# Issues in Building Mobile-Aware Applications with the Rover Toolkit

by

Joshua A. Tauber

B.S., Computer Science (1991)
B.A., Government (1991)
Cornell University

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996



Author ..................................................................
Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by ..............................................................
M. Frans Kaashoek
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ..............................................................
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

**Issues in Building Mobile-Aware Applications with the Rover Toolkit**

by

Joshua A. Tauber

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

## Abstract

To excel under the harsh conditions of a mobile computing environment, applications must be designed to be aware of and take an active part in mitigating those conditions. The Rover Mobile Application Toolkit supports a set of programming and communication abstractions that enable mobile-aware applications to cooperate actively with the user and the underlying Rover runtime system to achieve increased availability, concurrency, resource allocation efficiency, fault tolerance, consistency, and adaptation. Analysis and experimental evaluation of a calendar tool redesigned to be mobile-aware demonstrate that such application-level control allows correct operation, increases interactive performance by up to a factor of 1000, and dramatically reduces network utilization, under intermittently connected conditions.

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor of Computer Science and Engineering

## Acknowledgments

Frans Kaashoek's imprint appears not only on the cover of this thesis but throughout its text, structure, and concepts. I cannot emphasize enough the unwavering nature of Frans's support and positive attitude. Frans always convinced me this thesis was possible even when I did not believe it myself.

Anthony Joseph conceived and built most of Rover. Selflessly, he spent the wee hours of many a morning debugging Rover and Webcal with me. He even came through at the last minute to re-run experiments for me — from New York. For a year and a half he has provided a source of ideas and techniques to keep me going when I had none and he happily shot down my ideas and techniques when I had too many. Thank you Anthony.

The last minute crew of Emmett Witchel, Eddie Kohler, and Greg Ganger came through when all others deserted me. They read drafts and helped me tame the numbers into tables and graphs. If you can understand the numbers in this thesis, thank them. If not, blame me. I also want to thank Emmett for putting up with a neurotic, hygienically-challenged officemate for the last week (or was that a month ?) and for having such an interesting name.

My cheering section of Helen Snively, Howard Ramseur, and Geoff Hyatt encouraged me, praised me, cajoled me, questioned me, even lied to me in order to get me to write. Hey, guys, it worked!

To Ken Birman, I will always be grateful for those fateful words of support "Go to grad school, Josh." To Sam Toueg, the best teacher I ever had, I owe a double debt of thanks. He inspired me to emulate him as an educator and invited me to join him as a researcher.

I'd like to thank the whole of the Parallel and Distributed Operating Systems Group, past and present, for making my experience at MIT the fun frolic it has been. You are always enlightening and entertaining. Most especially I want to thank, Neena Lyall who makes everything run smoothly, even when I do stupid things.

Thank you Mitchell Charity for your endless discussions of semantics (both related to this thesis and not).

By the way, this work was supported in part by the Advanced Research Projects Agency under contract DABT63-95-C-005, by an NSF National Young Investigator Award, and by grants from IBM and AT&T. I'm sure you wanted to know that.

Thanks Mom and Dad — for everything.

Thank you Stephanie for putting up with all the late nights, early mornings, piles of unwashed dishes, foul moods, short tempers, and other symptoms of chronic thesitis. You are my toughest proof reader and my most supportive fan. I don't tell you often enough how much both those qualities mean to me. Thanks for being there, always.

To Shulamit. Now, I've caught up.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Application designers for mobile "roving" computers face a unique set of communication and data integrity constraints that are absent in traditional workstation settings. For example, although mobile communication infrastructures are becoming more common, network bandwidth in mobile environments is often limited, and at times, unavailable. Thus, mobile applications should minimize dependence upon data obtained over such limited, unreliable networks. On the other hand, mobile computers are able to store much less data than their heavier, stationary counterparts. In addition, a portable computer may be dropped, sat on, spilled on, or left next to an unshielded speaker, resulting in the loss of stored data. Thus, mobile applications should minimize dependence on vulnerable data stored in limited, unreliable resources on a mobile host. Mobile application designers have a common need for system facilities that minimize dependence upon continuous connectivity, that provide tools to optimize the utilization of network bandwidth, and that minimize dependence on data stored uniquely on the mobile computer. The Rover Toolkit provides mobile application developers with a set of tools to isolate mobile applications from the limitations of mobile communication and computation systems.

## 1.1 Design Goals

Mobile computing system designers typically have one or more of the following goals in mind while building applications: high availability, high concurrency, resource allocation efficiency, fault tolerance, consistency, and adaptation. Many of these goals are similar to goals in general distributed computing. However, mobile computing emphasizes different aspects of these goals. Certain goals, which were merely laudable achievements in a stationary distributed system, become critically necessary in a mobile system.

In mobile computing, *high availability* means the ability to use the application on the mobile host even while disconnected. In particular, the user should receive the same high level of responsiveness and performance in an intermittently-connected environment as in a fully-connected one. *High concurrency* means simultaneously accomplishing work at several points in the system. One form of concurrency is the overlap of computation at the client, message transit time and computation at the server. This client-server concurrency hides the latency of slow networks and overloaded servers. A second form of concurrency is client-client concurrency in which multiple clients may access and update the same data at the same time — possibly without being connected to each other or the server. Third, mobile computing requires *high resource allocation efficiency* for several critical resources: network bandwidth, processor cycles, stable storage, and energy. Each may be several orders of magnitude smaller or slower on the mobile host than on a typical stationary host. In addition, money becomes a resource to be allocated when network usage is not free. *Fault-tolerance,* or reliability, takes on a new importance in mobile computing. First, in an intermittent environment, network faults must be expected, and dealt with simply as another type of event. Second, the mean time to failure in a mobile environment is drastically lower than in a stationary one. Data stored on a mobile computer is extremely vulnerable to loss. Therefore, mobile-computing systems must be resilient to these losses. *Adaptation* means mobile applications need to take advantage of the changing availability of resources in the mobile environment. It means moving an application's workload between servers and clients, both to address computationally under-powered clients and overloaded servers. It also means permitting applications to be involved in connectivity related decisions. *Consistency* in mobile computing means the same as in typical distributed computing (i.e., an application's view of the data or objects in the system is independent of the application's location) but it is harder to maintain because hosts may be disconnected from each other for extended periods of time. To avoid blocking during disconnection, there is an increased emphasis on optimistic schemes.

## 1.2 The Argument for Mobile-Aware Computing

Fundamentally, there are two approaches to building mobile-computing applications: mobile-aware and mobile-transparent. Previous mobile-computing systems, such as Coda [26] and Little Work [16], have promoted the mobile-transparent approach. The objective of the mobile-transparent approach is to hide entirely the mobile characteristics of the environment so that applications may be used without alteration. This is accomplished by creating a mobile-aware proxy for some service (in the above cases, the file system). The proxy runs on the mobile host and provides the standard service interface to the application while attempting to mitigate the effects of the mobile environment. The proxy on the mobile host cooperates with a mobile-aware server on a well-connected, stationary host. We have shown elsewhere that Rover can be used for either approach [21, 7]. The Rover Web browser proxy is an example of the mobile-transparent approach. The Rover exmh mail reader and the Rover Webcal calendar tool are examples of the mobile-aware approach.

Applying the end-to-end argument to mobile computing systems exposes that, while the mobile-transparent approach is appealing, it is fundamentally limited in its applicability. Saltzer *et. al.* state the "end-to-end argument" as follows:

> [Communication functionality] can completely and correctly be implemented only with the knowledge and help of the application standing at the endpoints of the communications system. Therefore, providing that questioned function as a feature of the communication system itself is not possible [41].

The functionality needed to create correct, well-performing applications in an intermittently-connected environment requires the cooperation of the application. For example, it is not possible to maintain an entirely consistent, well-performing, mobile file system without cooperation from the applications mutating the file system.

In fact, it is not possible to write a file system that guarantees data consistency without application cooperation in the stationary, non-distributed case. File systems allow applications to cooperate in sequences of read-modify-write operations or to use pessimistic locking to maintain consistency. Most applications choose to use some form of non-atomic read-modify-write semantics. In a non-mobile environment, this decision constitutes an informed choice. Applications elect to allow a window of vulnerability between read and writes. In the non-mobile environment, that choice works because, reads and writes are not far separated in time. In the mobile environment this choice is ill-informed. Reads and writes may be separated by days.

Consider an application writing records into a file shared among stationary and mobile hosts. During disconnection, the application on the mobile host inserts a new record. A file system proxy on the mobile host will note that the file has changed and save the last write to the file. Meanwhile, the application on a stationary host alters some other record in the file. Upon reconnection, the file system can note that (possibly) conflicting updates have occurred. However, the file system cannot resolve the conflict.

Coda recognizes this limitation and allows applications to cooperate with the file system through adjunct applications called application-specific resolvers (ASRs). However, ASRs alone are insufficient. In this 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 Coda file system proxy also changes the semantics of the file system in order to hide the condition of the underlying network. 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 timing and sequencing information have already *unrecoverably* lost that information before the ASR is ever invoked.

In contrast, a fully mobile-aware application can store not only the *value* of the file with the inserted record but the *operation* inserting the record into the file. Furthermore, the application need not rewrite the entire file. It only needs to log the operation to insert one record. Thus, the data that the mobile host must send over the slow, unreliable network to reconcile with the server is significantly decreased. Finally, since the application has more information about its own semantics, the application can design the operation to automatically resolve its own conflicts. It may even construct the resolution semantics "on-the-fly" in response to events in the mobile environment. Thus, a mobile-aware application that actively cooperates with the underlying system integrates a simplified model of consistency control with increased performance.

The end-to-end argument does not require that every application use an original, *ad hoc* approach to mobile-computing. On the contrary, it requires that the underlying communication and programming systems define an application programming interface that optimizes common cases and supports the transfer of appropriate information between the layers. Since mobile-computing applications share common design goals, they will need to share design features and techniques.

## 1.3 Rover: The Toolkit Approach

The Rover Toolkit provides exactly such a mobile-aware application programming interface. Rover allows mobile-aware applications to obtain information about the mobile-environment and act on it to to maintain consistency and enhance performance. Simultaneously, Rover optimizes common-case schemes for replication and communication in intermittently connected environments. The Rover Toolkit offers applications a distributed object system based on a client-server architecture. Clients are Rover applications that typically run on mobile hosts, but could run on stationary hosts as well. To improve reliability, servers typically run on stationary hosts and hold the long term state of the system. The Rover Toolkit consists of a library linked into all applications and runtime modules on client and server machines. Rover applications actively cooperate with the runtime system to *import* objects onto the local machine, *invoke* well-defined methods on those objects, *export* to servers logs of method invocations on those objects, and *reconcile* the client's copies of the objects with the server's.

The key task of the programmer in building an application with Rover is to define *relocatable dynamic objects* (RDOs) for the data types manipulated by the application and data transported between client and server. The programmer divides the program into portions that run on the client and portions that run on the server. The two parts communicate by means of *queued remote procedure calls* (QRPC). The programmer then defines methods that update objects, including code for conflict detection and resolution. All code making up the application and data touched by the application can be cast into RDOs.

To illustrate the issues that mobile-aware applications designers face, this thesis studies one application in detail: Webcal. Webcal is a mobile-aware version of the Ical calendar tool. Webcal was built with the Rover Toolkit for the purpose of highlighting the differences between mobile-aware and mobile-unaware applications. In particular, this thesis illustrates how application-level control is necessary for correct operation of Webcal while also enhancing its interactive performance. Other applications built with Rover provide additional evidence of the usefulness of this approach.

## 1.4 Main results

The Rover Mobile Application Toolkit supports a set of programming and communication abstractions that enable mobile-aware applications to cooperate actively with the user and the underlying Rover runtime system to achieve the goals of mobile application design. Using Webcal as a case study, this thesis shows that:

- Rover successfully decouples Webcal interactive performance from network performance. Measurements show that the interactive performance of Webcal is excellent and nearly equivalent across three orders of magnitude of network bandwidths and across one order of magnitude in data size. Over a slow network, the use of QRPC and RDOs allows mobile-aware applications to return to processing user interactive tasks up to 1000 times faster than non-mobile-aware applications.

- Rover object operation logs allow mobile-aware applications to update data in time proportional to the size of the data changed, rather than the total size of application data. Transport of logs of small operations on objects can significantly ffireduce latency and bandwidth consumption compared to reads and writes of whole files.

- Rover applications replicate data to ensure high availability and reliability. Servers maintain a highly reliable data store available to any connected client. Mobile clients import copies of data and code so as to be available to the user even during disconnection.

- Rover allows mobile-aware applications to place functionality dynamically on either side of a slow network connection to minimize the amount of data transiting the network. Interface functionality can run at full speed on a mobile host, while large data manipulations may be performed on the well-connected server.

- Rover involves applications in consistency-control decisions. The most restrictive regimes are too expensive and unnecessary for most mobile applications. Similarly, the most efficient are not restrictive enough for some. Applications cooperate with Rover to achieve the appropriate balance for their cases.

- Rover exposes information about the mobile environment to applications. Mobile-aware applications can adjust their behavior or notify the user of new conditions.

- Experience indicates that porting applications to Rover generally requires relatively little change to the original application. Using Rover proxies, some applications have been used unchanged (e.g., Netscape and Mosaic).. Others have been made fully mobile aware with a change to approximately 10% of the original code and as little as three weeks work.

## 1.5 Thesis Outline

The remainder of this thesis is divided into six chapters. Chapter 2 gives a brief overview of the Rover Application Toolkit's abstractions, architecture, and programming model. Chapter 3 discusses the issues that arise in making applications mobile-aware. Chapter 4 analyzes the design and implementation of the mobile-aware calendar tool, Webcal. Chapter 5 presents a quantitative evaluation of Webcal and Rover. Chapter 6 discusses related work. Chapter 7 concludes.

# Chapter 2

# Rover Toolkit Overview

The Rover toolkit provides a mobile-aware application programming interface that optimizes common cases and supports the transfer of information between the application and the underlying system. Rover allows mobile-aware applications to obtain and act on information about the mobile environment to to maintain consistency and enhance performance. Simultaneously, Rover optimizes common-case schemes for replication and communication in intermittently connected environments. The Rover toolkit offers applications a distributed object system based on a client-server architecture. Rover applications are divided between clients that typically run on mobile hosts and support user interaction and servers that typically run on stationary hosts and hold the long term state of the system. The Rover Toolkit consists of a library linked into all applications, and runtime modules on client and server machines.

Rover applications employ a variation of the check-in, check-out model of data sharing: they *import* objects onto the local machine, *invoke* methods provided by the objects, and *export* logs of the method invocations on the objects back to servers which *reconcile* the logs with the server copies of the objects and their logs.

To create an application with Rover, a programmer divides the program into client and server functionality. The key "Roverizing" task of the programmer is to define RDOs for the data types manipulated by the application and data transported between client and server. The programmer then defines methods that update an object, including code for conflict detection and resolution.

This section briefly describes the Rover Toolkit from three perspectives. First, I describe the key communication and programming abstractions of the Toolkit. Second, I describe the structure of the Toolkit, defining the function of each component. Finally, I describe the functioning of the Toolkit in managing an RDO. For further insight into the Rover Toolkit, see [21, 22].

## 2.1 Rover Abstractions

### 2.1.1 Relocatable Dynamic Objects (RDOs)

*Relocatable dynamic objects* (or, simply, Rover objects) are objects with well-defined interfaces that can be dynamically relocated from the server to the client, or vice versa. RDOs are named by unique object identifiers and stored on servers. Rover caches objects on mobile hosts in a cache that is shared by all applications running on that host. Cached objects are secondary copies of objects which may diverge from the primary copies retained by servers. An RDO might be as simple as a calendar entry with its associated operations (*e.g.,* set appointment time) or as complex as a module that encapsulates part of an application (*e.g.,* the user interface of a calendar tool). These more complex RDOs may run in a new thread of control when they are imported. The safe execution of RDOs is ensured by executing them in a controlled environment. Controlling RDO behavior is a subject of continuing research but not of this thesis.

### 2.1.2 Queued Remote Procedure Call (QRPC)

*Queued remote procedure call* (QRPC) is a communication mechanism that permits applications to continue to make RPC requests even when a host is disconnected, with requests and responses being exchanged upon network reconnection. This asynchronous communication model allows applications to decouple themselves from the underlying communication infrastructure. During disconnected operation, the network simply appears to be very slow.

Unlike simple message passing, QRPC incorporates stub generation, marshaling and unmarshaling of arguments, and at-most-once delivery semantics. QRPC differs from traditional asynchronous RPC in its failure semantics. [10] A traditional RPC fails when a network link is unavailable or when a host crashes [4]. QRPCs are stored in a stable log so that if links become unavailable or the sender or receiver crashes, they can be replayed upon recovery. They are deleted from the log only after a response has been received from the server. In addition QRPC differs from traditional RPC in that it is decoupled from the communications channels. QRPC requests and replies may be sent on distinct communication channels and may even be sent over several channels (due, for example, to the failure of a network link in midstream). Applications are notified of the completion of QRPCs by callback.

### 2.1.3 Events

While not a new idea, *events* are key to application-level control in Rover. The Rover runtime monitors a number of properties of the mobile environment including the network and other hardware resources as well as the runtime system itself. Applications also may register callbacks in order to be notified of these changes in the environment. Application-visible system events include the queuing and dequeuing of QRPCs, creation and deletion of other Rover clients, network connection and disconnection, and change in the quality of service of the network. Similarly, applications may poll to view the log of operations or dependency vector for a particular object. Other events in the mobile environment made visible through events include changes to the power system and I/O devices.

## 2.2 Rover Architecture

The Rover Toolkit consists of a library linked into all applications and runtime modules for client and server machines. The client runtime module is structured as three layers and consists of four components. At the top, is the application linked with the Rover library. Below the application is the system layer managed by the access manager and encompassing the object cache and the operation log. Finally, the network scheduler is the transport layer for the system sitting between the network and the rest of the runtime system. The server run time module is similar in structure. The function of each component is discussed below.

### 2.2.1 Library

Each Rover application is linked with the *Rover library*. This library defines the Rover application programming interface and manages communication between the client application and the Rover runtime system. In addition, the library manages the portion of the Rover object cache located in the address space of the client. Rover applications are typically structured in an event driven, non-blocking style to allow the Rover library to handle messages from the Rover runtime system as they arrive. However, this structure is not strictly necessary. The use of distinct processes to handle the division of responsibility between the client application and the Rover runtime system provides all necessary parallelism and asynchrony.

### 2.2.2 Client Access Manager

Each client machine has a local Rover *access manager* that mediates all interactions between client applications and servers. The access manager gives applications a consistent communication interface even in the presence of intermittent network connectivity. The access manager services requests for objects, mediates network access, manages the object cache, and logs modifications to objects. Client applications use the access manager to import objects from servers and cache them locally. Applications invoke the methods provided by the objects and, using the access manager, make changes visible globally by exporting logs of the changes to the objects back to the servers.

Within the access manager, objects are imported into the object cache, while QRPCs are exported to the QRPC log. The access manager routes invocations and responses between applications, the cache, and the QRPC log. The log is drained by the network scheduler, which mediates between the various communication protocols and network interfaces.

### 2.2.3 Object Cache

The *object cache* provides stable storage for local copies of imported objects. The cache stores both the last known durable state of objects (updated only at the direction of the server) and the current tentative state of objects (updated by client applications). In addition, each object is cached with a logical dependency vector received from the server that indicates the last modification time of the durable state of the object. The object cache consists of a local private cache located within the application's address space (for efficiency) and a global shared cache located within the access manager's address space (for stability and sharing). Client applications do not usually directly interact with the object cache. The client library and access manager coordinate to map import and export operations onto objects cached both within the application's address space and the access manager's address space.

### 2.2.4 QRPC Log

Once an object has been imported into the client application's local address space, method invocations without side effects are serviced locally by the object. In order for an operation to have a durable effect on an object, an export QRPC is entered in the stable *QRPC log* located at the client. The QRPC causes the server to perform the operation on the primary copy of the object. The log is flushed to the server asynchronously with application activity.

Support for intermittent network connectivity is accomplished by allowing the log to be flushed back to the server incrementally. Thus, as network connectivity comes and goes, the client will make progress towards reaching a consistent state. The price of local updates is that the client's and server's copies of an object will diverge over time. At some point, network connectivity will be restored and exported modifications to objects will have to be reconciled with any changes to the server's copies. Only after reconciliation can an operation be considered durable.

### 2.2.5 Network Scheduler

The Rover *network scheduler* drains operations from the log and transmits them to servers. The network scheduler controls when and which communication interfaces are opened and what should be sent over the interface. The scheduler is responsible for retransmission in the case of link failure. Thus, the Rover runtime system permits applications to largely ignore network connectivity as an issue.

On the other hand, the applications provide useful information to help optimize network activity.

The network scheduler groups related operations together for transmission. The Toolkit leverages off the queuing performed by the log to gain transmission efficiency. The network scheduler may reorder logged requests based upon two primary criteria: the application's consistency requirements and application-specified operation priorities. Using these criteria, the scheduler provides an ordering for flushing operations from the log. Scheduling QRPCs is a subject of continuing research but not of this thesis.

### 2.2.6 Server

The Rover server runtime system is analogous to that of the client. However, the server does not have the added complexity of an object cache. Instead, each application is responsible for managing its own object store (or cooperating with other applications to co-manage objects) and servicing QRPCs directed to it. The server runtime is responsible for de-multiplexing, logging, and scheduling QRPC requests and replies. In addition, the server library provides calls to help manage and store objects and logs of operations on objects.

## 2.3 Using and Managing RDOs

Usually, applications use two flavors of queued remote procedure call to transport and update relocatable dynamic objects: import and export. The use of these QRPCs and their relation to object invocation and reconciliation are described below.

### 2.3.1 Import

When an application imports an object, Rover first checks the object cache. If the object is resident and the application accepts cached object copies, the application receives a working copy of the cached object. When an application invokes a method on an object, it may either directly invoke the operation on its working copy of the object or it may call Rover to invoke the operation. In the latter case, in addition to performing the operation on the working copy of the object, the operation and the new tentative state of the object are stored in the object cache. The operation is stored in a log associated with the object. (Do not confuse the log of operations on an object stored in the cache with the QRPC log.) The tentative copy of the object is stored in addition to the durable state received directly from the server during import. In addition, the access manager sends the operation log to the server in an export QRPC. The durable state of the object is updated when the server reports the result of performing and reconciling the operation on its primary copy of the object. By using working objects with tentative state, applications can continue computation even if the mobile host is disconnected. Rover applications typically reflect the fact that an object has tentative state to the user (*e.g.,* by displaying them in a different color).

If an object is not present in the object cache at import time, Rover lazily fetches it from the server using a queued remote procedure call. Rover stores the import QRPC in the QRPC log and returns control to the application. The application can register a callback routine with Rover that will be called by Rover to notify the application when the object arrives. Whenever the mobile host is connected, the Rover network scheduler drains the QRPC log in the background and forwards QRPCs to the server.

Upon arrival of an import QRPC at the server, the server access manager invokes the appropriate application stub to the requested object. The application hands the RDO back to the access manager which logs the reply and sends it to mobile host. If a mobile host is disconnected between sending the request and the reply, Rover will replay the request from its QRPC log upon reconnection.

Upon receiving an import QRPC reply, Rover inserts the returned object into the cache the application and deletes the QRPC from the QRPC log. In addition, if a callback routine is registered, Rover will perform the callback to inform the application that the object has arrived. The application can then invoke methods on the local copy.

### 2.3.2 Export

When an an application invokes a method through Rover to modify a cached object, Rover lazily updates the primary copy at the server by sending the method call in an export QRPC to the server, and returns control to the application.

When the export QRPC arrives at the server, the server access manager invokes the application export stub. The application invokes the requested method on the primary copy of the object. Typically a method call first checks whether the object has changed since the client last received an update for the object. Rover maintains version vectors for each object so that applications can easily detect such changes. If the object has not changed, the method modifies the primary copy and hands the log of the result back to the server access manager. The server then logs the reply and sends it back to the mobile host.

If a method call at the server detects that the object has changed since the client last updated the object, the server copy of the object must be reconciled with the operation log sent by the client. The Rover Toolkit itself does not enforce any specific concurrency control mechanism or consistency guarantees of objects. Instead, it provides a mechanism for detecting conflicts and maintaining operation logs while leaving it up to applications to reconcile objects. For example, the Rover Webcal distributed calendar tool exploits semantic knowledge about calendars, appointments, and notices to determine whether a change violates consistency. Concurrently deleting two different appointments in the same calendar does not result in a conflict. However, the client is informed of the concurrent delete, so that the client copy of the calendar will reflect the second delete. If there is a conflict that cannot be reconciled, the method returns with an error. These errors are reflected to the user so that he or she can resolve the conflict.

In general, the server reply consists of a log of operations which will transform the durable state of the client copy of the RDO to reflect the newly computed state of the primary RDO at the server. The reply may indicate either the successful completion of the operation originally submitted by the client or a substitute operation to be performed. The substitute may include any number of operations that occurred at the server between the time the client received that last update and the time of the reply. Upon arrival of the reply at the client, Rover deletes the QRPC from the stable log, retrieves the permanent copy of the object from the cache, applies the entire log of operations indicated by the server and invokes the callback associated with the original operation (if one is registered). The resulting new durable state of the object is cached and the tentative state deleted. To complete the reconciliation process, Rover then replays any other outstanding tentative operations logged on the object, thereby creating a new tentative state to store in the object cache, and invokes any callbacks on those operations.

# Chapter 3

# Design Issues in Mobile-Aware Computing

## 3.1 Object Design

As the central structures about which all Rover design decisions revolve, relocatable dynamic objects (RDOs) provide the key fulcrum for application-level control in Rover applications. The primary abstractions used by Rover applications are RDOs. All code making up applications and all data touched by applications can be cast into RDOs. Thus, mobile-aware applications leverage RDOs to achieve high performance while maintaining correctness.

RDOs are replicated and cached for reliability and availability. This replication requires applications to maintain the consistency of RDOs. RDOs may function at either the client or server. RDOs run on both the client and server and must be designed to function while QRPCs may be arbitrarily delayed by network disruptions. All permanent changes to the data store are performed through RDO method invocation, relayed by QRPCs. These methods must be carefully designed to maintain consistency, resolve conflicts, and yet achieve high performance. Using the available information about the network, RDO method invocations should be scheduled to balance network utilization with the importance of the data. Similarly, the type of network connection should be selected.

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, application designers can carefully manage resource utilization in a way impossible at the level of a replication system that handles only generic data. In Rover, applications are built on an object model, so this coupling is extremely natural. For example, applications can trade computation time for data transfer time to alleviate network latency. An RDO can include compression and decompression methods along with compressed data in order to obtain application-specific and situation-specific compression to reduce both network and storage utilization. A replication system attempting to provide mobile-transparent service cannot benefit from such domain specific knowledge.

Another advantage of application-level control over generic data-level replication is that methods can use application-specific knowledge to make efficient use of resources. This is clearly the case in deciding the proper size of objects for replication. For example, file-based mobile computing systems such as Coda [26] and Little Work [16] replicate the entire file. Any update to the file causes the entire file to be written back to the primary file server. However, when working across a slow or intermittent link, a small update size is key to efficient network usage and low latency. An application should never be forced to write unnecessary data to the network. Operation logging allows applications to write only the update and not the entire object. Mobile-unaware applications will treat the mobile file system the same as a non-mobile one and unnecessarily write large amounts of data. Therefore, application designers must be careful in selecting the size of objects and operations. Objects should not be so large as to cause extra data to be moved across the network but should not be so small that overhead dominates data content.

## 3.2 Computation Relocation

Rover gives applications is control over the location where computation will be performed. In an intermittently-connected environment, the network often separates an application from the data on which it is computing. By moving RDOs across the network, application designers may move data and/or computation from client to server and vice-versa, depending on which type of migration is more efficient. Typically, computation is smaller. Furthermore, since RDOs can be dynamically reloaded at runtime, applications can reconfigure the location of computation or data in response to current or predicted resource limitations. Finally, applications can perform computations against large data sets which return small results and only move the result across the network. These techniques enable applications to be as responsive in mobile environment as they are in stationary ones.

For example, Graphical User Interfaces (GUIs) can be downloaded and customized for a mobile host. Using GUIs over wide-area or low bandwidth networks can be an exercise in frustration. Typical GUI-based programs process a large amount of data concerning mouse movements, windowing events, key presses, etc., in order to generate a fairly small number of changes to the permanent data that the program manipulates. Executing the GUI on the mobile host while sending updates to stored data across the network allows the user to have full-speed interactions with the program while using the network efficiently. Input events can be processed locally, even if the client becomes disconnected. In addition, these interactive programs can change their appearance depending upon the available display resources and available bandwidth for fetching large images.

Similarly, performing actions on the server side of the network can have great benefits. A Rover server can be dynamically configured to perform arbitrary computation on behalf of the client. Application-specific data compression was mentioned earlier. A similar, but longer-lived computation, is to perform filtering actions against a dynamic data stream. Without an RDO executing at the server, the application would either have to poll or rely upon server callbacks. While this might be acceptable during connected operation, it is not acceptable during disconnected operation or with intermittent connectivity. Furthermore, every change to the data would have to be returned to the client for processing. With RDOs, the desired filtering or processing can be performed at the server, with only the processed results returned to the client. For example, a financial client making decisions based upon a set of stock prices could construct an RDO that would watch prices at the server and report back only significant changes, thereby significantly reducing the amount of information transmitted from the server to the client.

## 3.3 Notification

Since the mobile environment may be extremely dynamic, it is important to present the user and the application with information about its current state. The Rover Toolkit provides applications with information about the environment for dynamic decision making or presentation to the user. Applications may use either polling or callback models to determine the state of the mobile environment.

The environment consists of the state of imported RDOs, the state of the network, and the state of the mobile host. The RDO life cycle consists of: being imported but not yet present; being present in the local environment; being modified locally by method invocations; having log(s) of operations exported to the server; being reconciled or committed; and being evicted from the environment. A network interface may be present or absent, and, if present, is characterized by cost and quality of service: latency and bandwidth. Applications can register methods to be invoked for each change in the state of an RDO or of the network. The state of the host includes its available hardware resources including screen size, data and energy storage capacity, and input devices.

Applications can forward these notifications to users or use them for silent policy changes. For example, in the calendar application described in Section 4, appointments that have been modified but have not yet reconciled are displayed in a distinctive color. Such notification allows users to tolerate a wider array of application behaviors. Flagging tentative appointments lets the user know the appointment may be canceled due to conflict. Furthermore, the user knows that, since the data has not been reconciled with the server, no other user has yet seen the proposed change to the calendar.

This same application uses notifications about network connectivity to attempt to schedule communication with the server. The application only enqueues server polling operations while the network is connected. Thus, information about the network state enables the application to reduce future network usage. Similarly, the Rover Web browser proxy [21, 7] can use information about the available network bandwidth to decide whether to "inline" images in Web pages.

Using knowledge about the state of the host allows Rover applications to be dynamically extended. Rover starts as a minimalistic "kernel" that imports 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. However, if the application finds more hardware and network resources available—say if the mobile host is docked—further RDOs may be loaded to handle these cases [23].

## 3.4 Replication

Data and code replication is the chief technique Rover employs to enable client applications to achieve high availability, concurrency, and reliability. Since each host has a copy of all relevant code and data, applications can continue to operate in the absence of network connections. Similarly, replication enables each host to operate on the replicated data concurrently. Since servers are less prone to data loss, maintaining data at the server protects against mishaps at the client.

While replication can bring great benefits, application designers must carefully select the proper replication strategy to minimize its costs. Keeping multiple replicas consistent entails additional communication, increased latencies, potential for dead-lock, and use of additional resources. Applications should not replicate any more data than absolutely necessary and should strive to keep update messages small. Strategies for reducing consistency-related-costs are discussed in detail in the next section.

## 3.5 Consistency

With replication also comes the need for consistency control. No one consistency scheme is appropriate to all applications. Therefore, Rover leaves the selection of consistency scheme to the application. Alternatives range from no consistency control to pessimistic, application-level, two-phase locks guaranteeing fully serializable execution. However, a limited number of consistency control schemes lend themselves naturally to intermittently connected environments. Pessimistic consistency control schemes may block a mobile host from making progress whenever it is disconnected.

Rover supports primary-copy, tentative-update replication. That is, the system always treats the information received from the server as overriding the tentative state and operations stored at the client. However, nothing prevents applications from always accepting client updates. Application designers create operations to support whichever consistency paradigm they select.

Only a limited number of schemes seem particularly appropriate to mobile computing. (See [14] for a full analysis of the alternatives.) The simplest scheme is simply no consistency control, or hand-edited control. This trivial approach is appropriate for some applications. Applications can take advantage of other aspects of the Rover Toolkit without imposing a consistency control scheme.

One common *ad hoc* approach taken by Lotus Notes [25], mail systems [37], and the Internet name service [36] among others, is to require all replicas of the data store to converge to the same values. Three techniques can be used to obtain convergence without serializing updates: append, replace-with-value, and commutative updates. In the first two, each update is time-stamped or version-vectored. Time-stamps require a notion of eternal global time and some level of clock synchronization. Version-vectors increase with each system "event" and may produce a logical notion of "happens before" [32]. Some implementations of version-vectors consider each update at the local-host an "event". Thus, logical time increases faster at more actively writing hosts. For append, available updates are stored in time-stamp order. For time-stamped replace-with-value, only updates with time-stamps later than the last received update are accepted, "stale" updates are silently discarded by the server. Commutative updates are data transformations that may be applied in any order. No update depends on the result of any previous update. Commutative updates have the advantage that the server need only confirm the updates' success, not positions in the stream of updates. Using convergent schemes, all connected clients eventually see the same value for the object but not necessarily all updates.

The logging of method invocations rather than the simple overwrite of data values allows increased flexibility in dealing with possible conflicts. For example, a financial account object with debit, credit, and balance methods provides a great deal more semantic information to the application programmer than a simple account file containing only the balance. Debit and credit operations from multiple clients could be arbitrarily interleaved as long as the balance never becomes negative. In contrast, concurrent updates to a balance value would require that each client transaction have access to the global balance and that updates to that balance be globally ordered. Such a scheme would be inappropriate for many mobile computing applications which need update-anywhere semantics, even during disconnection.

Convergence is one desirable property but may not be sufficient for all applications. Since stale updates are discarded, not all clients see the same stream of updates. The converged state will not necessarily encompass the effects of all updates. If intermediate states and updates are not important to the application, convergence-based consistency control may be appropriate for mobile computing systems. It is easier to implement and may run faster and avoid the complexity of conflict resolution that serializable schemes require.

Some applications require greater consistency guarantees. At the extreme, applications may require ACID (atomic, consistent, isolated, durable) transactions. Rover provides no direct support for transactions. There is no call to lock an RDO. However, application-level locks, version vectors, or dependency-set checks may be used to implement fully-serializable transactions within Rover method calls. Unfortunately, pessimistic, or eager, concurrency control — acquiring locks on all shared resources before use — is generally inappropriate for intermittently-connected environments.[1] A single disconnected host may stop all computers sharing a database from making progress.

Optimistic, or lazy, concurrency control schemes allow updates by any host on any local data. Any conflicts caused by this policy are settled later by reconciliation. This property makes optimistic concurrency control attractive for mobile computing. However, [14] predicts that the number of conflicts (and therefore, the number of reconciliations) in an optimistic concurrency control scheme grows quadratically with the rate of transactions and the number of hosts in the system. In a peer-to-peer replication scheme this growing number of conflicts means that as the system scales, each peer is likely to have an increasingly incorrect view of the system.

Rover applications use a primary-copy optimistic replication scheme to avoid this problem. The server side of the application is responsible for maintaining the consistent view of the system. The client side diverges from that view only by the actions of a single user. Thus, the client only needs submit tentative operations to the server to reconcile the system state. After the server executes the operations, and relays the results, the client (and user) can be assured any updates are durable.

The definition of conflicting modifications is strongly application- and data-specific. Therefore, Rover does not try to detect conflicts directly. Since the submitted operation is tentative and was originally performed at the client on tentative data, the result of performing the operation at the server may not be exactly what the client expected. However, the result may be acceptable. The application designer must embed conflict detection checks and resolution procedures in the tentative operation to discover if the result is acceptable. Note that conflict detection may depend not only on the application but on the data or even the operation involved. For example, our calendar tool implements different consistency protocols for calendars and items. Calendars are essentially sets of other objects. As inclusion and exclusion operations are the only allowed methods on calendars, we simply let the server serialize the operations and never report a conflict. On the other hand, individual appointments are treated as units of consistency. Concurrent updates to the

[1] From the transactions point of view, these schemes are eager. Each transaction eagerly seeks to complete on all hosts at once. From a conflict point of view, they are pessimistic. The locking discipline pessimistically assumes each transaction will cause a conflict.

same appointment may or may not result in conflict (depending on the type of update). These two distinct data-type specific policies exemplify how the Rover architecture allows application designers to build applications with the desired degree of consistency for application-level operations.

# Chapter 4

# Webcal: A Mobile-Aware Calendar Tool

This section examines one application ported to Rover: Webcal, a mobile-aware version of the Ical calendar tool. The purpose of this examination is to highlight the differences between the mobile-aware and mobile-unaware versions of this application. In particular, this section points out both how application-level control is necessary for correct operation and how it enhances interactive performance.

Ical depends on a high-bandwidth, low-latency, continuous network connection for a number of functions. The key to understanding how Webcal differs from Ical is to understand these dependencies and how they hobble Ical in a mobile environment. Section 4.1 explains Ical's basic functionality. Section 4.2 explains how Ical functionality is impeded in a mobile environment. Section 4.3 explains how the design of Webcal addresses these weaknesses. Section 4.4 discusses the implementation of Webcal.

## 4.1 Ical basics

The Ical calendar program, written by Sanjay Ghemawat, provides an X interface for displaying and maintaining appointment calendars [11]. Calendars may contain other calendars and include items displayed to the user. An item is either an appointment or a notice. Appointments start and finish at particular times of the day. Notices do not have any starting or ending time. Notices are useful for marking certain days as special.

The Ical interface includes monthly and daily views of appointments and notices. Items can be added, deleted, or edited with point-and-click mouse operations, with keyboard entries, or with user generated scripts. Each item is displayed in full on the day it occurs with text describing the item appearing in visually depicted blocks of time for appointments.

In Ical, calendars are stored in files. A calendar is, essentially, just a set of items. A main calendar also has the list of options specifying user preferences and a list of other calendars whose items should be displayed. Items from the main calendar and from these “included” calendars are shown to the user. All options for a calendar, all calendars it includes, and all the items it includes are stored together in the single file. Ical calendars tend to grow over time. Users rarely delete items. Calendars can grow to be hundreds of items and tens of kilobytes in size.

Under Ical, reading and writing large calendars is not a problem since the bandwidth to the file system is large. The file system is either local to the executing machine or it is accessible over a high bandwidth network. Ical reads and writes entire calendar files as one atomic unit, blocking all other operation until the I/O operation is completed. Thus, Ical depends on continuous availability of a high-bandwidth connection to the file system in order to provide adequate performance.

Ical uses whole-file reads and writes for a reason. They make maintenance of data consistency simpler. In Ical, files are the unit of consistency. Ical uses file-modify times as a time stamp. Whenever a calendar file is read, Ical stores the file-modify time as a time stamp. When writing out calendars, either at the explicit request of the user or periodically, Ical checks the file-modify time. There are two cases. First, if the file-modify time has not changed, the file may be overwritten with impunity. Alternatively, if the file-modify time has changed; the file has been re-written since the last read. Ical assumes the stored data and the data it wants to write conflict. The user is asked to select which set of data to use.

In both cases Ical is vulnerable to data loss. Both these cases are minimized by the low latency of disk operations. In the latter (conflicting) case, the user is asked to select which data to use, and therefore, implicitly which data to throw away. The user must select between the data stored on disk (generally changed in some unknown way) and the data Ical is trying to store (generally changed by the user). Either way, reconstructing the lost data is left entirely to the user. (The user may also try to recover by renaming the data but this can be as cumbersome as reconstructing it.) In the former (non-conflicting) case, the vulnerability is more subtle. The read and comparison of the file-modify time is not atomic with the writing of the data. However, since the network file system is implemented over a low latency network, this time is small enough that the risk is tolerable. (The actual write of the data is performed in an atomic manner, so there is no risk of corruption, only of inconsistency.)

These vulnerabilities are tolerable because Ical periodically and frequently reads its entire data set. Thus, there are two windows of vulnerability when a write by concurrently running programs will create a conflict. The first is the period between the time Ical polls the file system for data and the time Ical attempts to write data. The second is the period between the time the user enters data into Ical and the time Ical polls the file system for data. Both of these windows are increased in size by the time between file system write and flush. (That is, polling for data only reports data older than the write-flush window.) During reads and writes Ical is blocked. The user can make no progress. Thus, the fact that reads and writes are operations quick enough to do often allows the user to set the frequency of reads and writes to be high enough that conflicts are rare.

Since calendars are shared among many users (popular calendars are typically shared by tens of people at the MIT Laboratory for Computer Science), the data must remain continuously available to all users. Therefore, calendars usually reside on a networked file system and consistency is really determined by the underlying file system.

## 4.2 Problems with Mobile Ical

In a mobile environment, Ical's assumption of a high-bandwidth, low-latency, continuously available link to the file system is violated. Network file operations are often unavailable and when available simply take too long. Writes and reads may only happen when a network connection is available. Thus, simple periodic reads and writes cannot be allowed. (They would either block or crash the program.) Bandwidth is a scarce resource. Writing or reading huge amounts of unchanged data in order to update small items wastes inordinate amounts of bandwidth and degrades performance unacceptably.

The latency for even small network operations becomes large. The turn around time between a stat and a write can become so large as to be an unacceptable approximation of atomicity. The larger and inherent problem is that during periods of disconnection, the file system version of the data and the program version of the data diverge. The time between reads may grow to be on the order of days, not seconds. Thus, entire user sessions become part of one Ical write operation. Ical's all-or-nothing policy regarding writes becomes completely inappropriate.

An even larger latency problem shows up if one tries to run Ical on a machine connected to the fast wired network while displaying its interface remotely on the mobile-host. While file operations will be full speed, the X-interface must run over the slow link from the mobile computer to the fully connected machine. Obviously, this means the user is unable to use the program altogether during periods of disconnection. Even if the user were willing to accept that restriction, running X over a slow network so disrupts interactive performance as to be unusable. Even when using a version of the X protocol optimized for use over slow connections, the round trip latency is simply too large and bandwidth too small to transmit each mouse movement, window-entering event, or keyclick to the fixed machine and receive window-drawing instructions in response.

## 4.3 Webcal Design

Webcal is the Rover redesign of Ical. Functionally, Webcal improves on Ical by requiring minimal user intervention in the presence of intermittent network connections, optimizing the usage of network bandwidth, and displaying the tentative nature of data to the user. Three design differences between Ical and Webcal account for these advantages. First, Webcal uses a fine-grained object model in place of Ical's coarse-grained, file-based data model. Second, Webcal splits computation between the client and the server. Third, Webcal implements a mobile-aware user interface.

### 4.3.1 Small data granularity

Webcal altered the Ical data model to conform better to the mobile environment. Semantically, items are independent of the calendars in which they are listed. Items can be changed without affecting other items or the encompassing calendar. The only relation between an item and a calendar is the "includes" relation. That is, a calendar may include an item and each item is included by some calendar. Therefore, items, in addition to calendars, are first class objects in Webcal. Each item maps to an RDO that is named, stored, fetched, cached, and updated independently of other items in its calendar. Each item has a globally unique name, independent of its including calendar. Detection of conflicting concurrent updates is performed on the granularity of individual items.

The effect of the change to an item-centered data model is two-fold. Concurrent updates result in conflict less often and all updates are smaller. First, the number of objects in data storage increases by roughly two orders of magnitude (i.e., by the number of items per calendar). Thus, the chance for two different users to interfere with each others' work decreases proportionately. When such conflicts do occur, they are isolated to the particular items which have been updated in a conflicting manner. Thus, false sharing (and false conflicts) in the database decreases. Only concurrent changes that alter the same item trigger conflict resolution in Webcal. Second, the size of each object in the data store also decreases by orders of magnitude since each object is now an item instead of a collection of items. Webcal communicates updates to the server rather than whole calendars. Thus, much less static data transits the network. Both updates and (rare) conflict resolution can proceed quickly.

### 4.3.2 Computation relocation

In addition to altering the Ical data model, Webcal alters the Ical computational model by splitting functionality between the client and server. The client is responsible for interacting with the user. The server manages data storage. Thus, the graphical user interface is a set of RDOs executed entirely on the client. The client performs all interface computation on the client machine. Such computation ranges from window layout to tracking and triggering alarms. No data transits the network unless the data store is affected. When data is stored, Webcal exports the affected RDOs by means of QRPCs. The use of QRPC for communication decouples the application from the functioning of the network. Locating the interface RDOs on the client allows Webcal to accept user input and function as if the client were fully connected, even while completely disconnected.

The server is responsible for serializing concurrent updates to Webcal calendars. The server maintains the single master copy of a Webcal object store. Objects may be stored in Ical calendar files for backward compatibility or as objects in a database. In addition, each object has a log of operations that is the serialized history of that item or calendar. The current object state is simply the memo-ized result of playing the log. Each log entry is time stamped by the server when it is made. These time stamps are used in the reconciliation process to extract a suffix of the log to send back to the client.

### 4.3.3 User notification

There are only two differences in the user interface between Ical and Webcal. First, Webcal displays to the user the information that the Rover Toolkit makes available about the status of RDO updates. Items that are known to reflect the durable state of the item are displayed normally. Items that have changes that are not yet stable are displayed in a special color. In addition, whenever such items are selected, the status line of the calendar displays text labeling the current state of the item.. As an item moves from one category to the other—either because the user changes an item or because new information arrives from the server—the display is updated. The second interface difference flows from the change in the data model. Users are asked to resolve conflicts on an item-by-item—rather than whole-calendar—basis. Due to disconnection, the time between a user edit occurring and the time Webcal informs the user of a conflict caused by that edit may be a period of hours or days. Thus the user is much less likely to remember exactly what action precipitated the conflict. The decreased granularity of conflict notification allows the user to see precisely where the conflict is and take action to resolve it.

### 4.3.4 Consistency

Clients generate operations on RDOs (corresponding to items and calendars) in the store. Each update is associated with a dependency vector signifying the durable version of the object against which the operation was generated. In Webcal, this dependency vector is *not* used to determine whether to apply its tentative operation or reconciliation code. Rather, Webcal uses a content-dependent scheme described below in Section 4.4.2. Each client plays the operation it generates against its object cache but the server invocation of the operation overrides that tentative action at the client. Invoking operations at the single server results in a single, serialized log of object updates.

| Rover Program | Base code | New Rover client code | New Rover server code |
|---|---|---|---|
| Webcal | 26,000 C++ and Tcl/Tk | 2,600 C++ and Tcl/Tk | 1,300 C++ and Tcl/Tk |

Table 4.1: Lines of code changed or added in porting Webcal

Clients receive the (serialized) suffix of the object operation log that represents all operations on the object (by that client and all others) since the last time the client received an update.

Upon receiving that log, the client reverts the object back to the durable state from the tentative version due to the application of unconfirmed operations. The client then evaluates the log suffix against the old durable state of the object, bringing the client cache into synch with the server. This new durable state is then cached. The user is notified of any rejected operations and asked to resubmit them. The client then continues by reapplying the still unconfirmed operations to the object to generate the new tentative state of the object to store in the cache.

## 4.4 Webcal Implementation

The implementation of Webcal had the following four goals:

- Webcal must maintain data consistently across periods of disconnection without preventing users from making progress or entering new data.
- Webcal should minimize network utilization for common operations.
- Webcal should maintain interactive application performance during periods of disconnection equivalent to that available while connected by a high speed network.
- Webcal should be easy to implement. Therefore, there should be only minimal changes to the Ical code base and great ease of debugging.

This section addresses how the implementation meets those goals. The first part describes the Ical Tcl interface through which Webcal is implemented on top of Ical. Following that is a description of the algorithms implemented to maintain data consistency. The chapter concludes with a discussion of caveats and compromises of the implementation.

### 4.4.1 Ical Tcl Interface

In Ical, calendars and items are stored persistently on file systems. Copies of these calendars and items are read into Ical's address space. The Tcl code operates on these copies through well-defined interfaces. Part of this interface is implemented in C++ and the rest is implemented by Tcl support libraries. The C++ code exports calendar and item objects to the Tcl interpreter. A number of operations are provided to create such objects and to manipulate dates and times. In addition, the calendar and item objects have numerous methods that can be called from Tcl code. The Ical GUI code is the frontend of the program that uses this Tcl interface to access and manipulate calendar and item data stored in the C++ backend and file system.

In the port of Ical to Webcal, rather than reimplement the entire C++ backing in Tcl-based RDOs, one more layer of Tcl objects is added on top of the already extant system. Each C++-backed Tcl calendar or item object is *shadowed* with a Rover RDO. These RDOs transport the data for calendars and items to and from the server which maintains the object store. Values for these objects are transfered to and from the C++-backed Tcl objects using Ical's Tcl interface. In the Webcal client, access to the file system is disabled and replaced by calls to import or export the shadow RDOs. Thus, the Rover layer now acts as the backend to the C++-backed Tcl GUI frontend. Table 4.1 lists the total amount of code written for Webcal and relates to the base of Ical code.

### 4.4.2 Data Consistency in Webcal

Webcal uses a combination of time stamps, checkpointing, and logging to maintain data consistency. Calendar and item data is transported between the client and server portions of Webcal in RDOs. These RDOs do not directly implement user interface operations. Instead, the RDOs update Ical C++-backed Tcl calendar and item objects. Thus, data consistency must be maintained in two ways: between the client and server version of the RDOs and between the client RDO and the Ical interface objects.

The following section describes the "life cycle" of an item in Webcal. This section concentrates on the item RDO because it exercises all the complexity of maintaining consistent state. The handling of calendar RDOs is similar but simpler.

#### Webcal Item Import

Initially, an item is imported from the server into the client using Algorithm ITEM_IMPORT (Figure 4-1). As with other Rover RDOs, Ical items are imported through the Rover access manager. When a response from the server arrives, the Rover runtime system initiates the ROVER_UPDATE step as shown in Figure 4-2 Algorithm ROVER_UPDATE is run whenever new data is received from the server, i.e. in response to either an import or an export. First, Rover breaks the server message into its components: code, data, time stamp, and log. In response to an initial request for an item, the server will return code and state for the RDO. In response to subsequent communication, only the log will be present. The library then instantiates the durable state of the RDO, from the cached value or from the new information returned by the server. If the server sent a log in the message,

```
ITEM_IMPORT(item) {
    ROVER_IMPORT(item, ITEM_CALLBACK)
}

ROVER_IMPORT(item, callback) {
    cache→item→callback := callback
    send (item)
}
```

Figure 4-1: Algorithm ITEM_IMPORT

the library applies the operations received from the server. Each operation on an item begins with a conflict detection check. The operation checks the current value of the RDO and program against the expected values stored in the operation. An unexpected value will cause the client to raise an exception and notify the user. Having passed this test, the operation completes and stores an internal checkpoint of its state for later use. In and of itself, this has no effect on the Webcal GUI. However, the callback immediately updates the frontend Ical GUI item with the result that the user now sees the same data the server stored when the message to the client was generated.

### Webcal Item Export

When user changes are to be saved (initiated either by a timer, the network monitor callback, or the user), Webcal initiates ITEM_EXPORT step. There are four parts to ITEM_EXPORT as shown in Figure 4-4.

First, Webcal compares the content of the Ical frontend GUI item with the content stored in the internal RDO checkpoint. During the course of use, the GUI is allowed to drift away from the backend RDOs. User actions affect what the user sees and the internal state of the program but not the Rover backend that Webcal adds to Ical. So, if content of the GUI frontend item is the same as the content of the Rover backend RDO, the user has not changed anything since information was last received from or sent to the server. Thus, no new operation needs to be sent to the server.

Second, the export generates the operation. Each operation consists of the four parts discussed in the next section.

Third, Webcal must check that the log generation process has been atomic. Rover provides no explicit locking mechanism. Thus, any call into the Rover library may allow the library to invoke event driven callbacks, including callbacks that modify RDOs. While the approach avoids deadlock, it does require an extra check in the export path. It is possible for the Rover library to receive an update to the RDO while the operation log is being generated. In that case the log would be invalid and the generation process is restarted. Therefore, Webcal saves the dependency vector for the RDO before the log is generated and checks that it has remained unchanged through the course

```
ROVER_UPDATE(item, server_message) {
    cache→item→time_stamp := server_message→time_stamp
    cache→item→code := server_message→code
    cache→item→durable_state := server_message→data
    server_log := server_message→log
    unmarshall (cache→item→durable_state)
    APPLY_SERVER_LOG(item, server_log)
    cache→item→durable_state := marshall (item, server_log)
    APPLY_TENTATIVE_LOG(item)
    cache→item→tentative_state := marshall (item)
}

APPLY_SERVER_LOG(item, server_log) {
    foreach operation in server_log {
        apply (item, operation)
        if operation in cache→item→log {
            invoke (cache→item→callback)
            delete operation from cache→item→log
        }
    }
}

APPLY_TENTATIVE_LOG(item) {
    foreach operation in cache→item→log {
        apply (item, operation)
        if error {
            invoke (cache→item→callback)
            delete operation from cache→item→log
        }
    }
}
```

Figure 4-2: Algorithm ROVER_UPDATE

```
ITEM_CALLBACK(item) {
    flush item to frontend
}
```

Figure 4-3: Algorithm ITEM_CALLBACK

```
ITEM_EXPORT(item) {
    IF (item→checkpoint == item→frontend_value)) {
        return /* Nothing to export */
    }
    repeat {
        stamp := ROVER_GET_TIME_STAMP(item)
        operation := GENERATE_OPERATION(item)
    } until {stamp == ROVER_GET_TIME_STAMP(item)}
    ROVER_EXPORT(item, operation, ITEM_CALLBACK)
}


GENERATE_OPERATION(item) {
    append (operation, conflict detection check)
    append (operation, conflict resolution code)
    append (operation, method invocations to update RDO)
    append (operation, checkpoint update code)
    return operation
}


ROVER_EXPORT(item, operation, callback) {
    apply (item, operation)
    append (cache→item→log, operation)
    cache→item→callback := callback
    cache→item→tentative_state := marshall (item)
    send (item, operation)
}
```

Figure 4-4: Algorithm ROVER_EXPORT

of the operation. Since the Rover library (with the cooperation of the Webcal server) maintains the dependency vector automatically, this check is easy and efficient.

Having passed the atomicity test, Webcal hands the operation log to the Rover library routine ROVER_EXPORT. The ROVER_EXPORT applies the operation to the RDO (bringing it into synch with the GUI), updates the object log and tentative state of the object in the cache, and finally sends the operation to the server. When the server reply is received, the ROVER_UPDATE step begins again.

### Webcal Operation Structure

Each operation consists of four parts (shown in Algorithm GENERATE_OPERATION in Figure 4-4): conflict detection, method invocation, conflict resolution, and checkpoint update. First, Webcal generates the conflict detection check portion of the operation. The conflict detection check is always evaluated before the corresponding operation is evaluated. The conflict detection check evaluates one of two criteria. At the server, the conflict detection check ensures that the RDO in question has the expected values. This checks for concurrent updates to the object by other clients as well as previous tentative operations that failed to commit. At the client, the conflict detection check must verify that the item frontend still matches the last checkpoint of the RDO taken at that client. That is, the operation should not overwrite changes (made to the frontend by the user) that have not yet been exported to the server. Thus, at the client, Webcal checks for concurrent access to the object between the user and the server.

Second, if a conflict is found, an exception is raised and the user notified of the conflict. The user may then select the manner in which to resolve the conflict. The user may discard the current changes to the item and accept overwriting of the current state of the frontend GUI item with the new durable state of the backend RDO. Alternatively, the user may select to re-apply the local changes to the GUI after the operation has completed.

Third, Webcal generates a set of RDO method invocations which will change the backend item RDO from its current (possibly tentative) state to one that precisely reflects the frontend Ical GUI item. While a set of method invocations is generated, the RDO is not actually changed at this point. The methods are only invoked during the Rover_Export procedure described above.

Fourth, an operation updates the internal RDO checkpoint. Webcal stores the new checkpoint to reflect the new state of the GUI. This checkpoint will be used for comparison if a new export process is initiated and for conflict detection when the server sends the next update to the client.

### 4.4.3 Caveats and Compromises

The port of Ical to Rover balances ease of implementation (porting) with efficiency of implementation. In both the data consistency model and the object storage layer, the Webcal prototype favors ease of implementation and backward compatibility over complete efficiency.

The process of checkpointing and generating a log on demand (by calculating differences) could be simplified by more extensive changes to the Ical GUI. Any significant user action could update a second dependency vector, noting that the item has been changed. Simultaneously the log for updating the RDO could be built up incrementally. While this method would be efficient and obviate the need for item checkpointing, it would require a great deal of change to the Ical GUI. The process would also be more error prone in that each point where the GUI affects the item data would have to be found and modified. Further, this method would not eliminate the need for each operation to encode its entire read set as part of its conflict detection check. At that point the basic design decision to layer shadow RDOs over the GUI rather than reimplement it becomes questionable.

The checkpointing/dependency vector tradeoff shows that the Rover Toolkit is not yet aggressive enough in pushing control up to the application layer. Applications control dependency vectors on the server side but not on the client side. This leads to the need in Webcal to implement a higher-level semantic operation.

The second compromise made in implementing Webcal stems from the desire for backward compatibility. Webcal can use and store calendars in Ical v7.0 format files, in addition to the Webcal data store. This allows Ical and Webcal users to share some calendars. The unfortunate result is that calendar files are converted to RDOs, and vice-versa, on the fly at run time. This conversion cost is fairly high and reduces server performance somewhat. However, as server performance is not on the critical path, this compromise is acceptable.

The third caveat to note is that Rover and Webcal are prototypes designed for ease of debugging. As a result, a few features are not fully implemented. For example, RDOs are never ejected from the client cache—effectively limiting the size of calendars a user can load. More importantly, QRPCs and RDOs cannot be batched for transport. The item-centered data model for Webcal described above is update optimized—small updates result in little work. However, the initial load of a calendar requires that each RDO be imported. Thus, the design causes data to be fragmented, increasing protocol overhead, and causes many network round trips. Batching would solve this problem nicely. Batching allows one QRPC to encapsulate several requests, reducing protocol overhead and round trips. An additional consequence of the fact that the Rover implementation is at the prototype stage is that data sizes for are larger than need be. Protocol overhead for QRPC is excessive and the data representation for Webcal calendars are inefficient.

# Chapter 5

# Evaluation

The following set of experiments were designed to validate the benefits of mobile-aware application design. The experiments measure the Webcal calendar tool and the Rover Toolkit as examples of mobile-aware design. The main results are:

- Rover successfully decouples interactive performance from network performance. The interactive performance of Webcal is excellent and nearly equivalent across three orders of magnitude network bandwidths and latencies, and across one order of magnitude in data size. Over a slow network, the use of QRPC allows Webcal to return to processing user interactive tasks up to 1000 times faster than Ical (Webcal's mobile-unaware predecessor.).

- Mobile-awareness allows data update times to be proportional to the amount of data changed, not the total size of application data. Transport of logs of small operations on objects can reduce latency and bandwidth consumption compared to reads and writes of whole files.

- Migrating RDOs provides Rover applications with excellent performance over moderate bandwidth links and in disconnected operation. By moving executable objects across the network to the mobile host, applications can significantly increase interactive performance.

## 5.1 Experimental Environment

### 5.1.1 Data

The experiments below exercise Ical and Webcal on three calendars constructed to reflect the behavior of users at MIT LCS. Table 5.1 shows the sizes of these calendars and the storage required to represent them in the Webcal data store. Most of the experiments involve two particular types of QRPCs: item import and item export. Imports break into two cases: cached and uncached. If the item is cached, the reply is merely a verification that the cache data is valid. If not, the

| **Measure** | **Small** | **Medium** | **Large** |
|---|---|---|---|
| Number of Items | 10 | 50 | 160 |
| Bytes in Ical file | 1448 | 7276 | 25319 |
| Webcal data bytes | 3421 | 16817 | 55187 |
| Webcal log bytes | 11529 | 57570 | 180955 |
| Webcal GDBM overhead | 7446 | 33001 | 94355 |
| Total bytes stored by Webcal server | 22396 | 107388 | 330497 |

Table 5.1: Calendar sizes used in measurements. Ical calendar sizes are selected to reflect the range of Ical calendars found at MIT LCS. Webcal object store contains calendars that are functionally identical to the equivalently sized Ical file.

| **Operation** | **Protocol** | **Code** | **Data** | **Log** |
|---|---|---|---|---|
| Cold item import request | 400 | 0 | 0 | 0 |
| Cold item import reply | 153 | 90 | 273 | 0 |
| Warm item import request | 418 | 0 | 0 | 0 |
| Warm item import reply | 151 | 0 | 0 | 0 |
| Warm export request | 315 | 0 | 0 | 785 |
| Warm export reply | 153 | 0 | 0 | 785 |

Table 5.2: Bytes transmitted (above the TCP layer) to perform typical Webcal operations.

reply contains the item data and code. The sizes of these operations are shown in Table 5.2. The implementation of an item consists of approximately 650 lines of Tcl code (~25600 bytes) that is loaded as a library as part of the start-up of Webcal. The 90 bytes of code sent in an import is the call in the library to create and unmarshall the item data.

The Webcal prototype uses the the Gnu Database Manager(GDBM) for its data store. Item operation logs and objects are stored together at the server. Currently, the Webcal data representation is designed for convenience of debugging and human readability rather than performance. This tendency is encouraged by our selection of Tcl as the implementation language for RDOs. Clearly, a byte compiled language would be more space efficient. Therefore the size of these items is excessive. Webcal calendars store the same data as Ical calendars in twice the space. In addition, each operation in the Webcal log is three times the size of item on which it operates. Neither of these data increases is inherent in the system. They are merely artifacts of the implementation. An optimized Webcal data format would be at least as small as the Ical format, likely smaller. Similarly an optimized Webcal log operation should be no more than one third the size of the original item. In addition, the prototype never truncates the log. Information older than the last time when all clients were up to date can be forgotten.

| Server: Pentium 120 | Transport | TCP | | | |
|---|---|---|---|---|---|
| | | Ping-Pong Latency | | | Throughput |
| | | null | 800 B | 2000 B | 1 MB |
| Client: TP 701C/75 | Ethernet | 10 | 13 | 31 | 4.45 |
| | WaveLAN | 69 | 117 | 284 | 1.09 |
| | 19.2 Wired CSLIP | 331 | 653 | 1874 | 0.027 |
| | 9.6 Cellular CSLIP [2] | 858 | 3170 | 2990 | 0.008 |

Table 5.3: The Rover experimental environment. Latencies are in milliseconds, throughput is in Mbit/s. Table shows mean times in milliseconds.

### 5.1.2 Baseline Performance

The Rover test environment consisted of a single server and multiple clients. The server was an Intel Advanced/XV (120 MHz Pentium) workstation running Linux 1.3.74 as the server. The Rover server ran as a stand alone TCP server. The clients were IBM ThinkPad 701C laptops (25/75MHz i80486DX4) running Linux 1.2.8. All of the machines were otherwise idle during the tests. The network options consisted of switched 10 Mbit/s Ethernet, 2 Mbit/s wireless AT&T WaveLAN, and Serial Line IP with Van Jacobson TCP/IP header compression (CSLIP) [20] over 19.2 Kbit/s V.32turbo wired and 9.6 Kbit/s ETC cellular dial-up links[1]. To minimize the effects of network traffic on our experiments, the switched Ethernet was configured such that the server, the ThinkPad Ethernet adapter, and the WaveLAN base station were the only machines on the Ethernet segment and were all on the same switch port.

The cost of a QRPC can be broken into two primary components:

1. Transport cost. The time to transmit the request and receive the reply.

2. Execution cost. The time to process the QRPC at the server.

#### Transport costs

The latency and bandwidth of various representative network technologies was measured to establish a baseline. The results are summarized in Table 5.3. The table shows the latency for 10, 800, and 200 byte ping-pong and the throughput for sending 1 Mbyte using TCP sockets over a number of networking technologies. These sizes were selected to demonstrate performance at the

[1] The configuration used was suggested by the cellular provider and the cellular modem manufacturer: 9.6 Kbit/s ETC. We connected to the laboratory's terminal server modem pool through the cellular service provider's pool of ETC cellular modems. This imposed a substantial latency (approximately 600ms) but also yielded significantly better resilience to errors. Other choices are 14.4 Kbit/s ETC and directly connecting to our laboratory's terminal server modem pool using 14.4 Kbit/s V.32bis. However, both choices suffered from significantly higher error rates, especially when the mobile host was in motion. Also, V.32bis is significantly less tolerant of the in-band signaling used by cellular phones (for cell switching and power level change requests).

[2] Unfortunately, I am not certain of the validity of the latency numbers for cellular TCP connections. Bad data was discovered late in the process and replaced with this data, collected via cellular connection from New York. Surprisingly, it seems the New York to Cambridge connection seems to be lower latency than connections initiated from here in LCS.

| Transport | Round Trip QRPC Latency | | | |
|---|---|---|---|---|
| | 350 B | 550 B | 900 B | 1900 B |
| Ethernet | 139 | 142 | 147 | 157 |
| WaveLAN | 147 | 221 | 154 | 164 |
| 19.2 Wired CSLIP | 832 | 922 | 973 | 1230 |
| 9.6 Cellular CSLIP | 4100 | 4210 | 4500 | 6210 |

Table 5.4: Time in milliseconds to perform a 350, 550, 900, and 1900 byte QRPC (including protocol overhead. Table shows mean times in milliseconds.

extremes and at the sizes corresponding to Webcal import and export operations. The throughput over wireless CSLIP is lower than expected (8.2 Kbit/s instead of 9.6 Kbit/s) because of the overhead of the ETC protocol and errors on the wireless links. The throughput over wired CSLIP is higher than expected (28 Kbit/s versus 19.2 Kbit/s) because of the compression that is performed by the modem on ASCII data. 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 Rover is sending Tcl scripts (ASCII) Rover applications will likely observe similar compression benefits when using wired CSLIP links.

**Effect of QRPC size on performance**

Table 5.4 summarizes the effect of QRPC size on performance. Combining the transport and overhead of the Rover run time system (but excluding any application) and examining the total costs, the round trip time for a 350, 550, 900, and 1900 byte QRPC (including protocol overhead) is measured. These sizes were selected to reflect the current round-trip size of Webcal item import and export operations (900 and 1900) and target sizes for optimized operations. Currently, the Rover QRPC protocol is designed for convenience of debugging and human readability rather than performance. The results show that substantial benefits are available by optimizing Webcal data representations. For example, merely reducing the export operation to the target size ($\sim$ 350 bytes) would save up to two seconds per export over slow links. Reducing the QRPC protocol overhead could have similar results.

## 5.2 Experiments

### 5.2.1 Interactive Performance

In order to demonstrate that asynchronous operation allows interactive application performance to be independent of the network speed, the time for an application to initiate network actions was measured. The two cases selected are the time for Webcal to perform Algorithm ITEM_IMPORT and Algorithm ITEM_EXPORT (See Section 4.4.2). Tables 5.5 and 5.6 summarize the results.

As a control, the time for Ical I/O operations was used. The two operations measured were the

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Webcal** | **Ical** | **Webcal** | **Ical** | **Webcal** | **Ical** |
| Local | | 0.437 | | 1.400 | | 1.490 |
| Ethernet | 0.045 | 0.704 | 0.050 | 2.670 | 0.055 | 2.380 |
| WaveLAN | 0.049 | 0.707 | 0.048 | 2.270 | 0.054 | 2.710 |
| 19.2 Wired CSLIP | 0.047 | 1.430 | 0.043 | 5.540 | 0.049 | 14.100 |
| 9.6 Cellular CSLIP | 0.044 | 4.210 | 0.046 | 16.700 | 0.049 | 50.100 |

Table 5.5: Time to initiate the import of one item from a Webcal calendar (i.e, the time to perform Algorithm ITEM_IMPORT) and the time to re-read Ical calendars over NFS. Table shows mean times in seconds.

time to re-read a calendar and the time to save a calendar. All measurements, except the local case, are over NFS.[3] What the data do not show are the NFS errors over CSLIP. Over the noisier links (Wired and Cellular CSLIP), sometimes several attempts were necessary to read or save a calendar successfully. The cellular NFS numbers were extremely difficult to obtain. Had Ical been used to save an actual calendar, it would have corrupted the calendar several times.

When Ical is performing an I/O operation, the application freezes for the duration of the operation — up to 50 seconds for slow links and large calendars. Rover QRPCs decouple Webcal from the slow networks. At the end of the operation, the application has contacted the access manager with all relevant data. As a separate process, the access manager is free to perform logging, queuing, and data transmission without substantial effect on the application. Thus, as the data shows, the time to initiate an operation is completely independent of the speed of the network.

We see that the interactive performance of Webcal is excellent. Over the 9.6Kbit cellular CSLIP link for a large calendar, the use of QRPC allows Webcal imports to return to processing user interactive tasks 1000 times faster than Ical re-reads. Even for Ethernet, interactive performance increases a factor of 15–43, depending on calendar size. Similarly, we see that Webcal export is 187 times faster than Ical save. For Ethernet, interactive performance increases a factor of 1–3.

### 5.2.2 Benefits of Small Operations

This following experiment demonstrates that mobile-aware applications benefit significantly in terms of decreased network bandwidth consumption from the use of object operation logs. The experiment compares calendar update operations in Ical and Webcal. The conclusion to note is that for Webcal, update time is proportional to the size of the changed data rather than the size of the calendar.

[3] Re-reading a calendar requires substantially more computation than saving. Re-reading involves reading from disk,deleting all the old items from memory and from the interface and then unmarshalling and instantiating the new ones. Saves mere marshall the items from memory into the storage format and then saving the items. Saves are done almost entirely at the C-++ level, while re-reads involve a lot of Tcl execution.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Webcal** | **Ical** | **Webcal** | **Ical** | **Webcal** | **Ical** |
| Local | | 0.043 | | 0.066 | | 0.127 |
| Ethernet | 0.244 | 0.265 | 0.285 | 0.486 | 0.280 | 0.932 |
| WaveLAN | 0.250 | 0.325 | 0.249 | 0.487 | 0.257 | 1.220 |
| 19.2 Wired CSLIP | 0.230 | 2.430 | 0.230 | 5.330 | 0.259 | 15.000 |
| 9.6 Cellular CSLIP | 0.243 | 10.200 | 0.252 | 23.100 | 0.267 | 50.000 |

Table 5.6: Time to initiate the export of one item from a Webcal calendar( i.e., the time to perform Algorithm ITEM_EXPORT) and the time to save Ical calendars over NFS. Table shows mean times in seconds.

The time to perform various update operations was measured to demonstrate the benefits (reduction in latency and bandwidth consumption) of small operations in a mobile environment. The time to complete import and export operations was measured. The time to completion measures the time from the initiation of the operation (with Algorithm ITEM_IMPORT or ITEM_EXPORT) until the completion of the operation callback (Algorithm ROVER_UPDATE). In other words, the time measured is the total the amount of time until the new data is received (for imports) or known stable (for exports), including time in the Rover run time system, in the access manager, on the wire, at the server and in the application. As in the last set of experiments, the Webcal operation times are compared to the corresponding Ical operations: calendar re-read and save.

To select a reasonable workload, we drew on our experience with group calendars. The rate at which calendars change is highly variable. In our experience at MIT LCS, group calendars change at the rate of a few items a week as meetings and lectures of group interest are scheduled. Personal calendars may change at a substantially higher rate. For example, a professor leaving on a trip may schedule an entire day's or week's itinerary at once. Several professors share appointment calendars with their personal assistants. In this set up, the assistant may enter the latest changes to an itinerary while the professor is on the plane. Upon landing the professor may then receive the new items or changes to the old ones. We selected operations on one and ten items.

Tables 5.7 and 5.8 summarize the results of the import experiments. Tables 5.9 and 5.10 summarize the results of the export experiments. We see that times to complete imports and exports depend only on the amount of new data to be obtained while re-reads and saves depend on the size of the whole calendar. If only a single item has changed, Webcal import is 1.6–10 faster than Ical re-read at obtaining the new information.

If ten items have been changed, the performance of the Webcal prototype is poor. The Rover prototype does not currently support batching of operations. Further, the implementation does not yet provide ordering guarantees on operations. Therefore, Webcal only allows one outstanding operation at a time. Comparing the times for a single import to the times for ten imports shows the result. Ten imports uniformly take ten times as long as one import. On a slow network, round

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Webcal** | **Ical** | **Webcal** | **Ical** | **Webcal** | **Ical** |
| Local | | 0.437 | | 1.400 | | 1.490 |
| Ethernet | 0.273 | 0.704 | 0.301 | 2.670 | 0.306 | 2.380 |
| WaveLAN | 0.295 | 0.707 | 0.345 | 2.270 | 0.351 | 2.710 |
| 19.2 Wired CSLIP | 1.313 | 1.430 | 1.361 | 5.540 | 1.409 | 14.100 |
| 9.6 Cellular CSLIP | 5.839 | 4.210 | 5.868 | 16.700 | 4.939 | 50.100 |

Table 5.7: Time to complete the import of one item from a Webcal calendar with a cold cache (i.e., the time from initiating Algorithm ITEM_IMPORT until the completion of Algorithm ROVER_UPDATE) and the time to re-read Ical calendars over NFS. Table shows mean times in seconds.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Webcal** | **Ical** | **Webcal** | **Ical** | **Webcal** | **Ical** |
| Local | | 0.437 | | 1.400 | | 1.490 |
| Ethernet | 2.626 | 0.704 | 4.125 | 2.670 | 4.354 | 2.380 |
| WaveLAN | 3.485 | 0.707 | 4.307 | 2.270 | 5.745 | 2.710 |
| 19.2 Wired CSLIP | 12.692 | 1.430 | 13.156 | 5.540 | 12.651 | 14.100 |
| 9.6 Cellular CSLIP | 55.264 | 4.210 | 57.709 | 16.700 | 53.649 | 50.100 |

Table 5.8: Time to complete the import of ten (10) items from a Webcal calendar with a cold cache (i.e., the time from initiating Algorithm ITEM_IMPORT for the first item until the completion of Algorithm ROVER_UPDATE for the tenth item) and the time to re-read Ical calendars over NFS. Table shows mean times in seconds.

trip latency dominates the cost. Ten exports take less than ten times as long as one because of the overlap of log replay computation with communication. A prototype system for batching QRPCs and encapsulating multiple RDOs in one request is under development and will be of substantial benefit.

Similarly (as mentioned in Section 5.1.1) reducing item and log size as well as protocol overhead will have a substantial impact. Each import causes about 900 bytes to transit the network while each export causes about 2000 bytes to cross. So, by the time ten items cross the network, Webcal is actually sending more data than Ical. Batching will keep QRPC protocol constant, independent of size. Increasing item representation efficiency should allow Webcal to send the whole calendar in slightly more bandwidth than Ical uses.

The conclusion to note from both these cases is that, even without batching, update time is proportional to the size of the changed data not the size of the calendar.

### 5.2.3 Benefits of Migrating RDOs

The final experiment demonstrates the benefits to user interactivity of relocating dynamic objects in an environment with moderate bandwidth links and disconnected operation. The experiment mea-

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Webcal** | **Ical** | **Webcal** | **Ical** | **Webcal** | **Ical** |
| Local | | 0.043 | | 0.066 | | 0.127 |
| Ethernet | 1.872 | 0.265 | 1.934 | 0.486 | 2.043 | 0.932 |
| WaveLAN | 1.933 | 0.325 | 2.279 | 0.487 | 2.015 | 1.220 |
| 19.2 Wired CSLIP | 3.577 | 2.430 | 3.944 | 5.330 | 3.841 | 15.000 |
| 9.6 Cellular CSLIP | 8.772 | 10.200 | 8.733 | 23.100 | 8.702 | 50.000 |

Table 5.9: Time to complete the export of one item from a Webcal calendar(i.e., the time from initiating Algorithm ITEM_EXPORT until the completion of Algorithm ROVER_UPDATE) and the time to save Ical calendars over NFS. Table shows mean times in seconds.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Webcal** | **Ical** | **Webcal** | **Ical** | **Webcal** | **Ical** |
| Local | | 0.043 | | 0.066 | | 0.127 |
| Ethernet | 6.501 | 0.265 | 7.806 | 0.486 | 8.080 | 0.932 |
| WaveLAN | 7.792 | 0.325 | 8.316 | 0.487 | 8.414 | 1.220 |
| 19.2 Wired CSLIP | 23.948 | 2.430 | 22.951 | 5.330 | 23.470 | 15.000 |
| 9.6 Cellular CSLIP | 78.243 | 10.200 | 77.205 | 23.100 | 73.361 | 50.000 |

Table 5.10: Time to complete the export of ten (10) items from a Webcal calendar (i.e., the time from initiating Algorithm ITEM_EXPORT for the first item until the completion of Algorithm ROVER_UPDATE for the last item) and the time to save Ical calendars over NFS. Table shows mean times in seconds.

sured the time to perform a simple task using Ical and Webcal: starting the application and viewing the appointments for a week's activities using the medium calendar. The experiment attempts to show that moving the interface RDO across the network decreases the amount of time a user has to wait for the response from the graphical user interface.

The results for the experiment are summarized in Figure 5-1. The figure shows two cases for Ical. In the X11R6 case, Ical ran unmodified on the server, accessing data locally, while displaying the user interface on the mobile host using X over the network. In the NFS case, it ran on the mobile host using NFS to access data at the server.

Webcal ran locally on the mobile host with the application binary and supporting RDOs locally cached. Thus, the times measured included the time to verify the RDOs representing the user's calendar and the calendar items contained within it. The fully disconnected case measured performance when all information was locally cached and validation requests were enqueued and logged to stable storage for later delivery.

What the numbers do not show is the extreme sluggishness of the user interface when using X with the server running remotely. Scrolling and refreshing operations are extremely slow. Clicking and selecting operations are very difficult to perform because of the lag between mouse clicks and display updates. In contrast, the Webcal interface is fully functional before all the data is loaded.

So, for example it is possible to make new entries while still loading the old. From this experience we conclude that dynamically migrating RDOs (in this case the GUI) delivers substantial user-observed performance benefits.

We then compare Rover Webcal in disconnected mode with the unmodified version of the application running on the server and using X over Ethernet to provide the user interface to the mobile host. When considering this comparison, it is important to recall the relative performance differences between the server (Pentium 120) and the mobile host (ThinkPad 701C), and that Rover logs validation requests. We see that the Rover application's performance is competitive with the unmodified applications (49 versus 14 seconds). Much of this time is spent in IPC round trips between the access manager and application as each RDO is loaded from the cache. Again, batching will increase Webcal performance here. From this experiment, we can conclude that Rover delivers excellent performance in disconnected operation.

If we compare the unmodified applications running over X with the Rover applications over a 9.6Kbit Cellular dial-up line, we see that the Webcal performs better than the unmodified applications using X over cellular (662 versus 345 seconds). Again, the time for Webcal to verify its data includes 50 round trips to validate the calendar items. In addition, Webcal loads approximately 20 separate RDOs to create its interface. While validations of these RDO are allowed to overlap, a substantial benefit would be gained from group validations.

In [21], a previous implementation of Webcal is compared to Ical. That implementation used one RDO for the whole calendar and one RDO for the entire application. We can use this as an approximation of the performance of Webcal over Rover with batched operations. In that paper, Webcal was measured to be twice as fast as Ical over NFS using wired CSLIP and 20 times as fast as Ical over X using cellular CSLIP.

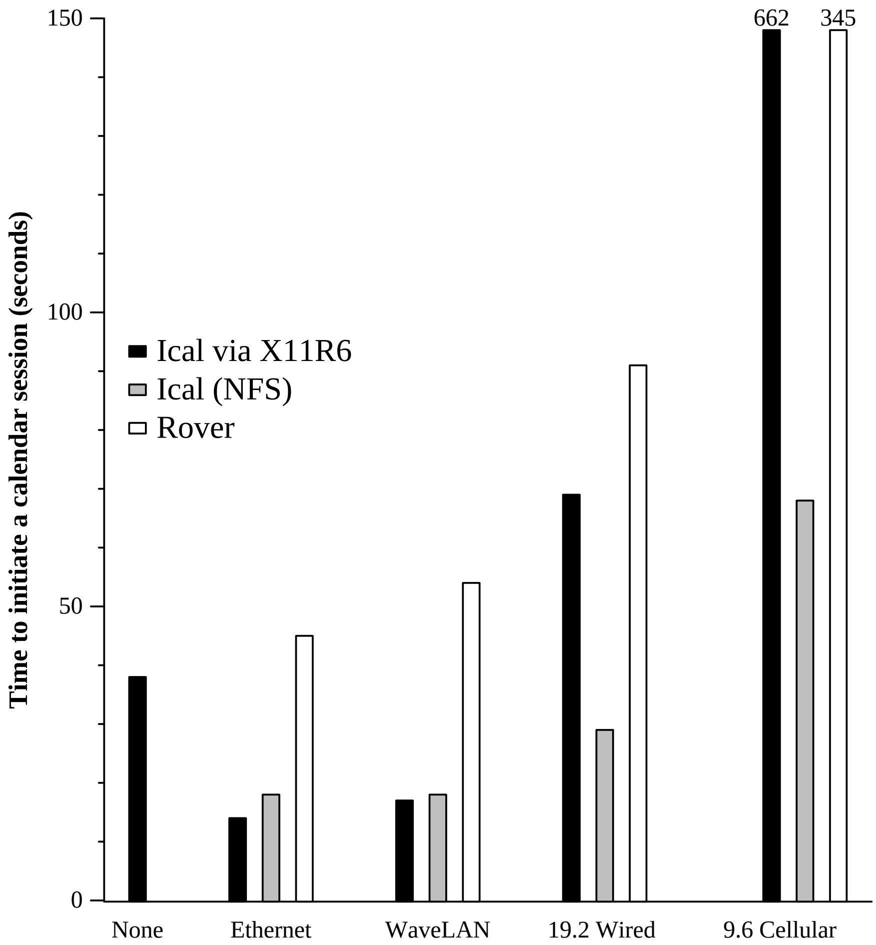


Figure 5-1: Time to initiate a calendar session. The figure shows time required to start the program and read one week's appointments from the medium calendar.

# Chapter 6

# Related Work

Earlier work on Rover introduced the Rover architecture, including both queued RPC and relocatable dynamic objects [21] . 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. deLespinasse studied the mobile-transparent Rover Web proxy[7]. Recent work has leveraged QRPC logs and RDOs to create fault-tolerant applications. [22]

The Coda project pioneered the provision of distributed services for mobile clients. In particular, it investigated how to make file systems run well on mobile computers by using optimistic concurrency control and prefetching [26, 42]. Coda logs all updates to the file system during disconnection and replays the log on reconnection. Coda provides automatic conflict resolution mechanisms for directories, and uses Unix file naming semantics to invoke application-specific conflict resolution programs at the file system level [27]. 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 [34].

The Ficus file system also supports disconnected operation, but relies on version vectors to detect conflicts [39]. The Little Work Project caches files to smooth disconnection from an AFS file system [19]. Conflicts are detected and reported to the user. Little Work is also able to use low bandwidth networks [18]. These projects, as weel as Coda, are focused on hiding mobility from the application.

The Bayou project [8, 45] defines an architecture for sharing data among mobile users. Bayou addresses the issues of tentative data values [47] and session guarantees for weakly-consistent replicated data [44]. To illustrate these concepts, the authors have built a calendar tool and a bibliographic database. Rover borrows the notions of tentative data, log replay, session guarantees, and the calendar tool example from the Bayou project. Rover extends this work with RDOs, QRPC, and events to deal with intermittent communication, limited bandwidth, and resource poor clients. In concurrent work, Gray *et. al.* performs a thorough analysis of the options for database replication in a mobile environment and concludes that primary copy replication with tentative updates is the most appropriate approach for mobile environments [14].

One alternative to the Rover object model is the Thor object model [31]. 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 [15], but it has not yet 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.

The BNU project implements an RPC-driven application framework on mobile computers. It allows for function shipping by downloading Scheme functions for interpretation [52]. Application designers for BNU noted that the workload characterizing mobile platforms is different from workstation environments and will entail distinct approaches to user interfaces [28]. The BNU environment includes proxies on stationary hosts for hiding the mobility of the system. No additional support for disconnected operation, such as Rover's queued RPC, is included in BNU. A follow-up project, Wit, addresses some of these shortcomings and shares many of the goals of Rover, but employs different solutions [51].

RDOs can be viewed as simple Agents [40] or as a light-weight form of process migration [9, 38, 43, 48]. Other forms of code shipping include Display Postscript [1], Safe-Tcl [5], Active Pages [17], Dynamic Documents [23], and LISP Hypermedia [33]. RDOs are probably closest to Telescript [53], Ousterhout's Tcl agents [35], and Java [13]. Most differences between RDOs and these other forms of code shipping are 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 a uniform naming scheme, an application-specific replication model, and QRPC.

The InfoPad project [29] and W4 [3] focus on mobile wireless information access. The InfoPad project employs a dumb terminal, and offloads all functionality from the client to the server. W4 employs a similar approach for accessing the Web from a small PDA. Rover, is designed to be more flexible. Depending on the power of the mobile host and the available bandwidth, Rover dynamically adapts and moves functionality between the client and the server.

A number of proposals have been made for dealing with the limited communication environments for mobile computers. Katz surveys many of the challenges [24]. Baker describes MosquitoNet, which shares similar goals with Rover, but has not been implemented yet [2]. Oracle recently released a product for mobile computers that provides asynchronous communication [10]; unfortunately, details and performance analysis are not available.

A number of successful commercial applications have been developed for mobile hosts and limited-bandwidth channels. For example, Qualcomm's Eudora is a mail browser that allows efficient remote access over low-bandwidth links. Lotus Notes [25] is a groupware application that allows users to share data in a weakly-connected environment. Notes supports two forms of update operations discussed in Section 3.5: append and time-stamped. The Rover toolkit and its applications provide functionality that is similar to these proprietary approaches and it does this in an application-independent manner. Using the Rover toolkit, standard workstation applications, such as *Exmh* and *Ical*, can be easily turned into mobile-aware applications.

The commercial group calendar/scheduler market is robust. A number of companies offer enterprise-wide scheduling tools, including TimeVision, Meeting Maker, OnTime, Organizer, and CaLANdar. Most of these tools rely on connected operation to keep calendars up to date. Several support some form of store and forward "synchronization" of calendars, generally through e-mail. A few support disconnected operation of some sort, allowing a mobile user to download portions of a calendar for off-line use. Unfortunately, all use proprietary technology, and it is difficult to tell from the literature what semantics are implemented. The proliferation of proprietary "standards" provides all the more support to the argument that a common toolkit would be useful to application builders.

The DeckScape WWW browser [6] 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.

Several systems use E-mail messages as a transport medium, and obtain similar benefits as we obtain by using QRPC. The Active Message Processing project [49] has developed various applications, including a distributed calendar, which use E-mail messages as a transport medium. As another example, researchers at 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 [30]. This application is a centralized, client-server system with one server at DEC SRC that automatically dispatches tasks and collects results.

The research described in this thesis borrows from early work on replication for non-mobile distributed systems. In particular, we borrow from Locus [50] (type-specific conflict resolving) and Cedar [12] (check-in, check-out model of data sharing).

# Chapter 7

# Conclusion

Mobile-aware applications are best suited to face the unique set of challenges faced by mobile computers. Mobile-aware applications can excel even in the absence of high speed network connections. The Rover Toolkit supports mobile-aware applications through an interface built on the communication and programming abstractions of queued remote procedure call, relocatable dynamic objects, and events.

Interactive performance of Rover applications is decoupled from slow networks and nearly equivalent across three orders of magnitude of network bandwidths and latency. Relocatable dynamic objects allow mobile-aware applications to locate code together with data to increase resource utilization efficiency. Transporting logs of small operations on objects allows applications to update data in time proportional to the amount of the data changed, rather than the total size application data. Rover applications select the level of consistency control appropriate to the data. Rover optimizes optimistic concurrency schemes but does not impose any.

In summary, the Rover Mobile Application Toolkit enables mobile-aware applications to have the information and control necessary to adapt to the rigors of mobile computing.

## Appendix A

# Complete Data

This appendix contains complete data for the tables and graphs shown in Chapter 5.

| **Transport** | **Small Calendar** | | **Med Calendar** | | **Large Calendar** | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Local | 0.437 | 0.072 | 1.40 | 0.65 | 1.49 | 0.13 |
| Ethernet | 0.704 | 0.039 | 2.67 | 0.10 | 2.38 | 0.67 |
| WaveLAN | 0.707 | 0.39 | 2.27 | 0.21 | 2.71 | 0.27 |
| 19.2 Wired CSLIP | 1.43 | 0.66 | 5.54 | 0.43 | 14.1 | 0.84 |
| 9.6 Cellular CSLIP | 4.21 | 0.94 | 16.7 | 0.67 | 50.1 | 2.2 |

Table A.1: Time to re-read Ical calendar files using NFS over various transports. Re-reads include the computation to delete old items and include new ones. Thus, times for re-reads may exceed the times for equivalent saves. Table shows mean times, in seconds, and standard deviations.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Local | 0.0430 | 0.0054 | 0.0656 | 0.0065 | 0.127 | 0.0020 |
| Ethernet | 0.265 | 0.013 | 0.486 | 0.011 | 0.932 | 0.047 |
| WaveLAN | 0.325 | 0.13 | 0.487 | 0.019 | 1.22 | 0.037 |
| 19.2 Wired CSLIP | 2.43 | 0.029 | 5.33 | 0.13 | 15.0 | 0.55 |
| 9.6 Cellular CSLIP | 10.2 | 1.64 | 23.1 | 1.61 | 50.0 | 2.06 |

Table A.2: Time to save Ical calendar files using NFS over various transports. Table shows mean times, in seconds, and standard deviations.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Ethernet | 0.045 | 0.015 | 0.050 | 0.014 | 0.055 | 0.004 |
| WaveLAN | 0.049 | 0.016 | 0.048 | 0.016 | 0.054 | 0.010 |
| 19.2 Wired CSLIP | 0.047 | 0.010 | 0.043 | 0.013 | 0.049 | 0.004 |
| 9.6 Cellular CSLIP | 0.044 | 0.012 | 0.046 | 0.010 | 0.049 | 0.003 |

Table A.3: Time to initiate the import of one item from a Webcal calendar i.e, the time to perform Algorithm ITEM_IMPORT. Table shows mean times, in seconds, and standard deviations.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Ethernet | 0.244 | 0.006 | 0.285 | 0.106 | 0.280 | 0.088 |
| WaveLAN | 0.250 | 0.003 | 0.249 | 0.018 | 0.257 | 0.003 |
| 19.2 Wired CSLIP | 0.230 | 0.005 | 0.230 | 0.002 | 0.259 | 0.037 |
| 9.6 Cellular CSLIP | 0.243 | 0.003 | 0.252 | 0.025 | 0.267 | 0.043 |

Table A.4: Time to initiate the export of one item from a Webcal calendar, i.e., the time to perform Algorithm ITEM_EXPORT Table shows mean times, in seconds, and standard deviations.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Ethernet | 0.273 | 0.023 | 0.301 | 0.023 | 0.306 | 0.009 |
| WaveLAN | 0.295 | 0.023 | 0.345 | 0.074 | 0.351 | 0.041 |
| 19.2 Wired CSLIP | 1.313 | 0.103 | 1.361 | 0.089 | 1.409 | 0.163 |
| 9.6 Cellular CSLIP | 5.839 | 0.745 | 5.868 | 0.828 | 4.939 | 0.338 |

Table A.5: Time to complete the import of one item from a Webcal calendar with a cold cache, i.e., the time from initiating Algorithm ITEM_IMPORT until the completion of Algorithm ROVER_UPDATE. Table shows mean times, in seconds, and standard deviations.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Ethernet | 2.626 | 0.080 | 4.125 | 0.187 | 4.354 | 0.125 |
| WaveLAN | 3.485 | 0.265 | 4.307 | 0.163 | 5.745 | 1.973 |
| 19.2 Wired CSLIP | 12.692 | 1.198 | 13.156 | 1.608 | 12.651 | 0.642 |
| 9.6 Cellular CSLIP | 55.264 | 3.948 | 57.709 | 3.750 | 53.649 | 3.035 |

Table A.6: Time to complete the import of ten (10) items from a Webcal calendar with a cold cache, i.e., the time from initiating Algorithm ITEM_IMPORT for the first item until the completion of Algorithm ROVER_UPDATE for the tenth item. Table shows mean times, in seconds, and standard deviations.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Ethernet | 0.340 | 0.056 | 0.344 | 0.031 | 0.372 | 0.089 |
| WaveLAN | 0.360 | 0.065 | 0.350 | 0.033 | 0.366 | 0.059 |
| 19.2 Wired CSLIP | 1.054 | 0.034 | 1.065 | 0.029 | 1.137 | 0.065 |
| 9.6 Cellular CSLIP | 4.275 | 0.105 | 4.175 | 0.181 | 4.005 | 0.117 |

Table A.7: Time to complete the import of one item from a Webcal calendar with a warm cache, i.e., the time from initiating Algorithm ITEM_IMPORT until the completion of Algorithm ROVER_UPDATE. Table shows mean times, in seconds, and standard deviations.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Ethernet | 2.646 | 0.142 | 4.066 | 0.156 | 4.169 | 0.100 |
| WaveLAN | 2.972 | 0.620 | 4.847 | 1.084 | 4.563 | 0.670 |
| 19.2 Wired CSLIP | 9.734 | 0.898 | 10.315 | 2.621 | 9.833 | 0.899 |
| 9.6 Cellular CSLIP | 49.424 | 6.732 | 48.658 | 8.439 | 39.784 | 0.217 |

Table A.8: Time to complete the import of ten (10) items from a Webcal calendar with a warm cache, i.e., the time from initiating Algorithm ITEM_IMPORT for the first item until the completion of Algorithm ROVER_UPDATE for the tenth item. Table shows mean times, in seconds, and standard deviations.

| Transport | Small Calendar | | Med Calendar | | Large Calendar | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Ethernet | 1.872 | 0.099 | 1.934 | 0.114 | 2.043 | 0.119 |
| WaveLAN | 1.933 | 0.053 | 2.279 | 0.887 | 2.015 | 0.096 |
| 19.2 Wired CSLIP | 3.577 | 0.173 | 3.944 | 0.897 | 3.841 | 0.522 |
| 9.6 Cellular CSLIP | 8.772 | 0.175 | 8.733 | 0.196 | 8.702 | 0.643 |

Table A.9: Time to complete the export of one item from a Webcal calendar, i.e., the time from initiating Algorithm ITEM_EXPORT until the completion of Algorithm ROVER_UPDATE. Table shows mean times, in seconds, and standard deviations.

| **Transport** | **Small Calendar** | | **Med Calendar** | | **Large Calendar** | |
|---|---|---|---|---|---|---|
| | **Mean** | $\sigma$ | **Mean** | $\sigma$ | **Mean** | $\sigma$ |
| Ethernet | 6.501 | 0.236 | 7.806 | 0.333 | 8.080 | 0.135 |
| WaveLAN | 7.792 | 1.381 | 8.316 | 0.700 | 8.414 | 0.853 |
| 19.2 Wired CSLIP | 23.948 | 1.511 | 22.951 | 1.238 | 23.470 | 1.549 |
| 9.6 Cellular CSLIP | 78.243 | 3.341 | 77.205 | 4.674 | 73.361 | 2.049 |

Table A.10: Time to complete the export of ten (10) items from a Webcal calendar, i.e., the time from initiating Algorithm ITEM_EXPORT for the first item until the completion of Algorithm ROVER_UPDATE for the last item. Table shows mean times, in seconds, and standard deviations.

| **Configuration** | **Transport** | **Time** |
|---|---|---|
| Ical via X11R6 | Ethernet | 0:14 |
| Ical (NFS) | Ethernet | 0:18 |
| Rover | Ethernet | 0:45 |
| Ical via X11R6 | WaveLAN | 0:17 |
| Ical (NFS) | WaveLAN | 0:18 |
| Rover | WaveLAN | 0:54 |
| Ical via X11R6 | 19.2 Wired CSLIP | 1:09 |
| Ical (NFS) | 19.2 Wired CSLIP | 0:29 |
| Rover | 19.2 Wired CSLIP | 1:31 |
| Ical via X11R6 | 9.6 Cellular CSLIP | 11:02 |
| Ical (NFS) | 9.6 Cellular CSLIP | 1.08 |
| Rover | 9.6 Cellular CSLIP | 5:45 |
| Rover | None | 0:38 |

Table A.11: Time to initiate a calendar session. The table shows time required to start the program and read one week's appointments from the medium calendar. Table shows time as minute:seconds.

# Bibliography


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

[2] M.G. Baker. Changing communication environments in MosquitoNet. In *Workshop on Mobile Computing Systems and Applications*, pages 64–68, Santa Cruz, CA, 1994.

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

[4] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Trans. Comp. Syst.*, 2(1):39–59, Feb. 1984.

[5] 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.

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

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

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

[9] 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. IEEE.

[10] A. Downing, D. Daniels, G. Hallmark, K. Jacobs, and S. Jain. Oracle 7, symmetric replication: Asynchronous distributed technology, September 1993.

[11] Sanjay Ghemawat. Ical, 1993. http://www.pmg.lcs.edu/ sanjay/ical_html.

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

[13] J. Gosling and H. McGilton. The Java language environment: A white paper, 1995. http://-java.sun.com/whitePaper/javawhitepaper_1.html.

[14] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *To appear in Proc. of the 1996 SIGMOD Conference*, June 1996.

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

[16] P. Honeyman, L. Huston, J. Rees, et al. The LITTLE WORK project. In *Proc. of the 3rd Workshop on Workstations Operating Systems*. IEEE, April 1992.

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

[18] 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.

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

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

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

[22] A. Joseph and F. Kaashoek. Building fault-tolerant mobile-aware applications using the rover toolkit. Submitted for Publication, May 1996.

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

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

[25] 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.

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

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

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

[29] M.T. Le, F. Burghardt, S. Seshan, and J. Rabaey. InfoNet: the networking infrastructure of InfoPad. In *Compcon '95*, pages 163–168, 1995.

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

[31] 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, 1993.

[32] L.Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 32(7), 1978.

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

[34] 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)*, Copper Mountain Resort, CO, December 1995.

[35] J.K. Ousterhout. The Tcl/Tk project at Sun Labs, 1995. http://www.sunlabs.com/research/tcl.

[36] J. Postel. *Internet Name Server.* IEN-116, August 1979.

[37] J. B. Postel. *Simple Mail Transfer Protocol.* Internet RFC 821, August 1982.

[38] 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.

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

[40] D. Riecken, editor. *Intelligent Agents*, volume 37. Communications of the ACM, July 1994.

[41] 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.

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

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

[44] 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.

[45] 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)*, Copper Mountain Resort, CO, 1995.

[46] D. B. Terry *et. al.*. Managing update conflicts in a weakly connected replicated storage system. In *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, CO, 1995.

[47] 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, CA, 1994.

[48] 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, WA, December 1985.

[49] 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.

[50] 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.

[51] T. Watson. Application design for wireless computing. In *Workshop on Mobile Computing Systems and Applications*, pages 91–94, Santa Cruz, CA, 1994.

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

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