

4.4 Implementation Summary

This chapter detailed the reference implementation of the Rover toolkit. The implementation of the Rover toolkit client library consists of approximately 1,200 line of Tcl/Tk code and 24,000 lines of C code. The server library consists of 1,100 lines of Tcl/Tk code and 8,400 lines of C code. There is an additional 9,500 lines located in a library that is common to both client and server.

Chapter 5

Applications Using Rover

This chapter discusses the steps involved in implementing new mobile-adaptive applications, porting existing applications to a mobile-adaptive environment, and constructing proxies for mobile-transparent applications. The chapter also defines the programming interface provided by the Rover toolkit and describes the sample applications that have been constructed using the toolkit.

The application porting process is based upon converting a file system-based application into an object-based application. For the the applications discussed in this section, the process consisted of porting all of the application's data to an object-based model. However, on-going research with the Rover File System Proxy, discussed below in Section 5.3, will make the process incremental and gradual.

5.1 Using Objects Instead of Files

There are several steps involved in porting an existing application to Rover, creating a new Rover-based application or constructing a proxy. Each step requires the application developer to make one of several implementation choices. The choices used in developing the initial set of Rover applications and proxies are presented in Table 5.1. While Rover does not provide any mechanical tools for building applications or proxies, it does provide a consistent framework.

The first step is to split the application or proxy into components and identify

Issue	Choice
Object Design	Use RDOs that encapsulate sufficient state to effectively service local requests, but are small enough to easily prefetch.
Computation Migration	Use RDOs to migrate computation that requires high bandwidth access.
Notification	Use colors and text to notify users of tentative information. Use Rover Toolkit functions to
	query environmental information.
Replication	Use RDOs to replicate information.
Consistency	Use logs of operations to detect conflicts and help resolve them.
Object Prefetching	Tradeoff of RDO size versus easier prefetching, but have to avoid overly aggressive prefetching.

Table 5.1: Implementation choices for the initial application set built using the Rover toolkit.

which components should be present on each side of the network link. It is very important that application developers think carefully about how application or proxy functions should be divided between a client and a server. The division will be mostly static, as most of the file system components will remain on the server and most of the GUI components will remain on the client. However, those components that are dependent upon the computing environment (network or computational resources) or are infrequently used may be dynamically generated and migrated. For example, the search operation performed by a client could be dynamically customized to the current link attributes: over a low-latency link, more work could be done at the client and less at the server, and vice versa for a high-latency link. Likewise, the main portion of an application's help information could be prefetched by a client, but less frequently referenced portions could be loaded on demand.

Once the application or proxy has been split into components, the next step is to appropriately encapsulate the application's or proxy's state within objects that can be replicated and sent to multiple clients. For example, a user's electronic mail consists of messages and folders. In a traditional distributed computing environment, one encapsulation is to store each message in an individual file and use directories to

group the messages into folders.

Metadata information for messages (*e.g.,* the size or modification date of a message) is determined by using file system status operations. In the mobile computing environment, the corresponding encapsulation stores messages as objects and folders as objects containing references to message objects. Each object encapsulates both the message or folder data and the appropriate metadata.

In migrating to the mobile environment, an application's or proxy's reading of files is replaced by the importing of objects and its writing of files is replaced by the exporting of changes to objects. The file system interface still exists in the server-side of the application. However, inserted between the two halves of the application or proxy is an object layer.

One of the primary purposes of the object layer is to provide a means of reducing the number of network messages that must be sent between the client and server; this reduction is done by migrating computation. Consider the E-mail folder scan operation, which returns a list of messages and information about the messages in a folder. Using a file system-based approach means scanning the directory for the folder, opening each message, and extracting the relevant information. This approach is an appropriate operation for a well-connected host, but would be very expensive and time-consuming over a high-latency link. Using an object-based approach, the server-side application constructs a folder object containing the metadata for the messages contained in the folder. The client-side application can then import the folder object in a single roundtrip request and avoid multiple roundtrip requests. The multiple requests are replaced by local computation – querying the folder object about the messages it contains.

The next step is to add support for interacting with the environment. For example, in the E-mail example, one of the important pieces of message metadata that a folder object contains is the message's size and the size of any attachments. This information can be used by the application and conveyed to the user to allow useful decisions to be made.

Support for prefetching is another environment interaction issue. For example, the

server-side HTTP proxy automatically prefetches inlined images. Also, the application developer must decide which mechanisms to use for notifying users of the status of displayed data.

The final important step is the addition of application-specific conflict resolution. For most stationary environments, conflicts are infrequent. However, for the mobile environment, they will be more common. Fortunately, application developers can leverage the additional semantic information that is available with Rover's method-based, instead of value-based, approach to object updating.

5.2 Toolkit Programming Interface

The programming interface between Rover and its client applications or proxies contains four primary functions: *create session*, *import*, *invoke*, and *export*. Client applications call *create session* once with authentication information to set up a connection with the local access manager and receive a session identifier. The authentication information is used by the access manager to authenticate client requests sent to Rover servers.

To import an object, an application calls *import* and provides the object's unique identifier, the session identifier, and optional callback and arguments. In addition, the application specifies a priority that is used by the network scheduler to reorder QRPCs. The *import* function immediately returns a promise [55] to the application. The application can then wait on this promise or continue execution. Rover transparently queues QRPCs for each import operation in the stable log. When the requested object is received by the access manager, the access manager updates the promise with the returned information. In addition, if a callback was specified, the access manager invokes it.

The current implementation also has a *load* operation that is an *import* combined with a call to create a process. Applications use the *load* operation to import RDOs that need a separate thread of control. When the access manager receives an RDO that was requested by a *load*, it creates a separate process and executes the RDO. The

reason for a separate *load* operation is historical. At the time that the prototype was implemented, the underlying development operating systems (the UNIX-based Linux and SunOS operating systems) did not support multiple threads per address space and only provided limited support for dynamic linking. In a future implementation, *load* may be directly incorporated within *import*.

Once an object is imported, an application can *invoke* methods on it to read and/or change it. Applications export each local change an object back to servers by calling the *export* operation and providing the object's unique identifier, the session identifier, a callback, and arguments. Like *import*, *export* immediately returns a promise. When the access manager receives responses to exports, it updates the promise and invokes any application-specified callbacks.

5.3 Rover Application Suite

The design and architecture chapter of this thesis (Chapter 3) discusses several important issues in designing mobile-adaptive applications. This section provides examples of how those issues are addressed in several mobile-adaptive applications and proxies for mobile-transparent applications that have been developed using the Rover toolkit (Table 5.1 lists the major implementation issues). The two proxies for mobile-transparent applications are: *Rover NNTP proxy*, a USENET reader proxy [13]; and *Rover HTTP proxy*, a proxy for Web browsers. The mobile-adaptive applications are: *Rover Exmh*, an E-mail browser; *Rover Webcal*, a distributed calendar tool; *Rover Irolo*, a graphical rolodex tool; and *Rover Stock Market Watcher*, a tool that obtains stock quotes.

An additional proxy for mobile-transparent applications, the *Rover File System proxy* (RFS), is under development. This application demonstrates the true versatility of the Rover toolkit, as it will allow many existing application to run unmodified in a mobile environment by providing a standard file system interface on top of the toolkit. The proxy will provide application developers with the flexibility to gradually and incrementally migrate applications from a file system-based model to an object-based

Rover Program	**Original code**	**New Rover client code**	**New Rover server code**
Rover Exmh	24,000 Tcl/Tk	1,700 Tcl/Tk 220 C	140 Tcl/Tk 2,400 C
Rover Irolo	470 Tcl/Tk	420 Tcl/Tk 220 C	280 Tcl/Tk 560 C
Rover Stock Watcher	none	200 Tcl/Tk 220 C	160 Tcl/Tk 260 Perl, 310 C
Rover Webcal	26,000 C++ and Tcl/Tk	2,600 C++ and Tcl/Tk	1,300 C++ and Tcl/Tk
Rover File System Proxy	none	55 Tcl/Tk 3,000 C	340 C
Rover HTTP Proxy	none	210 Tcl/Tk 3,200 C	1,600 C
Rover NNTP Proxy	none	510 Tcl/Tk 223 C	350 C

Table 5.2: Lines of code changed or added in porting or implementing *Rover Exmh*, *Rover Irolo*, *Rover Stock Watcher*, and *Rover Webcal* and implementing *Rover File System Proxy*, *Rover HTTP Proxy*, and *Rover NNTP Proxy*.

model.

Two of the mobile-adaptive applications are based upon existing UNIX applications. Rover Exmh is a port of Brent Welch's Exmh Tcl/Tk-based E-mail browser. Rover Webcal is a port of Ical, a Tcl/Tk and C++ based distributed calendar and scheduling program written by Sanjay Ghemawat. Rover Irolo and the Rover Stock Market Watcher were built from scratch.

This application suite was chosen to test several hypotheses about the ability to reasonably meet users' expectations in a mobile, intermittently-connected environment. These applications represent a set of applications that mobile users are likely to use.

In addition to using standard quantitative techniques to measure the performance and efficiency of the toolkit, it is important to qualitatively test the ideas contained in the Rover toolkit by using the toolkit to construct complete applications. Qualitative tests are necessary to investigate how using RDOs affects the structure of applications and the ease of programming.

As can been seen in Table 5.2, porting these file system-based workstation applications to a mobile-adaptive Rover applications requires varying amounts of work. For example, porting *Exmh* and *Ical* to Rover required simple changes to approximately 10% of the lines of code. Most of these changes came from replacing file system calls with object invocations; these modifications in *Rover Exmh* and *Rover Webcal* were made almost independently of the rest of the code.

Recent applications were written/ported in a few weeks, while some of the earlier applications required several person-months of work. Earlier applications required more time because the architecture of toolkit was changing while the applications were being ported. Many of the changes were a direct result of the experiences learned while porting the applications.

The Rover HTTP and NNTP proxies demonstrate how Rover mobile-adaptive proxies support existing applications (*e.g.,* Netscape and XRN) without modification. Creating these proxies for these services is far easier than modifying all the applications that use these services.

5.3.1 Proxies for Mobile-Transparent Applications

All of the proxies that have been developed using the Rover Toolkit are read-only. Adding support for write operations and for dynamically changing the priority of requests (*i.e.,* converting prefetch requests into foreground requests) are areas of ongoing and future research.

Rover NNTP proxy. Using the Rover NNTP proxy, users can read USENET news with standard news readers while disconnected and receive news updates even over very slow links.

The Rover NNTP proxy consists of two components: an NNTP proxy located on the client and a remote NNTP proxy running on a Rover server. The client NNTP proxy is implemented as a Rover client application and uses Rover's reliable, queued communication mechanism with automatic message compression and batching. The server proxy is linked with the Rover server using the server's application plug-in interface.

Whereas most NNTP servers download and store all available news, the Rover proxy cache is filled on a demand-driven basis. When a user begins reading a newsgroup, the NNTP proxy loads the headers for that newsgroup as a single RDO while articles are prefetched in the background. As the user's news reader requests the header of each article, the NNTP proxy provides them by using the local newsgroup RDO. As new articles arrive at the server, the server-side of the proxy constructs operations to update the newsgroup-header object. Thus, when a news reader performs the common operation of rereading the headers in a newsgroup, the NNTP proxy can service the request with minimal communication over the slow link.

Rover HTTP proxy. This application interoperates with most of the popular Web browsers. It allows users of existing Web browsers to "click ahead" of the arrived data by requesting multiple new documents before earlier requests have been satisfied.

The Rover HTTP proxy consists of two components: an httpd proxy located on the client and a remote httpd proxy running on a Rover server. The client proxy uses Rover's reliable, queued communication mechanism with automatic message compression and batching and is implemented as a set of modules linked with the access manager. This implementation choice was done to help with the design of the access managers external interfaces. The other Rover client proxies were constructed as Rover client applications. The server proxy is linked with the Rover server using the server's application plug-in interface.

The client proxy intercepts all local web requests and, if the requested item is not locally cached or is already being fetched, returns a null response code (HTTP response 204) to the browser and uses QRPC to enqueue the request in the operation log. When a connection becomes available, the network scheduler forwards the request to a Rover server. In the meantime, the user can continue to browse already available pages and issue additional requests for pages without waiting. The granularity of RDOs is individual pages and images.

The server proxy fetches the documents requested by client proxies and automatically applies compression to the returned documents. The compression is either Rover's automatic compression or application-specific compression. In addition to

compression, the client and server proxies cooperate in prefetching.

The client proxy specifies the depth of prefetching for pages, while the server proxy automatically prefetches links and inlined images to the depth specified by the client. The client proxy also parses pages as they arrive and builds reference lists of inlined images and links. Depending upon the specified prefetch depth, these reference lists indicate pages and images that the client proxy is expecting the server proxy to prefetch for it. Thus, the reference lists can be used by the client proxy to detect browser requests for pages that will be prefetched by the server proxy (e.g., a browser request for an inlined image will be ignored if the client proxy is already expecting the inlined images from the server proxy).

The client proxy tracks the status (request pending, request sent, arrived and not viewed, or arrived and viewed) of all HTTP objects (pages and images) that are requested directly by a web browser or indirectly due to the prefetching depth or are stored in the local Rover object cache. Users have two choices for viewing dynamically generated lists of the HTTP objects in the cache and the QRPC log. Both choices expose the object cache and QRPC log directly to the user by providing dynamic views of their HTTP-related contents and allowing the user limited control over their contents (*e.g.,* cancel request, delete object, and reload object).

One choice for viewing is a dynamically generated HTML list. When the web browser requests the page (http://home/cache), the client proxy generates the list for the browser. The usual setup is for the web browser to have two windows, one displaying the cache contents and request list and the other displaying the desired web pages. The list page window includes HTML links with HTML `target` commands to automatically cause the other browser window to request a link when it is clicked on in the cache list window. The other choice for viewing the list is a Tk window application. This application can directly control NCSA's *Mosaic* [61] and NCC's *Netscape Navigator* [62] browsers using their remote control interfaces.

One interesting note about the HTTP proxy is that the techniques of prefetching and compression that it pioneered have been adopted by several commercial web browsers and are in the process of being added to HTTP standards [23, 63].

Rover File System proxy. This application makes the process of porting a mobile-transparent application to a mobile environment easier by addressing a significant problem: the need to completely shift all of the application's data storage from one model (a file system) to a model that is better suited for disconnected operation (an object repository or database). RFS makes the porting process gradual and incremental by allowing the Rover toolkit to support both a file model and an object model for mobile applications.

Adding RFS to the toolkit raises a number of issues relating to file caching, prefetching, and conflict detection and resolution. Many of these issues have already been addressed to some extent by the Coda file system and others [33, 34, 47, 48, 58, 71], however, RFS is also addressing the issues associated with integrating a file system model with an object-based model. Specifically, how to map file system objects (inodes, directories, and files) onto Rover objects, cluster objects for prefetching, use Rover's conflict detection and resolution model, and partially replicate files.

RFS, as a part of the Rover toolkit, offers application developers a simple alternative: those portions of an application's state that are difficult to port, that change infrequently, or exhibit update conflicts that are easy to detect and resolve can rely on RFS for storage. On the other hand, application state that is sensitive to bandwidth and latency, or where conflicts may be complicated to detect or handle, can use the full functionality of Rover's object management support as well as its conflict resolution and detection. This set of choices will significantly ease the process of adding mobile-adaptive support to applications.

RFS consists of two components: a user-level installable file system located on the client and a remote file system proxy running on a Rover server. The RFS client proxy uses Rover's reliable, queued communication mechanism with automatic message compression and batching. The RFS client proxy is composed of a small kernel module and the RFS process, which runs as a Rover client application. RFS makes use of two caches: Rover's shared memory/disk cache, and a small in-memory cache managed directly by RFS. The latter cache is used to minimize the amount of communication between the RFS client proxy and the Rover toolkit. The server proxy is

linked with the Rover server using the server's application plug-in interface.

The prototype implementation supports disconnected operation by caching remote inodes, files, and directories; however, it is currently limited to read-only access. Starting with a simple read-only implementation allowed for immediate analysis of performance and experimentation with various strategies for prefetching, caching, cache (re)validation, and conflict resolution.

Future plans for RFS include extending the proxy to provide full support for both read and write, as well as prefetching, hoarding, and disconnected operation with lease-based concurrency control and server callbacks. Extending RFS will require addressing many of the issues raised by the Coda file system and others, while further tuning RFS performance.

The Rover File System will provide the final component of a complete development environment. Using RFS, application writers will be able to gradually and incrementally port existing mobile-transparent applications to a mobile-adaptive environment. This support will be an important incentive for migrating to a mobile computing environment.

5.3.2 Mobile-Adaptive Applications

Rover Exmh. *Rover Exmh* uses three types of RDOs: mail messages, mail folders, and lists of mail folders. By using this level of granularity, many user requests can be handled locally without any network traffic. Upon startup, Rover Exmh prefetches the list of mail folders, the mail folders the user has recently visited, and the messages in the user's inbox folder. Alternatively, using a finer-level of granularity (*e.g.,* header and message body) would allow for more prefetching, but could delay servicing of user requests, especially during periods of disconnection. In the other direction, using a larger granularity (*e.g.,* entire folders) would seriously affect usability and response times for slow links.

Some computation can be migrated to servers. For example, instead of performing a glimpse search of mail folders locally at the client and thus having to import the index across a potentially low bandwidth link, the client can construct a query request

RDO and send it to the server.

The GUI indicates that an operation is tentative using color coding. Conflict detection is based upon a log of changes to RDOs; the log allows the server to detect and resolve a conflict such as one user adding a message to a folder and another user deleting it. Unresolvable conflicts are reflected back to the user.

Rover Webcal. This distributed calendar tool uses two types of RDOs: items (appointments, daily todo lists, and daily reminders) and calendars (lists of items). At this level of granularity, the client can fetch calendars and then prefetch items using a variety of strategies (*e.g.,* plus or minus one week, a month at a time, etc.).

Rover Webcal uses color coding to aid the user in identifying those objects that have been locally modified but not yet propagated to a server. Conflict detection is based upon a log of changes to RDOs; this log allows the server to detect and resolve a conflict such as one user adding an item to a calendar and another user deleting it.

Rover Irolo. This graphical rolodex application uses two types of RDOs: entries and indices (lists of entries). The GUI displays the last time an entry was updated and indicates whether the item is committed or tentative. Conflict detection is based upon a log of changes to RDOs; this log allows the server to detect and resolve a conflict such as one user adding an entry to an index and another user deleting it.

Rover Stock Market Watcher. This application uses both computation migration and fault-tolerance techniques [39, 40]. The client constructs RDOs for stocks that are to be monitored and sends them to the server. The server uses Rover's server-side fault-tolerant support to store the real-time information retrieved from stock ticker services.

5.3.3 Developing Reliable Applications

To measure the effects of Rover's support for server-side reliable execution on end-to-end application performance, the reliability extensions were added to two applications and a proxy: a stock market watcher application, a simple file search application, and the server portion of the Rover Web Browser proxy (described in more detail in Section 5.3). The applications and proxy were chosen to evaluate the ease of con-

structing reliable applications or modifying existing applications and to measure the performance of reliable applications (see Section 6.7.2).

5.3.4 Stock Market Tracker

As an exercise to gauge the added difficulty in constructing reliable applications, the reliability extensions were added to a simple financial stock market tracking application.

Operation of the application is as follows:

1. The user specifies a stock, an attribute for the stock (*e.g.,* price, volume, trend, or changes), and a threshold. The user is notified when the attribute for the stock exceeds the threshold.

2. The client-side application constructs a stock watching RDO containing the user's parameters and sends the RDO to the server.

3. The server-side application executes the RDO. When the RDO indicates that the user-specified threshold has been exceeded, the server-side application notifies the client.

When a server-side failure occurs, all volatile information is lost. As a result, any trend data that the RDO was generating and using that data is lost. There are two alternatives for saving the stock's attributes: manually generate RDOs that use the filesystem to store the information or use stable variables for the attributes.

Manually making the information stable would require an effort comparable to the effort required to implement stable variables (*e.g.,* constructing data marshalling and unmarshalling procedures). Instead, the reliable extensions were used to make the application's information persistent across server failures.

The application uses stable variables to store the stock's current attributes. As with the other examples of reliable applications, the changes only affected a few lines of code.

5.3.5 Rover Web Browser proxy

The Rover Web Browser proxy is a client-server application consisting of a client proxy on the client machine and a server proxy on the server machine (see Section 5.3.1). The modifications to support reliable operation only affected the server portion of the application.

When the server receives a request from a client for a web page, it fetches the page and returns it to the client. It also prefetches any inlined objects and returns them to the client in the same connection (thus avoiding multiple connection setups for each inlined object). Since multiple pages may refer to the same set of inlined objects (*e.g.,* logos, bullets, etc.), the server uses a hash table to keep track of the timestamps and sizes of those objects that it has already sent to a client.

The proxy was modified to make the hash table it uses stable. This change was a simple modification consisting of changes to 55 lines of code. The primary benefit of these simple changes is that after a server failure, the stable hash table allows the server to avoid sending duplicate objects to clients, conserving potentially expensive and/or limited bandwidth.

Experimental evaluation of the modifications shows that they have a negligible effect on performance (see Section 6.7.2). Other costs (httpd server overhead and transport cost) and their variability dominate the server execution times.

5.3.6 Text file search

A simple text file search application was constructed to explore the ease of using stable variables. The application uses stable variables to store information about the state of the search (*i.e.,* the remaining files and directories to be searched and the matches found so far). An excerpt from the server code is provided in Figures 5-1 and 5-2. The difference between the stable and volatile versions is only a few lines of code. After a failure, the stable version can resume the search in-progress with only a small amount of lost work.

Experimental evaluation of the modifications shows that, when there are no faults,

```
# Declare the stable variables
#    currentFiles: list of files to be searched in the current directory
#    currentDirs: list of directories to be searched
#    currentResult: results of search so far
if {![info exists currentFiles]} {Rover_stable currentFiles ""}
if {![info exists currentDirs]} {Rover_stable currentDirs ""}
if {![info exists currentResult]} {Rover_stable currentResult ""}

# Main search procedure
#    Search for (search) in all files and directories in or below (topdir)
proc search {topdir search} {
    global currentDirs currentResult

    # currentDirs will already be set by the recovery procedure
    #   if we're recovering
    if {$currentDirs == ""} {set currentDirs $topdir}

    # Iterate through the list of directories
    for {} {[llength $currentDirs] > 0} {} {
        # Get the next directory to search
        set dir [lindex $currentDirs 0]

        # Expand the current dir and append it for breadth-first search
        set newList [concat $currentDirs [findDirs $dir]]

        # Search the directory and save the result
        searchDir $dir $search

        # Update the list of directories to search
        set currentDirs [lrange $newList 1 end]
    }
    return $currentResult
}
```

Figure 5-1: Server-side code for a reliable file search application (part one).

the performance using logged stable variables with asynchronously flushed buffers is nearly identical to the performance using volatile variables (see Section 6.7.3). When there is a failure, the reliable version of the application performs significantly better than the volatile version.

```
# File search procedure
#    Search for (search) in all files in (dir)
proc searchDir {dir search} {
    global currentFiles currentResult

    # currentFiles (and tempResult) will already be set if we're recovering
    if {$currentFiles == ""} {set currentFiles [findFiles $dir]}

    # Iterate through the list of files
    while {[llength $currentFiles] > 0} {

        # Execute grep on each file
        set fname [lindex $currentFiles 0]
        if ![catch {exec /usr/bin/fgrep $search "$dir/$fname"} data] {
            # Save the result
            if {$data != ""} {
                lappend currentResult [list "$dir/$fname" $data]
            }
        }

        # Update the list of files to search
        set currentFiles [lrange $currentFiles 1 end]
    }
}
```

Figure 5-2: Server-side code for a reliable file search application (part two).

5.4 Discussion

The experimental results presented in this section confirm the hypotheses listed at the beginning of this chapter.

At the micro benchmark level, batching and compression of multiple requests yields significant performance benefits for slower networks — increased throughput and reduced average latency. In addition, stable variables are easy to use, offer significant performance gains in the presence of failures, and, with asynchronous logging, yield performance comparable to that for volatile applications.

At the system level, unmodified mobile transparent applications gain usability and performance benefits from the Rover toolkit. For mobile-adaptive applications, the usability and performance gains are more substantial. Rover allows users to see

the same excellent GUI performance across a range of networks that varies by nearly three orders of magnitude in both bandwidth and latency.

Finally, when using asynchronous logging, the performance of reliable applications is comparable to that of volatile applications.

Chapter 6

Experiments

This chapter contains the results of experiments designed to explore several hypotheses about the Rover toolkit. The experiments fall into two categories: low-level benchmarks and system-level experiments.

The low-level benchmarks test the following hypotheses:

1. Even with the overhead of stable logging, using QRPC instead of RPC significantly improves performance by enabling batching and compression of multiple requests and responses.

2. Stable variables offer significant performance gains in the presence of failures. However there must be a clear demarcation between stable and volatile variables due to the overhead associated with stable variables.

The system-level experiments test the following hypotheses about end-to-end performance:

1. Mobile-transparent applications benefit from using the Rover toolkit.

2. Mobile-adaptive applications offer significant performance advantages over existing non-adaptive versions of the applications.

3. Applications using Rover's reliable execution support gain significant performance improvements over ordinary applications in the presence of failures and have comparable performance when there are no failures.

6.1 Experimental Environment and Methodology

The network transports used for the experiments included a variety of wired and wireless local-area and wide-area network technologies:

1. 10 Mbit/s switched Ethernet. All of the machines used in the test were connected to the same segment. The segment used for the experiments was relatively idle during the experiments.

2. 2 Mbit/s wireless AT&T WaveLAN. The WaveLAN base station used for the experiments was placed on the same Ethernet segment as the server. In addition, the base station was logically isolated from other base stations in the building. However, WaveLAN's physical radio layer is a shared media; to minimize the impact of shared access on the experiments, the experiments were run during idle periods.

3. 128 Kbit/s and 64 Kbit/s Integrated Digital Services Network (ISDN) links. On the remote end of the link, the client laptop was connected via an idle Ethernet segment to an Ascend Pipeline 50 ISDN router. The router was connected to a local Ascend Pipeline 50 router using the Public Switched Telephone Network (PSTN). The local router was connected to the Rover server using the building's networking infrastructure.

4. Serial Line IP with Van Jacobson TCP/IP header compression (CSLIP) [36] over 28.8 Kbit/s *V.34* wired and 9.6 Kbit/s *V.32* analog cellular dial-up links. For the experiments, the client used the PSTN to connect to the building's terminal server modem pool. The cellular link supports 14.4 Kbit/s *V.32* connections, however at the data higher rate, the link suffers from significantly higher error rates.

These networks are representative of the networks that are available in the local and wide area. Ethernet is available in most office workplaces, while multi-megabit wireless office networks are being deployed in many offices. Wired dialup PSTN network

connections are the most widely deployed remote network access solution, while ISDN is gradually becoming available in many locations as a cost-effective home-office network connectivity solution. Wireless wide-area network connectivity, however, is still in its infancy — there are a variety of proprietary solutions that are available in limited geographic areas. Analog cellular networks are an open standard and their coverage reaches most metropolitan and suburban locations.

It is important to note that ordinary TCP/IP was used over all the networks, including the wireless networks. While Rover applications might benefit from the use of a specialized TCP/IP implementation, it is not a necessary requirement. Another advantage of using Rover is that Rover application sends less data than an unmodified application; thus, Rover applications are also less sensitive to errors on wireless links.

The test environment consisted of a single server and multiple clients. The standalone TCP/IP server executed on an Intel 200 Mhz Pentium Pro workstation. The clients were IBM ThinkPad 560 laptops (133 Mhz Pentium). The HTTP server used in the HTTP and web browser experiments was an Intel Advanced/EV (120 Mhz Pentium) workstation. All of the machines were otherwise idle during the tests.

The clock resolution on the ThinkPad and Pentium Pro is 10.0 milliseconds.

Except for Ethernet, the traffic in all of the networks uses shared public resources and traversed shared links. As such, there is increased variability in the experimental results for those network transports. To reduce the effects of the variations on all of the experiments, each experiment was executed multiple times and the results averaged. In addition, most of the experimental results include 90% confidence intervals.

6.2 Null QRPC Performance

The baseline performance for QRPC was established using TCP latency and bandwidth measurement experiments. The TCP latency experiment measured the time to open a TCP connection, send two bytes, receive two bytes, and close the connection. The TCP bandwidth experiment measured the time to transmit 1 MByte of ASCII data.

Transport	TCP	
	Throughput 1 MByte (Mbit/s)	Latency null RPC (milliseconds)
Ethernet	8.4 ±0.08	2.4± 0.72
WaveLAN	0.79 ±0.17	12 ± 0.85
128 ISDN	0.44 ±0.03	69 ± 3.19
64 ISDN	0.21 ±0.01	73 ± 2.43
28.8 Wired CSLIP	0.023±0.01	310 ± 7.51
9.6 Cellular CSLIP	0.017±0.02	620 ±12.20

Table 6.1: The Rover experimental environment. Measurements include 90% confidence intervals.

The cost of a QRPC has several primary components:

1. Client-side processing costs. The overhead associated with constructing a request, compressing the request, logging the request to stable storage, and decompressing the server's response.

2. Transport cost. This cost is the time to transmit the request and receive the reply: the TCP null RPC cost from Table 6.1 plus the per-byte network transmission cost.

3. Server-side processing costs. The time to process the QRPC at the server, including parsing the QRPC, decompressing the request, and generating and compressing the response. For the experiments, no server-side authentication was performed and no application code was executed.

Combining these costs yields a model of the approximate average time to perform a QRPC, based upon client and server processing power and network parameters. The model is presented in Figure 6-1 and the costs are analyzed in detail in the following sections. Where appropriate, costs are presented in terms of processor cycles. Note that the client and servers have different processor architectures (Pentium and Pentium Pro); thus, comparisons of cycle counts between the two may not by appropriate.

Host	Operation	Overhead		Per-byte	
		(milliseconds)	(cycles)	(ms)	(cycles)
Client	Comp	8.0 ±0.01	1,100,000± 1,330	0.0025	330
	Decomp	0.97±0.0005	130,000± 67	0.004	530
Server	Comp	2.4 ±0.07	480,000±14,000	0.027	5,400
	Decomp	0.58±0.003	120,000± 600	0.00004	8

Table 6.2: Costs for using Rover compression and decompression algorithms. Measurements include 90% confidence intervals.

The QRPC measurements used a synthetic benchmark consisting of a client issuing a series of 50 null QRPCs with no interval between requests. This benchmark is representative of what occurs when the client prefetches or imports a series of RDOs. For example, when Rover Exmh starts executing at the client, it imports the 48 RDOs that compose the client GUI. Afterwards, it imports the RDOs containing the recent messages in the user's inbox. In similar fashion, Rover Webcal issues multiple imports for the calendar entries.

Modern mobile computers have fast processors. Most mobile applications, however, are communication-bound; thus, processor cycles are available for compression and decompression algorithms. The benchmark application is representative of a communication-bound mobile application.

Each experiment was performed with each of Rover's automatic request compression and heuristic batching functions enabled and disabled. Batched requests were sent as one single batch of 50 requests.

The request and result data for each null QRPC consisted of sending 14 bytes and receiving 0 bytes The total amount of data sent and received depends upon the QRPC headers, since the headers include host address information and variable size message sequence number and identifier information. The message size numbers below are averages of the actual message sizes.

For an uncompressed, non-batched QRPC, a total of 211 bytes were sent and 53 bytes were received. When Rover's *zlib*-based compression is used with non-batched QRPC, the amount of data transmitted actually increases to 224 bytes sent and 61

bytes received. Applying compression to a single null QRPC (or its null result) yields more output than input, because QRPC request and response headers are already compact and the compression algorithm has fixed 12 byte headers.

6.2.1 QRPC Client Costs

The time for the client-side toolkit to construct a single QRPC was measured as 3.14 $\pm$ 0.25 milliseconds (418,000 $\pm$ 33,300 cycles). For a batch of 50 requests, the time was measured as approximately 25.5 milliseconds (3,430,000 cycles).

The time for the client-side toolkit to process the results of a single QRPC was measured as 0.247 $\pm$ 0.002 milliseconds (32,900 $\pm$ 260 cycles).

6.2.2 QRPC Compression Costs

There are eight costs associated with using the *zlib* compression and decompression algorithms. At clients and servers there is initialization overhead for compression and decompression; in addition, there are per-byte costs for compression and decompression. Table 6.2 lists the measured costs. The measured per-byte costs are a function of the amount of data being compressed; thus, the numbers in the table are approximate. The measurement experiments were performed with warm caches.

Client-side compression and decompression and server-side decompression is done using input and output memory buffers. Server-side compression is done using an input memory buffer and an output communication socket.

6.2.3 QRPC Transport Cost

The transport cost for a QRPC (t_{net}) is dependent upon the latency and bandwidth of the network technology. For comparison purposes, Table 6.1 provides TCP performance measurements for several representative networks. The table shows null RPC latency for a ping-pong over TCP sockets and the throughput for sending 1 MByte of ASCII data using TCP sockets.

$$
\begin{aligned}
\bar{t}_{\text{QRPC}} &= t_{\text{Client}} + t_{\text{net}} + t_{\text{Server}}, \\
\\
t_{\text{Client}} &= t_{\text{constr}} + t_{\text{cLog}} + f_{\text{cComp}}(sz_{\text{request}}) + f_{\text{cDecomp}}(sz_{\text{compResult}}) + t_{\text{resultProc}} \\
t_{\text{net}} &= t_{\text{connect}} + \frac{sz_{\text{compRequest}} + sz_{\text{compResult}}}{bw} \\
t_{\text{Server}} &= t_{\text{parse}} + f_{\text{sDecomp}}(sz_{\text{compRequest}}) + t_{\text{sLog}} + t_{\text{ServerApp}} + f_{\text{sComp}}(sz_{\text{result}}) \\
f_{\text{cComp}}(x) &= t_{\text{cCompOvr}} + t_{\text{cComp}} \times x \\
f_{\text{cDecomp}}(x) &= t_{\text{cDecompOvr}} + t_{\text{cDecomp}} \times x \\
f_{\text{sComp}}(x) &= t_{\text{sCompOvr}} + t_{\text{sComp}} \times x \\
f_{\text{sDecomp}}(x) &= t_{\text{sDecompOvr}} + t_{\text{sDecomp}} \times x
\end{aligned}
$$

where,

$$
\begin{aligned}
t_{\text{constr}} &= \text{Client QRPC construction time} \\
t_{\text{cLog}} &= \text{Client QRPC logging time} \\
sz_{\text{request}} &= \text{Size of marshalled request} \\
sz_{\text{compResult}} &= \text{Size of QRPC header and compressed marshalled result} \\
t_{\text{resultProc}} &= \text{Client result processing time} \\
t_{\text{connect}} &= \text{Network connection time (usually } t_{\text{latency}} \times 2\text{)} \\
bw &= \text{Network bandwidth} \\
sz_{\text{compRequest}} &= \text{Size of QRPC header and compressed marshalled request} \\
t_{\text{parse}} &= \text{Server time to parse QRPC} \\
t_{\text{sLog}} &= \text{Server QRPC logging time} \\
t_{\text{ServerApp}} &= \text{Server-side application's time to execute request} \\
sz_{\text{result}} &= \text{Size of result} \\
t_{\text{cCompOvr}} &= \text{Client compression algorithm overhead} \\
t_{\text{cComp}} &= \text{Client compression algorithm per-byte cost} \\
t_{\text{cDecompOvr}} &= \text{Client decompression algorithm overhead} \\
t_{\text{cDecomp}} &= \text{Client decompression algorithm per-byte cost} \\
t_{\text{sCompOvr}} &= \text{Server compression algorithm overhead} \\
t_{\text{sComp}} &= \text{Server compression algorithm per-byte cost} \\
t_{\text{sDecompOvr}} &= \text{Server decompression algorithm overhead} \\
t_{\text{sDecomp}} &= \text{Server decompression algorithm per-byte cost}
\end{aligned}
$$

Figure 6-1: Average time to perform a QRPC. When batching is used, the times t_{connect}, t_{cCompOvr}, and $t_{\text{sDecompOvr}}$ are amortized over the number of requests in the batch.

Disk	**Capacity (MBytes)**	**Seek time (ms)**	**Rot. Latency (ms)**	**RPM**	**Max. transfer rate (MByte/s)**	**Interface**
Quantum 2110S	2,111	10.5	6.7	4,500	10.0	SCSI-II
IBM DTNA-22120	2,120	13.0	7.5	4,000	16.6	IDE

Table 6.3: Attributes of the media used for stable logging (from manufacturer's specification sheets).

TCP throughput over Ethernet and WaveLAN is lower than expected because of software overhead and, for WaveLAN, media contention. TCP throughput over other network transports is higher than expected because of the compression performed by the modems and ISDN routers. A slower network means that the network queues grow giving the network hardware or software more data to apply compression to before transmission. For compressible data, more data to compress at one time results in better compression ratios. The 1 MByte of ASCII data used for the test is very compressible (GNU's *gzip -6* yields a 14.4:1 compression ratio). Since RDOs are constructed of Tcl scripts (ASCII), it is reasonable to expect that Rover applications will also observe similar compression benefits when using links with built-in compression. In contrast, the 1 MByte of random binary data used for the test was incompressible (a 1:1 compression ratio). A typical Rover binary, however, has a 3.30:1 compression ratio.

6.2.4 QRPC Stable Logging Cost

The cost of logging null QRPCs to disk was measured at client and server machines. The server has a PCI bus-based NCR 53c815 Fast SCSI-II interface to two Quantum Fireball 2110S disks. The clients have IDE-based IBM Travelstar DTNA-22120 disks. The attributes of the disks are listed in Table 6.3.

The client logging experiment measured the per-QRPC cost of logging 100 null QRPCs to the disk and synchronously flushing the file buffers after each logging action. The amount of data logged per QRPC was 39 bytes. The experiment yielded an average client logging time of 17.1 $\pm$ 0.47 milliseconds.

The server logging experiment measured the per-QRPC cost of logging 100 null

new QRPC, QRPC start, and QRPC result records. After the new and start records were logged, the file buffers were synchronously flushed — a total of two flush operations per QRPC. The amount of data logged per QRPC was 161 bytes. The result of the experiment was an average logging overhead of 18.1 ± 0.19 milliseconds.

Overall, the experimental results show that while client and server logging does have a cost (35.2 milliseconds), its relative impact on performance is a function of the transport media. Since most Rover users will often be connected via slower links (*e.g.,* wired or cellular dialup), the cost of stable logging will be a minor component of overall performance (*e.g.,* less than 5% for cellular links). For wired networks, asynchronous log flushing can be used at servers and clients. Using asynchronous log flushing yields performance comparable to when no logging is performed at the client or the server. Thus, for all networks, it is acceptable to pay the additional cost for client and server logging of QRPCs.

In addition, given that in a mobile environment the time to retransmit a QRPC from a client to a server after a failure is likely to be significant (*e.g.,* a client disconnected for an extended period of time or connected over a high latency link), it is acceptable to pay a small performance penalty for server logging of incoming QRPCs. Furthermore, by using asynchronous log flushing, a significant performance gain is achieved in exchange for a small window of vulnerability. For highly connected, relatively faultless environments, like Ethernet and WaveLAN, clients and servers can determine whether to synchronously or asynchronously flush log buffers. The cost of fault-tolerance need be paid only if necessary.

The use of an alternative to a file system, a different type of file system (*e.g.,* a log-based file system), a flash RAM card, or a small battery-backed static RAM card could offer substantial performance benefits over using an ordinary file system with a disk [21].

6.2.5 QRPC Server Costs

The measured time to parse, dispatch, and process a null QRPC at the server ($t_{\text{parse}} + t_{\text{ServerApp}}$) is approximately 0.31 milliseconds (62,000 cycles). The processing time

Network	Compression		No Compression	
	Actual	Model	Actual	Model
Ethernet	39± 2.9	41	51±11.	50
WaveLAN	49± 8.0	53	61± 1.8	63
128 ISDN	110± 7.6	110	110± 1.6	120
64 ISDN	140±17.	120	120± 5.1	130
28.8 Wired CSLIP	460± 7.5	440	490±16.	460
9.6 Cellular CSLIP	640±46.	780	610±12.	800

Table 6.4: Comparison between experimental and model results. Results are in milliseconds and include 90% confidence intervals.

only included the time to parse, dispatch, and execute the QRPC. Authentication information is sent with the QRPC, however no authentication is performed by the server.

6.3 Discussion of QRPC Costs

To validate the model of the end-to-end costs for a null QRPC, several experiments were performed. Each experiment measured the average roundtrip time to send a QRPC from a client to a server and to receive the result at the client. Table 6.4 provides a comparison between the experimental and model results.

The results show that the model provides a reasonably accurate representation of the costs for null QRPC. Three-quarters of the model's results are within 10% of the experimental results. Most of the differences are because the model does not completely account for the compression provided by some networks. This difference is especially visible in the cellular network results.

The QRPC tests involved transmitting only 5 KBytes of data. However, the network bandwidth numbers used for the models were those from Table 6.1 and involved transmitting a compressible 1 MByte text file. Due to wireless network errors, the large file throughput performance for a wireless networks will be lower than for small files. The QRPC performance model does not directly capture the effects of wireless errors. Such errors could be modeled by increasing the value for network's latency and

decreasing the value for the network's bandwidth. The errors could also be modeled by injecting errors during experiments and measuring their effect on performance.

The effects of compression on single and batched requests is investigated further in Section 6.4.

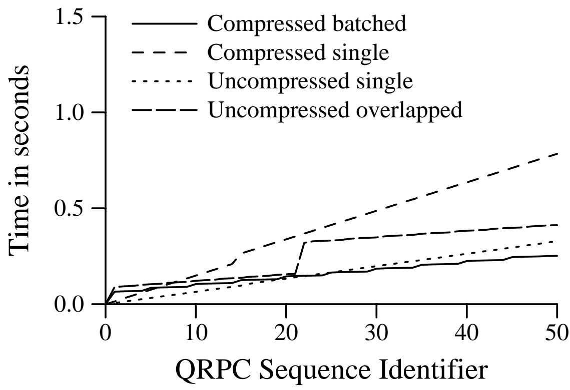
6.4 QRPC Batching and Compression

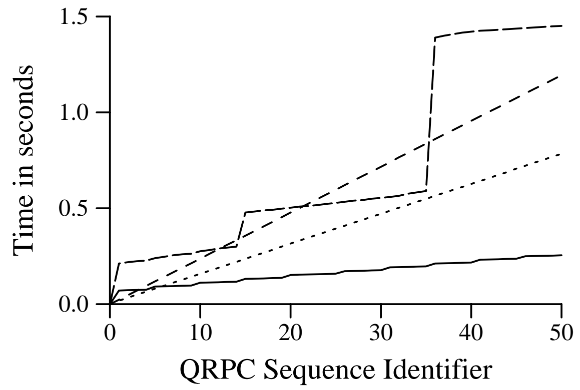
Figure 6-2 shows the effects of batching and compression on the performance of QRPC with asynchronous logging. Batching and compression is transparent to applications and it offers significant performance gains by eliminating multiple roundtrip messages and increasing the efficiency of the compression algorithms.

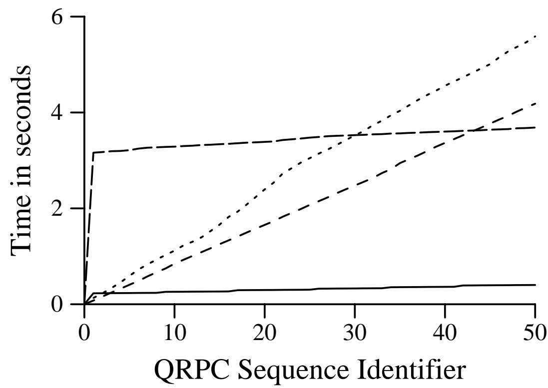
The experiments measured the effects of batching and compression using the heuristic from Chapter 3.3.4 on the time to perform a synthetic benchmark consisting of a client issuing a series of 50 null QRPCs with no interval between requests. This benchmark is representative of what occurs when the client prefetches or imports a series of RDOs. For example, when Rover Exmh starts executing at the client, it imports the 48 RDOs that compose the client GUI. Afterwards, it imports the RDOs containing the recent messages in the user's inbox. In similar fashion, Rover Webcal issues multiple imports for the calendar entries.

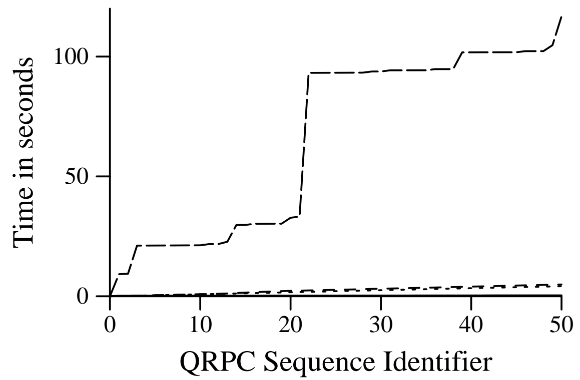
The measurements consisted of four experiments. The four lines in Figure 6-2 show the progress in executing QRPCs:

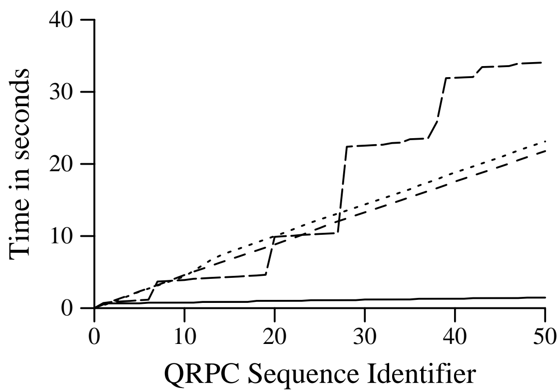
1. The *compressed batched* line shows the rate of progress when both Rover's automatic header and data compression and the network scheduler's automatic batching are applied. For this test, the compression ratio was approximately fifteen to one and the batch size was all fifty requests in one message. Since most of the data being sent is request headers and compression was applied across the entire batch of requests, it might appear that the compression ratio is skewed by using null QRPCs. However, as the results from Section 6.2.3 show, request data has a comparable compression ratio (14.4:1).











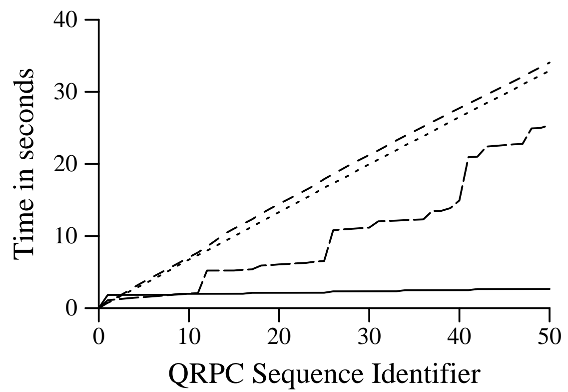


Figure 6-2: Time in seconds to execute 50 null QRPCs with asynchronous log record flushing.

2. The *compressed single* line shows the rate of progress with Rover's automatic header and data compression and only a single request outstanding. The compression ratio was 1.11 to one. The request headers are compact and have limited compressibility. As with the first measurement, non-null requests are usually more compressible than null requests. For faster networks, the computational overhead of compressing and decompressing requests is greater than the savings from sending less data.

3. The *uncompressed single* line shows the rate of progress without compression or batching and with only a single request outstanding. This line demonstrates that there are networks where Rover's compression is not as effective as the compression performed by the underlying network or where the execution time of applying software compression and decompression are greater than the time to transmit the uncompressed data.

4. The *uncompressed overlapped* line shows the rate of progress without compression or batching, but with multiple outstanding requests — one connection per request. The multiple outstanding requests caused two problems: request reordering and network resource contention. Although the requests were dispatched by the access manager in order, many requests were received at the server out of order. Also, the high number of simultaneous connections caused contention for network resources. The results of both problems were significant delays in request processing. This experiment is an example of what happens if applications are operating in an uncoordinated fashion (*e.g.,* special-purpose, single-use applications).

Overall, compression combined with batching offers performance gains with the largest gains occurring for the slowest networks. The main reason for the batching performance gain is the elimination of multiple roundtrip messages, reducing the slope of the completion curve. In addition, compression used with batching offers benefits because it allows multiple QRPC headers within a batch to be compressed. However, using batching imposes a delay on the time before a request is sent and the response

$$
\begin{aligned}
t_{\text{first}} &= t_{\text{constrBatch}} + f_{\text{cComp}}(sz_{\text{requestBatch}}) + f_{\text{cDecomp}}(sz_{\text{compResult}}) + t_{\text{resultProc}} \\
&\quad + t_{\text{connect}} + \frac{sz_{\text{compRequestBatch}} + sz_{\text{compResult}}}{bw} \\
&\quad + t_{\text{parse}} + f_{\text{sDecomp}}(sz_{\text{compRequestBatch}}) + t_{\text{ServerApp}} + f_{\text{sComp}}(sz_{\text{result}})
\end{aligned}
$$

where,

$$
\begin{aligned}
t_{\text{constrBatch}} &= \text{Client QRPC batch construction time} \\
sz_{\text{requestBatch}} &= \text{Size of marshalled batch request} \\
sz_{\text{compRequestBatch}} &= \text{Size of QRPC header and compressed marshalled batch request}
\end{aligned}
$$

Figure 6-3: Time to complete the first request in a batch of QRPCs.

$$
\begin{aligned}
\bar{t}_{\text{delta}} &= f_{\text{cDecomp}}(sz_{\text{compResult}}) + t_{\text{resultProc}} + \\
&\quad \frac{sz_{\text{compResult}}}{bw} + \\
&\quad t_{\text{parse}} + t_{\text{ServerApp}} + f_{\text{sComp}}(sz_{\text{result}})
\end{aligned}
$$

Figure 6-4: Time to complete subsequent requests in a batch of QRPCs.

is received.

To further analyze the effects of batching on performance, the analysis from the previous section was repeated. There are two metrics of interest:

1. The delay imposed upon the first request, t_{first} (see Figure 6-3). This delay is the cost of using batching. To compute t_{first}, the following values were used:
   - $t_{\text{constrBatch}}$ is the time to construct a QRPC batch. This time was measured as approximately 25.5 milliseconds (3,390,000 cycles).
   - sz_{request} is the size of the marshalled batch request. During the experiments, the amount of data sent as a single batch ranged from 4537 to 5737 bytes, depending upon the amount of header information. The amount of

header information is a function of the client host addresses used for the experiments.

- $sz_{\text{compRequest}}$ is the size of the compressed marshalled batch request. The amount of data sent ranged from 495 to 517 bytes. As with sz_{request}, the amount of data sent depended upon the amount of header information.
- Asynchronous logging was used for these experiments, thus t_{cLog} and t_{sLog} were approximately 0.
- For null QRPC, sz_{result} is 0 and $f_{\text{sComp}}(sz_{\text{result}})$ is 0.

2. The times for subsequent requests to complete, t_{delta} (see Figure 6-4). These times are represent the primary advantage of using batching, a reduction in the time between request completions. To compute t_{delta}, the following values were used:

   - The following values were set to 0 for the model: t_{constr}, f_{cComp}, t_{connect}, $sz_{\text{compRequest}}$, and f_{sDecomp}.
   - Asynchronous logging was used for these experiments, thus t_{cLog} and t_{sLog} were approximately 0.
   - For null QRPC, sz_{result} is 0 and $f_{\text{sComp}}(sz_{\text{result}})$ is 0.

Note that batching increases the efficiency of the compression algorithm. Without batching, the compression ratio between uncompressed and compressed data is 1:1.03. Using batching, the compression ratio varies from 9.17:1 to 11.1:1.

Table 6.5 shows the model's results and the actual experimental results for the t_{first} and t_{delta} metrics. The experimental results show that relative to issuing a single request at a time, batching imposes a delay on the first request. However, using batching also substantially reduces the time required for subsequent requests to complete. For example, on the cellular network, the first request completes in 1,800 milliseconds while subsequent requests complete on average once every 17 milliseconds. In comparison, the roundtrip time for a single request is 640 milliseconds (see Table 6.4).

Network	t_{first}		t_{delta}	
	Actual	**Model**	**Actual**	**Model**
Ethernet	65	50	3.6	4.4
WaveLAN	71	64	3.6	4.9
128 ISDN	230	130	3.4	5.4
64 ISDN	250	140	3.3	6.6
28.8 Wired CSLIP	660	560	16.	25.
9.6 Cellular CSLIP	1500	930	18.	32.

Table 6.5: Comparison between experimental and model batch results. Results are in milliseconds.

The model results show that for faster uncompressed networks, the model accurately predicts the expected value for t_{first}. The less accurate results for t_{first} are due to the TCP bandwidth models. The constants for the model are based upon sending ASCII file and are representative of the general compression attributes for a network. More or less compressible transmissions will yield different results. For example, when network compression is disabled on the cellular network, t_{first} is 2080 milliseconds and t_{delta} is 105 milliseconds.

Likewise, the results for t_{delta} show that the model accurately predicts the expected values for faster uncompressed networks. Networks with compression have the same effect on the t_{delta} model as on the t_{first} model. Nevertheless, the models provide useful information about expected application performance.

The single and batched QRPC models provide application developers with the tools that they need to predict application performance on a given network. Application developers can use the models to estimate application performance for existing and future networks.

6.5 Mobile-Transparent Application Performance

The performance of Netscape, using a mobile-transparent Rover HTTP proxy was compared against the same application executing independently. The experiment consisted of measuring the time to fetch and display ten WWW pages using a variety of

networks.

To minimize the impact of internet and remote HTTP server performance on the experiments, the ten WWW pages were copied from ten different WWW sites and stored on a local HTTP server. The number of images per page ranged from 1 to 14 images, for an average of 5 images. The total number of pages and images was 61.

When Netscape was used alone, 61 network connections were created. Each connection was used for a single page page or image. When Netscape was used with the Rover HTTP proxy, 10 network connections were created. One connection was used to transmit each page and its associated images.

Netscape by itself transmitted 36.7 KBytes and received 347 KBytes. Netscape in conjunction with the Rover HTTP proxy transmitted 26.3 KBytes (1.40:1) and received 291 KBytes (1.19:1). The data received by the proxy consisted of 41 KBytes of overhead and 250 KBytes of compressed data representing 348 KBytes of uncompressed data. The HTML portion of the pages accounted for 72 KBytes (28 KBytes compressed) and had a compression ratio of 2.6:1. The majority of the data consisted of images, which were far less compressible using the default Rover compression. Application-specific image compression techniques can offer up to two orders of magnitude of compression [24]. Using techniques such as delta encoding for communication between the client and server proxies would also yield significant performance benefits [57].

Using the proxy, the number of network connections is a function of the number of pages and not the number of pages and inlined images. For slower networks, opening a network connection is an expensive operation; thus, minimizing the number of connections is important.

Figure 6-5 shows the completion times for each network. On the faster networks (Ethernet, WaveLAN, 128 Kbit/s ISDN), Rover is slower because the requests are processed by two different machines (the web server and the server-side proxy). However, on the slower networks, Rover yields up to a 31% improvement in performance. As noted above, using application-specific compression would yield even better performance.

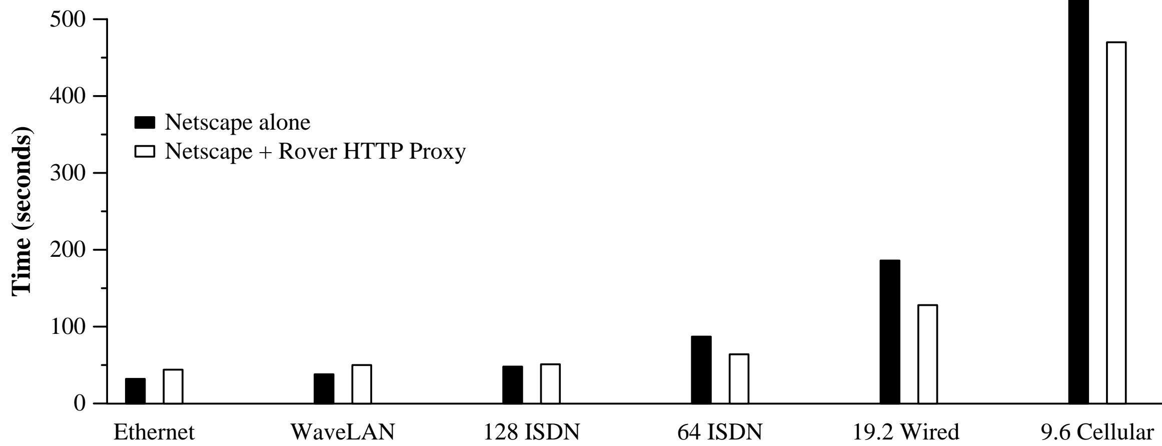


Figure 6-5: Time in seconds to fetch/display 10 WWW pages using Netscape alone and with the Rover HTTP proxy.

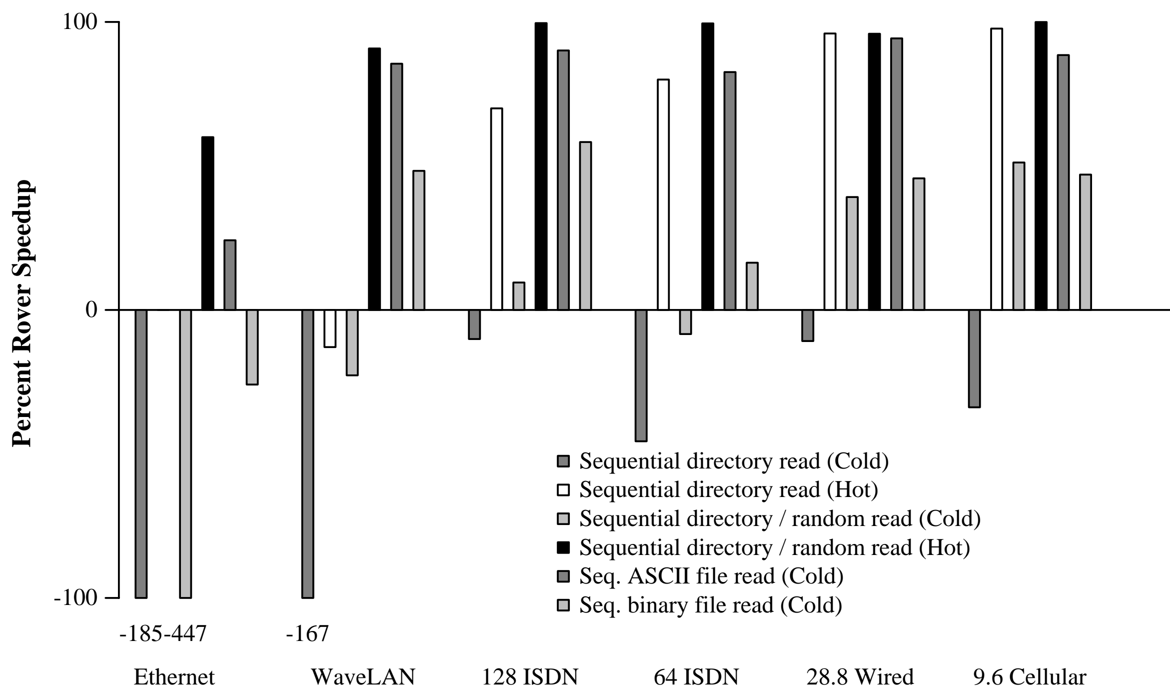


Figure 6-6: Speedup (or slowdown) of the Rover File System (RFS) Proxy over the Network File System (NFS).

It is important to note that the experiments do not reflect the "click-ahead" nature of the Netscape+Rover HTTP proxy application, which allows the user to browse the loaded pages while waiting for additional pages to load.

6.5.1 Rover File System Proxy

The performance of the Rover File System proxy was compared against the Network File System version 2.3 (NFS). The measurements were made with a remote file system on the Rover server containing 6.4 MB in 155 files, 15 directories, and no symbolic links. Both the Rover client and server used asynchronous log flushing. Each experiment was first performed with all caches (RFS client, Rover Access Manager, NFS client, and disk buffer) empty. The same experiment was then immediately repeated to measure hot cache performance. RFS was configured to mark all the data as read only; as a result, there was no additional network traffic while measuring RFS hot cache performance. Standard NFS file and directory caching was enabled. The experiments measured the speedup or slowdown of RFS relative to NFS: ($\frac{NFS-RFS}{NFS} \times 100$).

The sequential directory read performance was measured using the UNIX `find` command. Sequential directory read & random file read performance was measured using the UNIX `find` and `fgrep` commands. ASCII & binary sequential file read performance was measured using the UNIX `cat` command. Figure 6-6 presents the results for these experiments. Overall, RFS outperforms NFS over slower networks.

With an empty cache, RFS has a higher latency for sequential directory reads, however, with a hot cache, relative performance of sequential directory reads is comparable or significantly better (*e.g.,* RFS execution time over cellular networks is 2.3% of NFS execution time). The reason for better NFS performance is the overheads associated with: the user-level RFS and Rover implementations, QRPC, and QRPC over TCP. In contrast, the NFS client resides entirely in the kernel and uses UDP.

When file data is included (sequential directory read & random file read), RFS performance over slower networks is significantly better than NFS performance (*e.g.,* RFS cold cache execution time over cellular networks is less than half of NFS execution time). RFS performs better over slower networks due to its automatic header and data compression. On faster networks, the overhead associated compression is greater than the latency of sending uncompressed data.

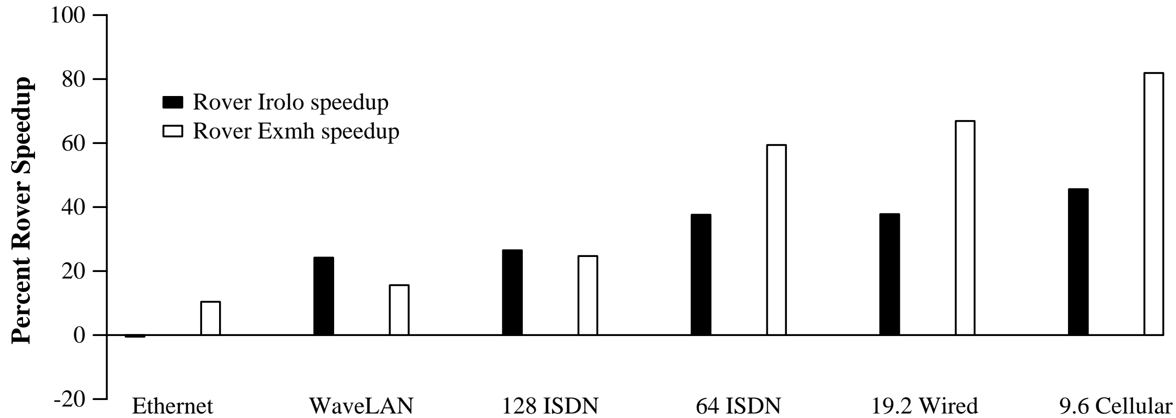


Figure 6-7: Speedup (or slowdown) of Rover mobile-adaptive versions of applications over the original X11-based applications when performing common tasks.

Rover's automatic compression offers significant performance benefits for cold cache ASCII file read performance, while Rover's use of TCP offers performance gains for cold cache binary file read performance. NFS's kernel-based implementation offers slightly better warm cache performance than RFS. For example, the worst RFS performance is for a warm cache 1 megabyte ASCII file read over Ethernet: 0.04 for NFS versus 0.96 milliseconds for RFS.

Overall, the positive results can be explained by Rover's automatic data compression and the caching policies employed by Rover and RFS. Both techniques are designed for the high-latency, low-bandwidth communication found in mobile environments. As mentioned earlier, the negative results are due to the overheads associated with: the user-level RFS and Rover implementations, QRPC, and QRPC over TCP. The positive results are due to the RFS design and the design of the Rover toolkit. Both the positive and the negative results are partly the consequences of design decisions that favored ease of development and portability over performance.

6.6 Mobile-Adaptive Application Performance

This section presents the performance benefits of caching RDOs and a comparison between mobile-transparent applications and mobile-adaptive applications running on both high-bandwidth, low-latency and low-bandwidth, high-latency networks.

To measure the performance benefits of the complete Rover system for mobile-adaptive applications, the performance of Rover Exmh, and Rover Irolo was compared against their unmodified X11-based counterparts, Exmh, and Irolo. For each application, a workload representative of a typical user's actions was performed and the time to complete the task was measured. The tasks were: reading eight MIME E-mail messages and browsing fifty rolodex entries.

To keep the measurements representative, the times did not include the cost of starting the application and loading the data required for the task. Excluding these times is acceptable because the system is typically used in this manner: the application is started and the data are loaded over a fast network and then the application is used repeatedly over a slow network (or without any network connectivity). Each task was repeated on each of the six network options.

For the X11-based applications, the SSH (Secure Shell) Remote Login Program, `ssh` [88] program was used to forward X11 connections over each of the network options. Ssh provides applications with authenticated, encrypted, and compressed network connections. The experiments used a compression level equivalent to GNU's *gzip* at level 6 compression. Without compression, the applications are nearly unusable over the slower networks.

Figure 6-7 presents the percent of speedup of the Rover version of each application over the original X11-based application ($\frac{X11-Rover}{X11} \times 100$). For the fastest network, Ethernet, Rover performance is comparable or better than the original application. For all of the other networks, Rover application performance is consistently better (ranging from a 16% performance gain on WaveLAN to an 82% performance gain on cellular dial-up). These results are especially encouraging, since they represent the target environment for Rover.

When no network is present, it is not possible to use the original X11-based applications. The Rover applications, however, show no change in performance as long as the application data are locally cached.

The amount of data sent and received by the Rover versions of the applications is significantly less than the amount sent by the original X11-based applications.

Storage media	Volatile	Tracing only	Stable logging	
			Async.	Sync.
None	0.038	0.040	—	—
Disk	—	—	0.099	10

Table 6.6: Approximate times in milliseconds to execute each iteration of the example counter code in Figure 4-2 at the server.

- Irolo sent 113 KBytes and received 133 KBytes versus Rover Irolo, which sent 20.6 KBytes (5.49:1) and received 16.7 KBytes (7.96:1).
- Exmh sent 517 KBytes and received 247 KBytes versus Rover Exmh, which sent 35.8 KBytes (14.4:1) and received 7.95 KBytes (31.1:1).

What the numbers fail convey is the extreme sluggishness of the user interface when using slower (*e.g.,* cellular) links without Rover. Scrolling and refreshing operations are extremely slow. Pressing buttons and selecting text are very difficult operations to perform because of the lag between mouse clicks and display updates. With Rover, the user sees the same excellent GUI performance across a range of networks that varies by nearly three orders of magnitude in both bandwidth and latency.

6.7 Fault-Tolerant Applications

Low-level and system level experiments were performed to provide a better understanding of the costs of using the language extensions for server-side reliable execution. The low-level experiments measured the individual costs for using stable variables. The system-level experiments used a reliable proxy (the server portion of the Rover Web Browser Proxy) and a reliable application (the file search application from Section 5.3.6) to measure the effects of the reliable execution extensions on end-to-end application performance. Implementation details for the application and proxy can be found in Section 5.3.3.

6.7.1 Stable Variables

The cost of using stable variables has two components: the Tcl write tracing overhead and the stable logging of changes. These costs were measured using the simple counter code, from Figure 4-2, at the server and used a disk for stable logging. While the code does not reflect the likely behavior of most applications, it provides a baseline for the overhead associated with using stable variables. The results are summarized in Table 6.6.

At the server, the extra cost to trace a write to a Tcl variable is negligible. Adding stable variable logging with asynchronous log buffer flushing doubles the cost versus a volatile variable. Using synchronous logging to a disk, the cost per iteration is two orders of magnitude higher than when a volatile variable is used.

The substantial execution time difference between volatile variables and synchronously logged stable variables suggests that there should be a clear distinction between stable and volatile variables. By distinguishing between stable and volatile variables, only those applications that choose to provide added reliability will incur the associated added logging costs. Performance of non-fault-tolerant applications will be unaffected.

Applications should also be allowed to choose between synchronous and asynchronous log flushing so that they have control over the reliability/performance tradeoff.

6.7.2 Rover Web Browser proxy

The proxy experiments measured the time to fetch the Rover project home page and its inlined images, http://www.rover.lcs.mit.edu/ . To help reduce the variability of the results, the experiments used copies of the home page stored on a private httpd server. The page consists 62.5 KBytes in three elements: two pictures (54 KBytes) and the HTML text (8.5 KBytes). The size of all the objects with Rover compression is 55.4 KBytes.

The proxy experiment measured the performance of the server portion of the proxy

Stable storage	None	Volatile only	Stable logging	
			Async.	Sync.
None	600±50	610±80	—	—
Disk	—	—	600±50	600±60

Table 6.7: Time in milliseconds (with 90% confidence intervals) to execute the server portion of the Rover Web Browser proxy while fetching the Rover project home page and its two inlined images.

— the time from when the Rover server dispatched the proxy module to when the module returned. The times include the transport and processing overheads of the local httpd server and some of the transport overhead of sending the data back to the client.

The results presented in Table 6.7 show the execution times when the proxy was implemented without hash tables, with volatile hash tables, and with logged stable hash tables with asynchronously and synchronously flushed buffers. As can be seen from the table, the performance is almost identical in all cases. Other costs (httpd server overhead and transport cost) and their variability are dominating the server execution times. Overall, the results show that the cost of stable hash tables is an acceptably small component of the execution time of the proxy.

What Table 6.7 does not show is the primary benefit to the proxy of using a stable hash table. After a server failure, the stable hash table allows the server to avoid sending duplicate objects to clients, conserving potentially expensive and/or limited bandwidth.

6.7.3 Text file search

The file search application was used to scan 1.9 MBytes in 328 files in 3 directories both without faults and with a single fault injected after 75% of the files were searched. The server-side execution times are presented in Table 6.8. The corresponding model for the execution times is presented in Figure 6-8.

The average time to search a file when stable variables are not used, $\bar{t}_{\text{fileWork}}$, is

Failures	Volatile	Tracing only	Stable logging	
			Async.	**Sync.**
None	10.1±0.057	10.1±0.053	10.2±0.050	14.9±0.046
Single (75%)	19.4±0.048	—	11.7±0.050	15.1±0.038

Table 6.8: Times in seconds to execute the server-side portion of the file search application. For the single failure case, the fault was injected after 75% of the files were searched.

$$
\begin{aligned}
\bar{t}_{\text{fileSearch}} &= \bar{t}_{\text{fileWork}} + \bar{t}_{\text{traceOver}} + \bar{t}_{\text{logOver}} \\
t_{\text{Search}} &= \bar{t}_{\text{fileSearch}} \times n_{\text{SearchedFiles}} + n_{\text{Failures}} \times (\bar{t}_{\text{fileSearch}} \times 0.5 + t_{recover}) \\
t_{\text{recover}} &= t_{\text{forkExec}} + t_{\text{readLog}}
\end{aligned}
$$

Figure 6-8: Time to complete the server-side portion of the text search application.

30.9 milliseconds (6,180,000 cycles).

When variable writes are traced, 350 writes are traced. The per-write tracing time, $\bar{t}_{\text{traceOver}}$, from Table 6.6 is 0.040 milliseconds (8,000 cycles), yielding a total tracing overhead of 14 milliseconds. As the results show, tracing has a negligible effect on performance.

When stable variables are used, the 350 writes log an average of 250 bytes (a total of 81,275 bytes) at a per-write logging cost, $\bar{t}_{\text{logOver}}$, of 0.12 milliseconds (24,000 cycles) using asynchronous logging or 14.7 milliseconds (2,940,000 cycles) using synchronous logging. In the failure-free cases, the performance using logged stable variables with asynchronously flushed buffers is nearly identical to the performance using volatile variables. Synchronous logging is two orders of magnitude slower than asynchronous logging; thus, synchronous logging yields a significant performance slowdown.

After a failure, the parent server forks and executes a new child server. This action, $t_{forkExec}$, takes 0.72 milliseconds (144,000 cycles). When stable variables are not used, the number of searched files, $n_{\text{SearchedFiles}}$, is incremented by the number of files and the times for t_{readLog}, $\bar{t}_{\text{traceOver}}$, and $\bar{t}_{\text{logOver}}$ are 0. The time to read the stable QRPC log is 12.6 ± 0.26 milliseconds (2,520,000 ± 2,472,000 cycles).

Failures	**Volatile**		**Stable logging**			
			Async.		**Sync.**	
	Actual	**Model**	**Actual**	**Model**	**Actual**	**Model**
None	10.1	10.1	10.2	10.2	14.9	15.0
Single (75%)	19.4	17.8	11.7	11.2	15.1	15.2

Table 6.9: Actual and model times in seconds to execute the server-side portion of the file search application.

When stable variables are used, $n_{\text{SearchedFiles}}$, is constant (equal to the number of files) regardless of the number of failures. A failure causes the loss of an average of half of one file's work. The time to restore the application's stable variables from the log is dependent upon on the type of logging. The time to read the stable QRPC log and restore the stable variables is: for asynchronous logging 1,003 $\pm$ 31 milliseconds (200,600,000 $\pm$ 6200000 cycles); for synchronous logging, 201 $\pm$ 13 milliseconds (40,200,000 $\pm$ 2,600,000 cycles). The asynchronous time is greater because the log read operations are blocked in the operating system by the pending log write operations.

Table 6.9 presents a comparison between the model's results and the actual results from experiments. All of the model results are within a few percent of the experimental results with the exception of the volatile, single failure case. In the volatile, single failure case, the difference is 8.3%. The reason for the difference is that the model assumes that an equal amount of work is done per file. In actuality, more work is done per file before 75% of the files have been searched than afterwards. The model predicts that the application spends 7,600 milliseconds searching 75% of the files, while the actual amount of work done is 8,000. Compensating for this difference yields a time of 18.2 milliseconds and a difference of 6.3% (comparable to the differences for other results).

Asynchronously logged stable variables perform as well as volatile variables; thus, in the presence of failures, stable variable application performance will be better than volatile application performance. For example, in the above experiment, when a single fault is introduced after 75% of the files have been searched, the execution time of

Amount of Logging	Asynchronous Logging	Synchronous Logging
Normal (250 bytes)	10.1±0.057	14.9±0.046
2x logging (500 bytes)	10.2±0.057	15.2±0.049
4x logging (1,000 bytes)	10.3±0.057	15.6±0.064

Table 6.10: The effects of doubling and quadrupling the amount of data logged by the text search application.

the volatile variable version of the application is nearly doubled. The asynchronously logged version of the application shows only a 14.7% degradation in performance.

Synchronously logged stable variables offer lower failure free performance (*e.g.,* in the above experiment, the synchronously logged version shows only a 1.3% degradation in performance). Thus, in the presence of failures, there is a crossover point where synchronously logged stable variables perform better than the volatile application. Combining the models for synchronously logged variables and volatile variables and solving for the crossover point yields a crossover point of a failure after 50% of the files have been searched (163 files).

Using stable variables allows an application developer to construct a simple, accurate model of an application's performance. The developer can use the model to decide whether or not stable variables are needed for an application.

To explore the effects of the amount of data logged on performance, the server was modified to log writes multiple times. Table 6.10 shows the effects on the execution time of the search application as a function of the number of times each write was logged. The results show that when asynchronous logging is used, the effect of the amount of data written has a minimal effect on performance (less than a 2% performance degradation when four times as much data is logged). The effect on synchronous logging is more significant than for asynchronous logging, but is still minimal (less than 5% performance degradation when four times as much data is logged).

The text search application demonstrates that stable variables are a simple, efficient mechanism for providing applications with reliable storage. Using asynchronous

logging yields performance equal to volatile applications and protects applications against their own and server failures, but not against operating system or hardware failures. Synchronous logging can be used for a greater degree of protection against faults with a somewhat higher performance penalty.

6.8 Discussion

The experimental results presented in this section confirm the hypotheses listed at the beginning of this chapter.

The low-level benchmarks show that the batching and compression of multiple requests yields significant performance benefits for slower networks — increased throughput and reduced average latency. In addition, stable variables are easy to use, offer significant performance gains in the presence of failures, and, with asynchronous logging, yield performance comparable to that for volatile applications.

The system-level experiments show that unmodified mobile transparent applications gain usability and performance benefits from the Rover toolkit. For mobile-adaptive applications, the usability and performance gains are more substantial. Rover allows users to see the same excellent GUI performance across a range of networks that varies by nearly three orders of magnitude in both bandwidth and latency.

Finally, stable variables provide a simple, efficient mechanism for building reliable applications that perform as well as volatile applications.

Chapter 7

Future Work and Conclusion

This thesis has shown that the integration of relocatable dynamic objects and queued remote procedure calls in the Rover toolkit provides a powerful basis for building mobile-transparent and mobile-adaptive applications. It is quite easy to adapt applications to use these Rover facilities, resulting in applications that are far less dependent on high-performance communication connectivity.

7.1 Future Work

This thesis proposes a general-purpose architecture for building applications for mobile environments. The ideas in the thesis are made concrete through the reference implementation, the Rover toolkit.

Some of the interesting areas for additional and future research are:

1. More powerful mobile code security. There already are several active research efforts ongoing in this area.

    - Safely execution of mobile code from insecure clients, servers, and environments.
    - Authenticating clients to servers, servers to clients, and users to servers.
    - Providing access control mechanisms for server-side databases.

2. Additional request batching techniques. Using application-specified batching commands as informational suggestions or overrides for the existing batching heuristic.

3. Client-side application failure recovery. Providing mechanisms for preserving the state of and automating the recovery of client-side applications.

4. Cache management tools and policies. Providing tools to help users manage client-side caches.

5. More complete reference implementation. Providing a reference implementation that implements the features and functionality that are not implemented in the current reference implementation (as discussed in Section 4.1).

7.2 Conclusion

The results of this thesis can he summarized as follows:

1. QRPC provides a mechanism that works well in intermittently connected environments. QRPC's reliable delivery transport frees application developers from worrying about communication failures. QRPC performance is acceptable even if every RPC is stored in stable logs at clients and servers before being processed. For lower-bandwidth networks, the overhead of using stable logs is dwarfed by the underlying communication costs.

2. Using a queued communication model enables QRPCs to be scheduled, batched, and compressed for more efficient use of communication channels and increased network performance. Using batching and compression minimizes the number of roundtrip messages and network connections used by QRPCs, reduces the average latency for a request, and minimizes the total amount of data sent over communication channels.

3. Using RDOs allows mobile-adaptive applications to migrate functionality dynamically to either side of a slow network connection minimizing the amount of

data transiting the network. Caching RDOs reduces latency and bandwidth consumption. Interface functionality can run at full speed on a mobile host, while large data manipulations may be performed on the well-connected server. RDOs also allow computation migration decision based based upon available computation resources. Computation can be offloaded from a temporarily computationally underpowered client or a temporarily overloaded server. Thus, computation migration aids in making Rover scalable to large numbers of clients.

4. Together, RDOs and QRPCs allow application developers to decouple many user-observable delays from network latencies. The result is excellent graphical user interface performance over network technologies that vary by three orders of magnitude in bandwidth and latency.

5. Experimental results demonstrate these Rover's fault-tolerance features impose low overhead. The low overhead is especially true in the low-bandwidth, high-latency environment typical of mobile clients.

6. The toolkit also helps application developers protect long-running applications — the applications that are the most likely to be affected by transient software and hardware faults. The fault-tolerant features described in this thesis provide a powerful, but easy to use, tool for building mobile-adaptive applications that are reliable in the presence of such faults.

7. The language extensions for stable variables provide a natural and simple way for programmers to maintain stable state, dramatically reducing the recovery time after a failure. The changes necessary to add fault-tolerant support to an application are usually minimal. However, the potential high cost associated with using stable variables means that their use should be determined by the application programmer and not by the system. Furthermore, giving application programmers control over the use of stable or volatile variables and synchronous or asynchronous log buffer flushing, allows them to make the appropriate reliability/performance tradeoffs.

Overall, the Rover toolkit allows application developers to rapidly build applications and proxies that yield good application performance. Measurements of end-to-end mobile application performance shows that mobile-transparent and mobile-adaptive applications perform significantly better than their stationary counterparts. For example, Rover offers a 12% performance improvement for the mobile-transparent Netscape application. Mobile-adaptive applications show performance improvements of up to a factor of 5.5 over slow networks.

Developing applications for the mobile environment is a complicated process. The volatility of the environment makes achieving correct operation difficult. The Rover toolkit simplifies the development process and helps ensure application correctness by providing developers with the tools that they need.

Bibliography

[1] Adobe Systems. *Programming the Display PostScript System with X*. Addison-Wesley Pub. Co., Reading, Massachusetts, 1993.

[2] M. Ahamad, P. Dasgupta, and R.J. Leblanc. Fault-tolerant atomic computations in an object-based distributed system. *Distributed Computing*, 4:69–80, 1990.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley Publishing Co., Reading, Massachusetts, 1996.

[4] A. Avizienis. Software fault tolerance. In *Proc. 1989 IFIP World Computer Conference*, pages 491–497, Geneva, 1989. IFIP Press.

[5] M.G. Baker. Changing communication environments in MosquitoNet. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 64–68, Santa Cruz, California, December 1994.

[6] J. Bartlett. W4—the Wireless World-Wide Web. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 176–178, Santa Cruz, California, December 1994.

[7] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[8] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[9] N. S. Borenstein. EMail with a mind of its own: The Safe-Tcl language for enabled mail. In *IFIP Transactions C*, pages 389–415, Barcelona, Spain, June 1994.

[10] M. H. Brown and R. A. Schillner. DeckScape: An experimental web browser. Technical Report 135a, Digital Equipment Corporation Systems Research Center, March 1995.

[11] Henry Chang, Carl Tait, Norman Cohen, Moshe Shapiro, Steve Mastrianni, Rick Floyd, Barron Housel, and David Lindquist. Web browsing in a wireless environment: Disconnected and asynchronous operation in ARTour Web Express. In *Proc. of the Third Annual ACM/IEEE International Conference on Mobile Computing and Networking*, Budapest, Hungary, September 1997.

[12] Oracle Corporation. Oracle mobile agents: Technical product summary, August 1995.

[13] Constantine Cristakos. The Rover NNTP proxy. Advanced Undergraduate Project, Massachusetts Institute of Technology, June 1996.

[14] D. H. Crocker. *Standard for the Format of ARPA Internet Text Messages.* Internet RFC 822, August 1982.

[15] N. Davies, G. Blair, K. Cheverst, and A. Friday. Supporting adaptive services in a heterogeneous mobile environment. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, December 1994.

[16] A. F. deLespinasse. Rover Mosaic: E-mail communication for a full-function web browser. Master's thesis, Massachusetts Institute of Technology, June 1995.

[17] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 2–7, Santa Cruz, California, December 1994.

[18] L. Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. Internet RFC 1951, May 1996.

[19] L. Peter Deutsch. *GZIP File Format Specification version 4.3*. Internet RFC 1952, May 1996.

[20] L. Peter Deutsch and Jean-Loup Gailly. *ZLIB Compressed Data Format Specification version 3.3*. Internet RFC 1950, May 1996.

[21] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *First Symposium on Operating Systems Design and Implementation*, pages 25–37, Monterey, California, November 1994.

[22] F. Douglis and J. Ousterhout. Process migration in the Sprite operating system. In *Proc. of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987.

[23] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. *HyperText Transfer Protocol – HTTP/1.1*. IETF HTTP Working Group Draft 08, July 1997.

[24] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proc. of the Seventh Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 160–173, Cambridge, Massachusetts, October 1996.

[25] D. K. Gifford and J. E. Donahue. Coordinating independent atomic actions. In *Proc. of the Spring COMPCON Conference*, pages 92–92, San Francisco, California, February 1985.

[26] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, March 1988.

[27] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 1996 SIGMOD Conference*, Montreal, Quebec, Canada, June 1996.

[28] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufmann Publishers, Inc., San Mateo, California, 1993.

[29] J. Gray and D. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, 1991.

[30] R. Gruber, M. F. Kaashoek, B. Liskov, and L. Shira. Disconnected operation in the Thor object-oriented database system. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 51–56, Santa Cruz, California, December 1994.

[31] P. Honeyman, L. Huston, J. Rees, and D. Bachmann. The LITTLE WORK project. In *Proc. of the 3rd Workshop on Workstations Operating Systems*, Key Biscayne, FL, April 1992.

[32] H. Houh, C. Lindblad, and D. Wetherall. Active pages. In *Proc. of the First International World-Wide Web Conference*, pages 265–270, Geneva, May 1994.

[33] L. Huston and P. Honeyman. Partially connected operation. In *Proc. of the Second USENIX Symposium on Mobile & Location-Independent Computing*, pages 91–97, Ann Arbor, MI, April 1995.

[34] L. B. Huston and P. Honeyman. Disconnected operation for AFS. In *Proc. of the First USENIX Symposium on Mobile & Location-Independent Computing*, pages 1–10, Cambridge, Massachusetts, August 1993.

[35] Information Sciences Institute. *Transmission Control Protocol: DARPA Internet Program Protocol Specification*. Internet RFC 793, September 1981.

[36] V. Jacobson. *Compressing TCP/IP Headers for Low-Speed Serial Links.* Internet RFC 1144, February 1990.

[37] D. Johansen, R. van Renesse, and F. B. Schneider. Operating system support for mobile agents. In *Proc. of the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Washington, May 1995.

[38] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, pages 156–171, Copper Mountain Resort, Colorado, December 1995.

[39] A. D. Joseph and M. F. Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. In *Proc. of the Second International Conference on Mobile Computing and Networking (MOBICOM '96)*, pages 117–129, Rye, NY, November 1996.

[40] A. D. Joseph and M. F. Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. *Wireless Networks*, 1997. To appear.

[41] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997.

[42] F. Kaashoek, T. Pinckney, and J. A. Tauber. Dynamic documents: Mobile wireless access to the WWW. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 179–184, Santa Cruz, California, December 1994.

[43] R. H. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1:6–17, 1994.

[44] R. Katz *et. al.*. The Bay Area Research Wireless Access Network (BARWAN). In *Proc. of the Spring COMPCON Conference*, February 1996.

[45] L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. Presented at the *Second Conference on Computer-Supported Cooperative Work*, Portland, OR, September 1988.

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

[47] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10:3–25, February 1992.

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

[49] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[50] J. Landay. User interface issues in mobile computing. In *Proc. of the Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, pages 40–47, October 1993.

[51] M.T. Le, F. Burghardt, S. Seshan, and J. Rabaey. InfoNet: the networking infrastructure of InfoPad. In *Proc. of the Spring COMPCON Conference*, pages 163–168, 1995.

[52] A. K. Lenstra and M. S. Manasse. Factoring by electronic mail. In *Advances in Cryptology — Eurocrypt '89*, pages 355–371, Berlin, 1989. Springer-Verlag.

[53] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proc. of the Eleventh Symposium on Operating Systems Principles (SOSP)*, Austin, Texas, December 1987.

[54] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*. Morgan-Kaufmann Publishers, Inc., San Mateo, California, 1993.

[55] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 260–267, Atlanta, GA, June 1988.

[56] J.C. Mallery. A Common LISP hypermedia server. In *Proc. of the First International World-Wide Web Conference*, pages 239–247, Geneva, May 1994.

[57] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachender Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proc. of the 1997 SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 181–196, Palais des Festivals, Canne, France, September 1997.

[58] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, pages 143–155, Copper Mountain Resort, Colorado, December 1995.

[59] S. Narayanaswamy, *et. al.*. Application and network support for InfoPad. *IEEE Personal Communications*, 3(2):4–17, April 1996.

[60] National Center for Supercomputing Applications. *Common Gateway Interface.* University of Illinois in Urbana-Champaign, 1995.

[61] National Center for Supercomputing Applications. *Mosaic.* University of Illinois in Urbana-Champaign, 1995.

[62] Netscape Communications Corporation. *Netscape Navigator.* Mountain View, California, 1995.

[63] Henrik F. Nielson, James Gettys, Anselm Baird-Smith, Eric Prud'hommeaux, Håkon Wium Lie, and Chris Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proc. of the 1997 SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 155–166, Palais des Festivals, Canne, France, September 1997.

[64] B. D. Noble, M. Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. In *Proc. of the Second USENIX Symposium on Mobile & Location-Independent Computing*, Ann Arbor, MI, April 1995.

[65] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, , and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proc. of the Sixteenth Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, October 1997.

[66] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Massachusetts, 1994.

[67] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP. In *Proc. of the Ninth Symposium on Operating Systems Principles (SOSP)*, pages 110–119, October 1983.

[68] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *Proc. of the USENIX Summer 1994 Technical Conference*, pages 183–195, Boston, Massachusetts, 1994.

[69] D. Riecken, editor. *Intelligent Agents*. Communications of the ACM, 37(7), 1994.

[70] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–28, November 1984.

[71] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experience with disconnected operation in a mobile environment. In *Proc. of the First USENIX Symposium on Mobile & Location-Independent Computing*, pages 11–28, Cambridge, Massachusetts, August 1993.

[72] J. M. Smith. A survey of process migration mechanisms. *Operating Systems Review*, 22(3):28–40, July 1988.

[73] Sun Microsystems Corporation. *Remote Method Invocation for Java*. http://-chatsubo.javasoft.com/current/rmi/index.html, July 1996.

[74] J. A. Tauber. Issues in building mobile-aware applications with the Rover toolkit. Master's thesis, Massachusetts Institute of Technology, June 1996.

[75] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. of the 1994 Symposium on Parallel and Distributed Information Systems*, pages 140–149, September 1994.

[76] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. Technical Report CSL-94-9, Xerox Palo Alto Research Center, July 1994.

[77] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in a weakly connected replicated storage system. In *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, pages 172–183, Copper Mountain Resort, Colorado, December 1995.

[78] M. Theimer, A. Demers, K. Petersen, M. Spreitzer, D. Terry, and B. Welch. Dealing with tentative data values in disconnected work groups. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 192–195, Santa Cruz, California, December 1994.

[79] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the V-System. In *Proc. of the Tenth Symposium on Operating Systems Principles (SOSP)*, pages 2–12, Orcas Island, Washington, December 1985.

[80] R. Van Renesse, T. Hickey, and K. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report TR 94-1442, Department of Computer Science, Cornell University, Ithica, New York, August 1994.

[81] J. Vittal. Active message processing: Messages as messengers. In *Proc. of IFIP TC-6 International Symposium on Computer Message Systems*, pages 175–195, Ottawa, Canada, April 1981.

[82] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proc. of the Ninth Symposium on Operating Systems Principles (SOSP)*, pages 49–70, Bretton Woods, NH, 1983.

[83] Dan S. Wallach, Dirk Balfanz, Drew Dean, , and Edward W. Felten. Extensible security architecture for Java. In *Proc. of the Sixteenth Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, October 1997.

[84] T. Watson. Application design for wireless computing. In *Proc. of the Workshop on Mobile Computing Systems and Applications*, pages 91–94, Santa Cruz, California, December 1994.

[85] T. Watson and B. Bershad. Local area mobile computing on stock hardware and mostly stock software. In *Proc. of the First USENIX Symposium on Mobile & Location-Independent Computing*, pages 109–116, Cambridge, Massachusetts, August 1993.

[86] W. Weihl and B. Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

[87] J. E. White. Telescript technology: The foundation for the electronic marketplace, 1994.

[88] Tatu Ylönen. SSH (Secure Shell) Remote Login Program, 1997. http://-www.cs.hut.fi/ssh.