

# Mobile Computing with the Rover Toolkit

Anthony D. Joseph, *Student Member, IEEE*, Joshua A. Tauber, *Student Member, IEEE*  
and M. Frans Kaashoek, *Member, IEEE*

**Abstract**—Rover is a software toolkit that supports the construction of both *mobile-transparent* and *mobile-aware* applications. The mobile-transparent approach aims to enable existing applications to run in a mobile environment without alteration. This transparency is achieved by developing proxies for system services that hide the mobile characteristics of the environment from applications. However, to excel, applications operating in the harsh conditions of a mobile environment must often be aware of and actively adapt to those conditions. Using the programming and communication abstractions present in the Rover toolkit, applications obtain increased availability, concurrency, resource allocation efficiency, fault tolerance, consistency, and adaptation. Experimental evaluation of a suite of mobile applications demonstrates that use of the toolkit requires relatively little programming overhead, allows correct operation, substantially increases interactive performance, and dramatically reduces network utilization.

**Index Terms**—Mobile computing, distributed objects, mobile code, wireless networks, software libraries, distributed systems, replication, caching, operating systems, remote procedure call.

## 1 INTRODUCTION

MOBILE computing environments present application designers with a unique set of communication and data integrity constraints that are absent in traditional distributed computing settings. For example, although mobile communication infrastructures are becoming more common, network bandwidth in mobile environments is often severely limited, unavailable, or expensive. Therefore, system facilities in mobile computing environments should minimize dependence upon continuous network connectivity, provide tools to optimize the utilization of available network bandwidth, minimize dependence on data stored on remote servers, and allow dynamic division of work between clients and servers. In this paper, we describe the *Rover toolkit*, a set of software tools that supports applications that operate obliviously to the underlying environment, while also enabling the construction of applications that use awareness of the mobile environment to adapt to its limitations. We illustrate the effectiveness of the toolkit using a number of distributed applications, each of which runs well over networks that differ by three orders of magnitude in bandwidth and latency.

### 1.1 Mobile versus Stationary Environment

Designers of applications for mobile environments must address several differences between the mobile environment and the stationary environment. Issues that represent minor inconveniences in stationary distributed systems are significant problems for mobile computers. This distinction requires a rethinking of the classical distributed systems techniques normally used in stationary environments.

Computers in a stationary environment are usually very reliable. Relative to their stationary counterparts, mobile

computers are quite fragile: A mobile computer may run out of battery power, be damaged in a fall, be lost, or be stolen. As pointed out by the designers of Coda, given these threats, primary ownership of data should reside with stationary computers rather than with mobile computers [1]. Furthermore, application designers should take special precautions to enhance the resilience of the data stored on mobile computers.

Relative to most stationary computers, a mobile computer often has fewer computational resources available. In addition, the available resources may change dynamically (e.g., a “docked” mobile computer has access to a larger display, auxiliary graphics or math coprocessors, additional stable storage, unlimited power, etc.).

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

Network partitions are an infrequent occurrence in stationary networks, and most applications consider them to be failures that are exposed to users. In the mobile environment, applications will face frequent, lengthy network partitions. Some of these partitions will be involuntary (e.g., due to a lack of network coverage), while others will be voluntary (e.g., due to a high dollar cost). Mobile applications should handle such partitions gracefully and as transparently as possible. In addition, users should be able, as far as possible, to continue working as if the network was still available. In particular, users should be able to modify local copies of global data.

When users modify local copies of global data, consistency becomes an issue. In a mobile environment, optimistic concurrency control [2] is useful because pessimistic methods are inappropriate (a disconnected user cannot ac-

• The authors are with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.  
E-mail: {adj, josh, kaashoek}@lcs.mit.edu.

For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number C97027.

quire or release locks), as pointed out by the designers of Coda [1]. However, using an optimistic approach does involve some difficulties. In particular, long duration partitions will cause a greater incidence of apparent write-write conflicts than in stationary environments. It is therefore important to use application-specific semantic information to detect when such conflicts are false positives and can be avoided.

## 1.2 The Argument for Mobile-Aware Computing

The attributes of the typical stationary environment have guided the development of classical distributed computing techniques for building client-server applications. These applications are usually unaware of the actual state of the environment; therefore, they make certain implicit assumptions about the location and availability of resources.

Such *mobile-transparent* applications can be used unmodified in mobile environments by having the system shield or hide the differences between the stationary and mobile environments from applications. Coda [1] and Little Work [3] use this approach in providing a file system interface to applications. These systems consist of a local *proxy* for some service (the file system) running on the mobile host and providing the standard service interface to the application, while attempting to mitigate any adverse effects of the mobile environment. The proxy on the mobile host cooperates with a remote server running on a well-connected, stationary host.

This mobile-transparent approach simplifies mobile applications, but sacrifices functionality and performance. Although the system hides mobility issues from the application, it usually requires manual intervention by the user (i.e., having the user indicate which data to prefetch onto the user's computer). Similarly, conflict resolution is complicated because the interface between the application and its data was designed for a stationary environment. Consider an application writing records into a file shared among stationary and mobile hosts. While disconnected, the application on the mobile host inserts a new record. The local file system proxy records the write in a log. Meanwhile, an application on a stationary host alters another record in the same file. Upon reconnection, the file system can detect that conflicting updates have occurred: One file contains a record that does not appear in the other file and one file contains a record that differs from the same record in the other file. Depending upon how records are named, it may not be possible to distinguish between the instance where the same record is modified in one file and not the other and the instance where a record is deleted and a new one is added. Thus, the file system alone cannot resolve the conflict.

The cause of this ambiguity is that these systems change the contract between the application and 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 semantics of the standard read/write interface for synchronization and ordering may fail.

Coda recognizes this limitation in conflict resolution and provides support for *application-specific resolvers* (ASRs) [4],

programs that are invoked based on the file name extension to resolve conflicts during synchronization. However, ASRs alone are insufficient, because the only information provided to an ASR is the two conflicting files, not the two operations and contexts that led to the differing results. An application has no way of knowing that the file system contract has been changed by Coda, and, therefore, no way to save the additional information the ASR would need to resolve the conflict. Thus, in the above example, there is no way for an 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.

So, although the mobile-transparent approach is appealing in that it offers to run existing applications without alteration, it is fundamentally limited in that it hides the information needed to allow applications to remain correct and to perform well in an intermittently connected environment. The alternative to hiding environmental information from applications is to expose information and involve applications and users in decision making. This alternative yields the class of *mobile-aware* applications.

In the above case, a mobile-aware application can store not only the *value* of a write, but also the *operation* associated with the write. That operation can include any relevant context. Storing the operation allows the application to use application-specific semantic and contextual information; for example, it allows for "on-the-fly" dynamic construction of conflict avoidance and conflict resolution procedures.

The mobile-aware argument can be viewed as applying the end-to-end argument ("Communication functionality can be implemented only with the knowledge and help of the application standing at the endpoints of the communications system." [5]) to mobile applications. The file system example described above illustrates that some applications need to be aware of intermittent network connectivity to achieve consistency. Similar examples exist for performance, reliability, low-power operation, etc.

The mobile-aware argument does not require that every application use its own, ad hoc approach to mobile computing. Rather, it allows the underlying communication and programming systems to define an application programming interface that optimizes common cases and supports the transfer of appropriate information between the layers. Since mobile-aware applications share common design goals, they can share design features and techniques. The Rover toolkit provides such a mobile-aware application programming interface that, unlike previous systems, is designed to support both mobile-aware and mobile-transparent approaches.

## 1.3 Rover: The Toolkit Approach

The Rover toolkit offers applications a distributed object system based on a client/server architecture with client caching and optimistic concurrency control [6] (see Fig. 1). Clients are Rover applications that typically run on mobile hosts, but could run on stationary hosts as well. Servers, which may be replicated, typically run on stationary hosts and hold the long-term state of the system.

The toolkit provides mobile communication support based on two ideas: *relocatable dynamic objects* (RDOs) and

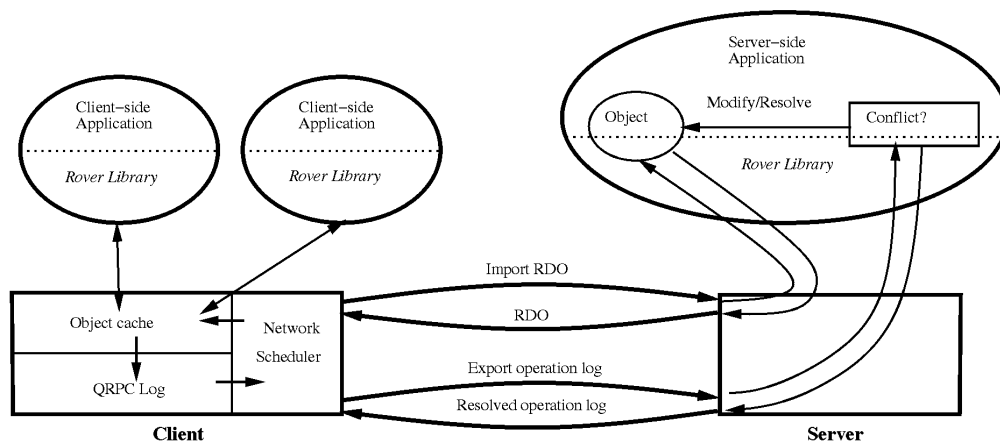


Fig. 1. Rover offers applications a client/server distributed object system with client caching and optimistic concurrency control. This figure shows the control flow within the toolkit.

*queued remote procedure call* (QRPC). A relocatable dynamic object is an object (code and data) with a well-defined interface that can be dynamically loaded into a client computer from a server computer, or vice versa, to reduce client-server communication requirements. Queued remote procedure call is a communication system that permits applications to continue to make nonblocking remote procedure calls [7] even when a host is disconnected—requests and responses are exchanged upon network reconnection.

Rover applications employ an optimistic, primary-copy, tentative-update, distributed object model of data sharing: They call into the Rover library to *import* RDOs and to *export* logs of operations that mutate RDOs. Client-side applications *invoke* operations directly on locally cached RDOs. Server-side applications are responsible for resolving conflicts and notifying clients of resolutions. A network scheduler (operating above the operating system level) drains the stable QRPC log, which contains the RPCs that must still be performed at the server. Where possible, in addition to opening and closing network connections, the network scheduler also interacts with the operating system to initiate or terminate connection-based transport links (e.g., dial-up) or to power-up/power-down wireless transmitters.

Servers can run either at stationary hosts or at mobile hosts. Running servers at mobile hosts is useful when forming “mobile workgroups,” where a group of clients isolated from other hosts can interact with local servers. However, for two reasons, we mostly use stationary hosts: First, an operation cannot be committed (i.e., it is tentative) until it reaches and is accepted by a server; second, a server’s copy of an object is the canonical version of the object. Thus, clients can ascertain the latest value for an object only by contacting the server that owns the object. Both commit and import operations require well-connected servers. If servers are on poorly connected mobile hosts, clients outside of the workgroup may encounter delays in committing operations or importing objects.

The key task of the programmer when building a mobile-aware application with Rover is to define RDOs for the data types manipulated by the application and for data transported between clients and servers. The programmer then divides the program into portions that run on the cli-

ent and portions that run on the server; these parts communicate by means of QRPC. The programmer then defines methods that update objects, including code for conflict detection, avoidance, and resolution.

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

#### 1.4 Main Results

This paper extends previous work on the Rover architecture [6], [8] by providing a more detailed discussion of the design and implementation of queued RPC and relocatable dynamic objects, including compressed and batched QRPCs; presenting in greater depth the argument for making applications mobile-aware; and explaining how applications use that awareness and the Rover toolkit to mitigate the effects of intermittent communication on application performance. We draw four main conclusions from our experimental data and experience developing Rover:

- 1) QRPC is well suited to intermittently connected environments. Queuing enables RPCs to be scheduled, batched, and compressed for increased network performance. QRPC performance is acceptable in the target environment even if every RPC is stored in stable logs at clients and servers. For lower bandwidth networks, the overhead of writing the logs is dwarfed by the underlying communication costs.
- 2) Use of RDOs allows mobile-aware applications to migrate functionality dynamically to either side of a slow network connection and thereby minimize reliance on network connectivity. Caching RDOs reduces latency and bandwidth consumption. User interface functionality can run at full speed on a mobile host, while large data manipulations can be performed on the well-connected server.
- 3) Porting applications to Rover and making them mobile-aware generally requires relatively little change

to the original application. In addition, building Rover proxies is easy and has allowed the use of applications (e.g., Netscape and USENET browsers) without modification. We have implemented several mobile-aware applications (Rover Exmh, Webcal) by modifying conventional instances of the same applications. We have built others from scratch (Irolo, Stock Market Watcher). Most were made mobile-aware with a few simple changes (to approximately 10% of the original code and with as little as three weeks of work). Only one of the applications required more than a month of work. In part, the extra time was due to additional evolution and redesign of the toolkit.

- 4) Measurements of end-to-end mobile application performance show that, by using Rover, mobile-transparent and mobile-aware applications perform significantly better on slow networks than their original versions. For example, when performing routine Web browsing using Netscape with the mobile-transparent Rover HTTP proxy, we observe performance improvements of up to 17% over Netscape alone. When using mobile-aware applications running over slow networks to perform routine tasks, we observe performance improvements of up to a factor of 7.5 over the original versions.
- 5) We also observe significant improvements in the performance of the user interfaces of client/server-based mobile-aware applications in comparison to those of conventional applications used remotely across slow networks. Without Rover, scrolling and refreshing operations are extremely slow and pressing buttons and selecting text are unpleasant because of the lag between mouse clicks and display updates. With Rover, the user sees excellent GUI performance across a range of networks that varies by three orders of magnitude in both bandwidth and latency.

## 2 RELATED WORK

The need for mobile-aware applications and complementary system services to expose mobility to applications was identified concurrently by several groups. Katz noted the need for adaptation of mobile systems to a variety of networking environments [9]. Davies et al. cited the need for protocols to provide feedback about the network to applications in a vertically integrated application environment [10]. Kaashoek et al. created a Web browser that exposed the mobile environment to code implementing mobile-aware Web pages [11]. The Bayou project proposed and implemented an architecture for mobile-aware databases [12]. Baker identified the dichotomy between mobile-awareness and mobile-transparency in general application and system design [13]. However, Rover is the first implemented general application-architecture to support both mobile-transparent system service proxies and mobile-aware applications.

Several previous projects have studied building mobile-transparent services for mobile clients. The Coda project pioneered distributed services for mobile clients. In par-

ticular, it investigated how to build a mobile-transparent file system proxy for mobile computers by using optimistic concurrency control and prefetching [1]. Coda logs all updates to the file system during disconnection and replays the log on reconnection; automatic conflict resolution mechanisms are provided for directories and files, using Unix file naming semantics to invoke ASRs at the file system level [4]. A manual repair tool is provided for conflicts that cannot be resolved automatically. A newer version of Coda supports low-bandwidth networks as well as intermittent communication [14]. Odyssey is a mobile-aware follow-on project that focuses on system support to enable mobile-aware applications to use "data fidelity" to control resource utilization. Data fidelity is defined as the degree to which a copy of data matches the original [15].

The Ficus file system is a mobile-transparent, user-level file system supporting disconnected operation and peer-to-peer data sharing [16]. The Little Work project caches files to smooth disconnection from an AFS file system. Conflicts are detected and reported to the user. Little Work is also able to support partial connection over low-bandwidth networks [3].

The BNU project implements an RPC-driven mobile-transparent application framework on mobile computers. It enables code shipping by downloading Scheme functions for interpretation [17]. The BNU environment includes mobile-transparent proxies on stationary hosts for hiding the mobility of the system. BNU applications do not dynamically adjust to the environment, nor do they have a concept of tentative or stale data. 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 [18].

A number of proposals have been made for various degrees of mobile awareness in operating system services and applications. The Bayou project [12] defines a mobile-aware database architecture for sharing data among mobile users. Bayou supports tentative operation logs and tentative data values combined with session guarantees for weakly consistent replicated data [19]. Each database write is associated with a dependency check and a deterministic, application-specific merge procedure. Each Bayou host replicates the entire database and may receive incremental updates from any other host. To illustrate these concepts, the authors built a calendar tool and a bibliographic database. Rover shares the notions of tentative operations and data, session guarantees, and the calendar tool example with the Bayou project. Rover's RDO method invocation via QRPC is more general than Bayou database writes in that it supports computation relocation and communication scheduling for dealing with intermittent communication, limited bandwidth, and resource-poor clients.

The DATAMAN project organized a framework for mobile-aware applications around an object-oriented, language-level event delivery architecture [20]. Applications interested in events in the environment declare EventObjects, which support both polling and triggers (callbacks). The DATAMAN application framework does not include support for data replication and consistency management.

The InfoPad [21], Daedalus [22], GloMop [23], and W4 [24] projects focus on mobile-aware wireless information access. The InfoPad project employs a dumb terminal and offloads all functionality from the client to the server with the specific goal of reducing power consumption. Daedalus and GloMop use dynamic “transcoding” or “distillation” to reduce the bandwidth consumed by data transmitted to a mobile host. Transcoding technology is completely compatible with Rover’s architecture. Applications on the mobile host cooperate with mobile-aware proxies on a stationary host to define the characteristics of the desired network connections. Similarly, to enable Web browsing, W4 divides application functionality between a small PDA and a more powerful, stationary host. Rover is designed for more flexible, dynamic divisions. Depending on the power of the mobile host and available bandwidth, Rover allows mobile-aware browsers to move functionality between the client and the server dynamically.

The DeckScape WWW browser [25] modifies the user interface of a Web browser, turning it into a “click-ahead” browser. DeckScape was developed simultaneously with the Rover HTTP proxy which transparently provides click-ahead for existing browsers. Since they implemented a browser from scratch, DeckScape is not compatible with existing browsers.

The BARWAN project addresses the problems associated with “overlay networks,” networks of varying bandwidth, latency, coverage, and application-level performance visibility [26]. BARWAN supports mobile, “data type aware” applications by using proxies on both sides of the client-server link to monitor the performance attributes of the network. The monitoring, in conjunction with mobile code and a dynamically extensible type system, enables mobile-aware applications to trade processing cycles for network traffic. Similarly, Davies’s Adaptive Services [10] uses a protocol-centric approach to expose information about the mobile environment to applications. In contrast, Rover is designed to focus on dynamic adaptation of program functionality and data types.

A number of successful commercial mobile-aware 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 [27] is a groupware application allowing users to share data in a weakly connected environment. Notes supports two forms of update operations: append and time-stamped. Conflicts are referred to the user. TimeVision and Meeting Maker are group calendar tools that allow a mobile user to download portions of a calendar for off-line use. The Rover toolkit and its applications provide functionality that is similar to these proprietary approaches, but in an application-independent manner. Using the Rover toolkit, standard workstation applications such as *Exmh* and *Ical* can easily be turned into mobile-aware applications.

Gray et al. perform a thorough theoretical analysis of the options for database replication in a mobile environment and conclude that primary-copy replication with tentative updates is the most appropriate approach for mobile environments [28].

### 3 DESIGN OF THE ROVER TOOLKIT

The Rover toolkit is designed to support the construction of mobile-aware applications and proxies for mobile-transparent applications. In this section, we describe the design of the key components of the Rover toolkit. In the following section, we discuss implementation details for each component of the toolkit.

#### 3.1 Object Design and QRPC

The central structures in Rover are relocatable dynamic objects (RDOs). All Rover design decisions are based upon RDOs; thus, they provide the key point of control in Rover applications. RDOs have four components: mobile code, encapsulated data, a well-defined interface, and the ability to make outcalls to other RDOs. The mobile code and encapsulated data components make an RDO relocatable from one machine to another. Each RDO provides methods for marshaling and unmarshaling itself so that it can be relocated to another machine and stored in a client’s cache. An RDO can invoke the methods of another RDO by using that RDO’s well-defined interface.

RDOs may execute at either clients or servers. Each RDO has a “home” server that maintains its primary, canonical copy. Clients use library functions provided by the Rover toolkit to import secondary copies of RDOs into their local caches and to export logs of mutating method invocations back to servers.

All application code and all application-touched data are contained within RDOs. RDOs may vary in complexity from simple data items with a small set of operations (e.g., calendar entries) to modules that encapsulate a significant part of an application (e.g., the graphical user interface for an e-mail browser). The toolkit provides functions that enable complex RDOs optionally to create threads of control when they are imported. The toolkit ensures the safe execution of RDOs by using authentication and by executing RDOs in a controlled environment. For our research prototype, we assume that RDOs may be faulty, but not malicious. Thus, the toolkit executes RDOs in separate address spaces, but does not rely on code inspection or other techniques to detect malicious code. These safety measures are appropriate for the sharing of objects between mobile hosts and servers in the framework of specific applications. However, there are several safety issues relating to the general use of mobile code that are not addressed by our current implementation. These safety issues represent an area of active research that is beyond the scope of this paper.

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

A QRPC may contain an RDO, a request for an RDO, or arbitrary application data. Rover clients use QRPC to fetch RDOs from servers lazily (see Fig. 1). When an application

issues a QRPC, Rover stores the QRPC in a local stable log and immediately returns control to the application. If the application has registered a *callback* routine, it will be invoked when the requested QRPC completes. Applications may poll the state of the QRPC or simply block to wait for critical data. However, this choice is undesirable, especially when the mobile host is disconnected. Upon reconnection, the Rover network scheduler drains the log in the background, forwarding any queued QRPCs to the server.

When a Rover application modifies a locally cached RDO, the cached copy is marked *tentative*. The RDO is only marked *committed* after the client knows that the Rover server has applied the mutating operations to the canonical RDO. These updates and resolutions are lazily propagated between the client and server using QRPC. In the meantime, the application may choose to use tentative RDOs. This choice allows the application to continue execution even if the mobile host is disconnected. Cached copies of the RDO at other clients are updated either by client polling or server callbacks.

### 3.2 Communication Scheduling

The Rover network scheduler may deliver QRPCs out of order (i.e., non-FIFO) based on consistency requirements and application-specified operation priorities. Reordering is important for usability in an environment with intermittent connectivity, as it allows the user, through applications, to identify important operations. For example, a user may choose to send urgent updates as soon as possible while delaying other sends until inexpensive communication is available.

Multiple QRPCs for the same server may be batched together, reducing communication overhead. If the operating system provides the appropriate interface, the network scheduler can use batching to reduce the amount of time that a wireless transmitter is powered up but idle.

QRPC supports split-phase operation; if a mobile host is disconnected between sending the request and receiving the reply, a Rover server will periodically attempt to contact the mobile host and deliver the reply. The split-phase communication model enables Rover to use different communication channels for the request and the response and to close channels during the intervening period. Several wired and wireless technologies offer asymmetric communication options, such as cable television modems, direct broadcast satellites, receive-only pagers, and PCS phones that can initiate calls but cannot receive them. By splitting the request and response pair, communication can be directed over the most efficient, available channel. Closing the channel while waiting is particularly useful when the waiting period is long and the client must pay for connection time.

The combination of the split-phase capability and the stable nature of QRPCs allows a mobile host to be completely powered-down while waiting for a pending operation to complete. When the mobile host resumes normal operation, the results of the RDO invocation will be relayed reliably from the server and any unsent messages at the mobile host will be delivered to the server. Thus, long-lived computation (e.g., a database search) can occur at the server while the mobile host conserves power.

### 3.3 Computation Relocation

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

For example, migrating a Graphical User Interface (GUI) to the client serves both these purposes. The code to implement a GUI is small compared to the graphical display updates it generates. At the same time, the GUI together with the application's RDOs can process user actions locally, avoiding additional network traffic and enabling disconnected operation.

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

### 3.4 Notification

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

Applications can forward notifications to users or use them for silent policy changes. For example, in our calendar application (see Section 5.3.2), appointments that have been modified but not propagated to the server are displayed in a distinctive color, a technique that was borrowed from the Bayou room scheduling tool [12]. This color scheme informs users that the appointment might be canceled due to a conflict.

### 3.5 Object Caching and Consistency

An essential component to accomplishing useful work while disconnected is having the necessary information locally available [1]. RDO replica caching is the chief technique provided by Rover to achieve high availability, concurrency, and reliability. In this section, we discuss strategies for selecting objects to cache and for reducing the costs related to maintaining consistency.

#### 3.5.1 Caching

During periods of network connectivity, Rover fills the mobile host's cache with useful RDOs. Rover leaves it up to applications to decide which objects to prefetch. We believe that the usability of applications will be critically dependent upon simple user interface metaphors for indicating collec-

tions of objects to be prefetched. Requiring users to list the names of objects that they wish to prefetch is inherently confusing and error prone. Instead, Rover applications can provide prioritized prefetch lists based on high-level user actions. For example, the Rover e-mail browser automatically generates prefetch operations for the user's inbox folder and recently received messages, as well as folders the user visits or selects.

The Rover toolkit provides clients with cache access and manipulation functions, including checking an RDO's consistency information, flushing RDOs from the cache, and verifying that an RDO is up-to-date. The toolkit also provides server portions of applications with functions for automatically handling RDO validation and cache tags for managing the consistency of client caches. The cache tags provide support for verifying, if possible, an RDO before use; "leasing" an RDO for a fixed period of time; relying on server callbacks for notification of a stale RDO; and indicating that an RDO is immutable.

While caching can bring great benefits, application designers must be careful to avoid unnecessary communication, increased latencies, and deadlock. Applications should avoid caching more data than is necessary to provide good availability, since unnecessary prefetching consumes valuable network resources. In addition, applications should strive to keep update messages small.

### 3.5.2 Consistency

Applications generally require consistency control when clients are allowed to perform concurrent updates on shared RDOs. The Rover toolkit provides significant flexibility in the choice of mechanism, ranging from application-level locking to application-specific algorithms for resolving uncoordinated updates to a single RDO. Certain applications will be structured as a collection of independent atomic actions [29], where the importing action uses application-level locks, version vectors, or dependency-set checks to implement fully serializable transactions within Rover method calls. Others may impose no ordering requirement, allowing all operations, no matter when they reach the server.

Since no single scheme is appropriate for all applications, Rover leaves the selection of consistency scheme to the application. However, only a limited number of schemes lend themselves naturally to mobile environments. For example, using pessimistic concurrency control may cause long blocking periods in a mobile environment; optimistic concurrency control schemes, on the other hand, allow immediate updates by any host on any local data. We therefore expect that optimistic schemes will be widely used, and Rover provides substantial, but not exclusive, support for primary-copy, tentative-update optimistic concurrency control. Specifically, the Rover library supports logging, rollback, and replay of operations; log manipulation; and automatically maintained RDO consistency vectors. All Rover applications built to date use primary-copy consistency control.

The server is responsible for maintaining the consistent view of the system. Update conflicts are detected and resolved by the server, and the results of reconciliation are

treated by clients as overriding tentative state stored at the client. Thus, to reconcile its state and to assure that any updates are durable a client needs to communicate only with the server.

Rover provides clients with facilities for automatically logging the operations performed by mutating method invocations. The operations consist of conflict avoidance and detection checks, the mutating operation, and the data for the operation. Logging operations, rather than only new data values, increases application flexibility in avoiding or resolving conflicts by allowing applications to use type-specific concurrency control [30]. For example, a bank account object with debit, credit, and balance methods provides a great deal more semantic information to the application 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, consistently updating a balance value by overwriting the old value would require the use of an exclusive lock on the global balance.

When the QRPC containing the mutating operation arrives at a server, the server invokes the operation on the primary copy of the object. Typically, the operation first checks whether the object has changed since it was imported by a mobile host. The operation may also check other objects to determine if they have changed since they were imported (so that potential read-write conflicts can be detected). The definition of conflicting modifications is strongly application- and data-specific. Therefore, Rover does not try to detect conflicts directly, although it maintains version vectors for each RDO to aid conflict detection.

Rover uses type-specific concurrency control [30] to enable applications to avoid conflicts. Using Rover, conflict avoidance may depend not only on the application, but on the data or even the operation involved. When a write-write conflict is detected, the application must determine whether the preconditions for the mutating operation are actually violated (yielding a genuine conflict). Since the submitted operation was originally performed at the client on potentially stale data, the result of performing the operation at the server may not be exactly what the client expected. However, the server can use application-specific semantics to create an acceptable value.

## 4 IMPLEMENTATION OF THE ROVER TOOLKIT

As shown in Fig. 2, the Rover toolkit consists of four key components: the access manager, the object cache (client-side only), the operation log, and the network scheduler. These pieces make up the minimal "kernel" of Rover. Further functionality may be imported 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 (e.g., if the mobile host is docked) further RDOs may be loaded to handle these cases [11]. We now discuss each of the four components in turn.

Each machine has a local Rover *access manager*, which is responsible for handling all interactions between client-side

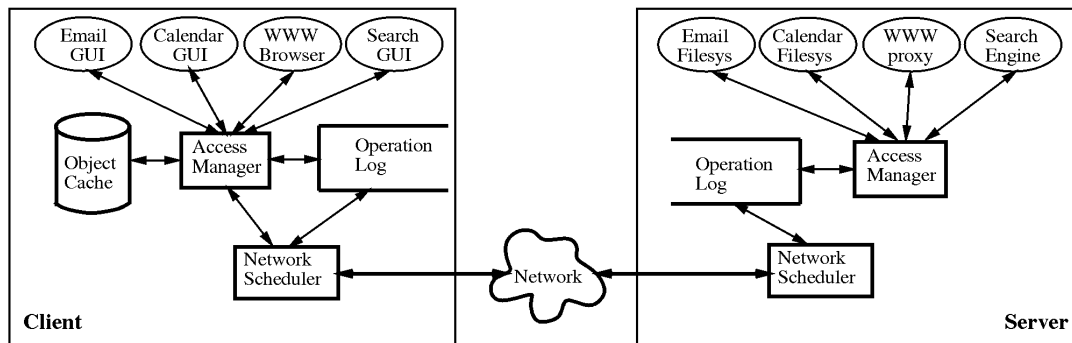


Fig. 2. The Rover architecture consists of four key components: access managers, operation logs, network schedulers, and object caches. The first three of these exist on both the client and the server.

and server-side applications and among client-side applications. The access manager services requests for objects (RDOs), mediates network access, logs modifications to objects, and manages the client object cache. Client-side applications communicate with the access manager to import objects from servers, cache them locally, and dynamically link them into existing computation. Server-side applications are invoked by the access manager to handle requests from client-side applications. Applications invoke the methods provided by the objects and, using the access manager, make changes globally visible by exporting them back to the servers.

In the current implementation, RDOs are implemented using the Tcl and Tk languages [31]. However, since the toolkit interface is designed to be language independent, it will be easy to explore the use of other interpreted or byte-compiled languages (e.g., Java [32]).

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

The access manager also handles failure recovery. This task is eased somewhat by our use of both a persistent cache and an operation log. After a failure, the access manager re-queues any incomplete QRPCs for redelivery. At-most-once delivery semantics are provided by unique identifiers and the persistent log. One issue that remains an open question is how to handle error responses from resent QRPCs for client-side applications that no longer are running. Our implementation currently ignores such responses, although the same reliability techniques we applied to servers [8] could also be applied to client applications.

The *object cache* provides stable storage for local copies of imported objects. The object cache consists of a local private cache located within the application's address space and a global shared cache located within the access manager's address space. Client-side applications do not usually interact directly with the object cache. When a client-side application issues an *import* or *export* operation, the toolkit satisfies the request based on the object's location and the cache consistency option specified for the object (see Section 3.5.1).

The *operation log* provides stable storage for QRPCs. Once an object has been imported into the client-side application's local address space, method invocations without side effects are serviced locally by the object. At the application's discretion, method invocations with side effects may also be processed locally, inserting tentative data into the object cache. Operations with side effects also insert a QRPC into the operation log located at the client. Each insertion is synchronous. Optionally, insertions can be logged asynchronously. This option allows the cost of logging to be amortized across multiple QRPCs, but opens a small (several second) window of vulnerability, where a client failure would cause loss of data.

The stable log is implemented as an ordinary file. Rover performs both a flush and a synchronize operation to force new QRPCs to the log. Thus, the log update is on the critical path for message sending. 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.

One issue Rover addresses with an application-specific approach is operation log growth during disconnected operation. Operation logging may lead to an operation log that grows in size at a rate exceeding that of a simple value log. The traditional approach is log compaction [1]. In addition to providing limited automatic log compaction, Rover also directly involves applications in log compaction by enabling them to download procedures into the access manager to manipulate their log records. For example, an application can filter multiple object verification QRPCs to leave only a single QRPC. In addition, applications can apply their own notion of "overwriting" to the operations in the log.

The original *network scheduler* examines each unsent operation to determine the destination and selects the appropriate transport protocol and medium over which to send it. Rover is capable of using a variety of network transports. Rover supports both connection-based protocols (e.g., HTTP over TCP/IP networks) and connectionless protocols (e.g., SMTP over IP or non-IP networks). Different protocols have different strengths. For example, while SMTP has extremely high latency, it is fundamentally a queued background process; it is more appropriate than more interactive protocols for fetching extremely large documents, such



TABLE 1  
DESIGN DECISIONS FOR THE INITIAL APPLICATION SET  
BUILT USING THE ROVER TOOLKIT

Issue	Design Decision
Object Design	Use RDOs that encapsulate sufficient state to effectively service local requests, but are small enough to easily prefetch
Caching	Use RDOs to cache information
Computation Migration	Use RDOs to migrate computation that requires high bandwidth access
Object Prefetching	Use domain knowledge to determine granularity of RDOs to avoid both extra round trips and overly aggressive prefetching
Notification	Use colors and text to notify users of tentative information
Consistency	Use logs of operations to detect conflicts and either avoid or help resolve them

as stored video, which require large amounts of time regardless of the protocol. Another advantage is that the IP networks required for HTTP or TCP are not always available, whereas SMTP often reaches even the most obscure locations.

We have experimented with two network schedulers. The original network scheduler sent a request as soon as it was received from a client application. The new improved scheduler uses the following heuristic to batch requests that are destined for the same server:

- When a request is received from one client application, it uses the access manager to check all the client applications (including the one that sent the original request) to see if any are in the process of sending a request.
- If there are additional requests pending from any client applications, the scheduler delays sending the original request. Upon receipt of the next request, the scheduler repeats the pending request check.
- When there are no pending requests from client applications (or the first request has been delayed for a preset maximum time), the scheduler batches the requests and sends them on the same connection; the results are also received on the same connection.

There are two situations in which we expect batching to occur: when the mobile host is disconnected and when a client issues a series of requests almost simultaneously (e.g., prefetching or importing several objects, exporting changes to multiple objects, or verifying the consistency of multiple objects). This heuristic imposes a small delay on requests: the time for the access manager to check each client application for pending requests and to receive the requests. However, since the check allows the scheduler to automatically batch requests, it is a small penalty to pay relative to that caused by a high roundtrip time. Thus, an application that issues several requests in a series will have the requests automatically batched and sent to the server using a single connection. The first few requests will incur increased latency; however, throughput and average latency will be substantially improved. We are also investigating alternatives that rely on applications to specify the set of requests that should be batched together.

The network scheduler leverages the queuing of QRPCs performed by the log to gain transmission efficiency. The result is a potentially significant reduction in per-operation

transmission overhead and an increase in connection efficiency through amortization of connection setup and tear-down across multiple requests and responses. This amortization is especially important when connection setup is expensive, either in terms of added latency or dollar cost. For example, the latency for a null RPC over a 9.6 Kbps Cellular CSLIP link is 2.23 seconds; batching offers a substantial performance benefit.

The network scheduler also applies compression to the headers associated with requests and, in the absence of application-specified compression, applies compression to application data. This compression offers significant performance advantages, especially when combined with batching. Typical compression ratios for the applications we have studied range from 1.5:1 to 9.7:1. The combination of batching and compression over low-bandwidth networks yields (on average) a two- to four-fold reduction in execution times over uncompressed, single requests.

## 5 MOBILE COMPUTING USING ROVER

In this section, we discuss the steps involved in implementing mobile-aware applications, porting existing applications to a mobile-aware environment, the programming interface provided by the Rover toolkit, and the set of sample applications that we constructed using the toolkit. Many of the same steps are involved in building a proxy to support a mobile-transparent service. Note that Rover provides a consistent framework for developing mobile applications rather than providing any mechanical tools for building applications.

### 5.1 Using Objects Instead of Files

There are several steps involved in porting an existing application to Rover or creating a new Rover-based application. Each step requires the application developer to make one of several implementation decisions. The choices we used in developing the initial set of Rover applications are presented in Table 1.

We illustrate the development process with an e-mail browser that we have constructed, Rover Exmh. The e-mail browser is a port of Brent Welch's Exmh Tcl/Tk-based e-mail browser.

#### 5.1.1 Object Design

The important first step is to split the application into components and carefully identify which components and

functions should be present on each side of the network link. The division will be mostly static, as most of the file system components will remain on the server and most of the GUI components will remain on the client. However, those components that are dependent upon the computing environment (network or computational resources) or are infrequently used may be dynamically generated. For example, the search operation performed by an Exmh client is dynamically customized to the current link attributes: Over a low-latency link, more work is done at the client and less at the server, and vice versa for a high-latency link.

In migrating to the mobile environment, reading files is replaced by importing objects, and writing files is replaced by exporting changes to those objects. The file-system interface still exists in the server-side portion of the application; however, inserted between the server-side file-system interface and the client-side graphical user interface is an object layer.

### 5.1.2 Caching

Once the application has been split into components, the next step is to appropriately encapsulate the application's state within objects that can be replicated and cached at multiple clients. For example, a user's electronic mail consists of messages and folders. In a traditional distributed file system, one method of encapsulation is to store each message in an individual file and use directories to group the messages into folders. Information about the size or modification date of a message is determined by using file system status operations. With Rover, the corresponding encapsulation stores messages as objects and folders as objects containing references to message objects. Each object encapsulates both the message or folder data and the appropriate metadata.

### 5.1.3 Computation Migration

One of the primary purposes of the object layer is to provide a means of reducing the number of network messages that must be sent between the client and server; this reduction is done by migrating computation. Consider the e-mail folder scan operation, which returns a list of messages and information about the messages in a folder. Using a file system-based approach means scanning the directory for the folder, opening each message, and extracting the relevant information. All of this information would flow across the link between the client and server; thus, this function is an appropriate operation for a well-connected host, but would be very expensive and time-consuming over a high-latency link. Using Rover's object-based approach, the server-side portion of the application constructs a folder object containing the metadata for the messages contained in the folder. The client-side portion of the application then imports the folder object in a single roundtrip request, avoiding multiple roundtrip requests. The multiple requests are replaced by local computation—querying the folder object about the messages it contains. Batching the metadata together trades an increase in the latency to view the metadata for the first message for reduced network communication and fast interactive use of the rest of the metadata.

### 5.1.4 Object Prefetching

Prefetching decisions may be made at design time or run time. The former relies on the designer's knowledge of the application structure. For example, the main portion of Rover Exmh's help information is prefetched by a client, but less frequently referenced portions are loaded on demand. These latter run time decisions add contextual information to the designer's knowledge. For example, the e-mail application automatically prefetches the contents of the most recently visited folders and the messages in the user's inbox.

### 5.1.5 Notification

The next step is to add support for interacting with the environment. For example, in the e-mail application, one of the important pieces of message metadata that a folder object contains is the message's size and the size of any attachments. This information is used by the application and conveyed to users, allowing them to make decisions about which mail messages to import. These sizes could be further used in combination with network performance metrics to support application prefetching decisions.

The application developer also must decide which mechanisms to use for notifying users of the cache status of displayed data. In the e-mail application, color is used to distinguish operations that have not been propagated to a server.

### 5.1.6 Consistency

The final important step is the addition of application-specific conflict avoidance and resolution. Whereas conflicts are infrequent in most stationary environments, in mobile environments, they will be more common. Fortunately, application developers can leverage the additional semantic information that is available with Rover's operation-based (instead of value-based) approach to object updating.

Conflict avoidance has two aspects: flexible precondition tests for operations and operations that are, where possible, commutative. Commutative operations have the advantage that the server may arbitrarily interleave operations presented by several clients without altering the client operations. The precondition test determines whether the operation can still be executed. Consider, for example, a banking application. Given an operation that subtracts \$10 from an account, the precondition should test not for an expected balance, but that the balance be equal to or greater than \$10. Likewise, the operation should not simply write an expected new balance, rather it should read the current balance, subtract \$10, and write the new balance.

Even with application-specific information, it may not be possible to automatically resolve a conflict. In such cases, the application will have to involve the user. The advantage with Rover is that the application can present the user with the current value for an object and the particular operation that was being performed, instead of simply providing the user with the current value and the value the application tried to write.

## 5.2 Toolkit Programming Interface

The programming interface between Rover and its client applications contains four primary functions: *create session*,

*import*, *invoke*, and *export*. Client applications call *create session* once with authentication information to set up a connection with the local access manager and receive a session identifier. The authentication information is used by the access manager to authenticate client requests sent to Rover servers.

Rover objects are referenced using *Uniform Resource Locators*. Each URL is composed of a server name and a unique (to that server) identifier. To import an object, an application calls *import* and provides the object's URL, the session identifier, a callback, and arguments. In addition, the application specifies a priority that is used by the network scheduler to reorder QRPCs. The *import* function immediately returns a promise [33] to the application. The application can then wait on this promise or continue execution. Rover transparently queues QRPCs for each *import* operation in the stable log. When the requested object is received by the access manager, the access manager updates the promise with the returned information. In addition, if a callback was specified, the access manager invokes it. Once an object is imported, an application can *invoke* methods on it to read and change it. Applications export each local change to an object back to servers by calling the *export* operation and providing the object's unique identifier, the session identifier, a callback, and arguments. Like *import*, *export* immediately returns a promise. When the access manager receives responses to exports, it updates the affected promises and invokes any application-specified callbacks.

### 5.3 Rover Application Suite

Section 3 discusses several important issues in designing mobile-aware applications; this section provides examples of how those issues are addressed in several mobile-transparent and mobile-aware applications that have been developed using the Rover toolkit (Table 1 lists the major implementation issues). The two mobile-transparent applications are *Rover NNTP proxy*, a USENET reader proxy; and *Rover HTTP proxy*, a proxy for Web browsers. The mobile-aware applications are *Rover Exmh*, an e-mail browser; *Rover Webcal*, a distributed calendar tool; *Rover Irolo*, a graphical Rolodex tool; and *Rover Stock Market Watcher*, a tool that obtains stock quotes.

Two of the mobile-aware applications are based upon existing UNIX applications: Rover Exmh is a port of Brent Welch's Exmh Tcl/Tk-based e-mail browser, and Rover Webcal is a port of Ical, a Tcl/Tk and C++ based distributed calendar and scheduling program written by Sanjay Ghemawat. Rover Irolo and the Rover Stock Market Watcher were built from scratch. The former was written first using a standard file system interface and then ported to Rover.

This application suite was chosen to test several hypotheses about the ability to reasonably meet users' expectations in a mobile, intermittently-connected environment. Our application suite represents a set of applications that mobile users are in fact currently using. However, current applications are not well integrated with the mobile environment. Because RDOs affect the structure of applications, it is important to qualitatively test the ideas contained in the Rover toolkit with complete applications in addition to using standard quantitative techniques.

As can be seen in Table 2, porting these file system-based workstation applications to a mobile-aware Rover applications required varying amounts of work. Some applications were written or ported in a few weeks. Only Webcal required more than a month of work. In part, the extra time was due to additional evolution and redesign of the conflict resolution semantics of the toolkit. Porting *Exmh* and *Ical* to Rover required simple changes to approximately 10% of the lines of code. Most of these changes consisted of replacing file system calls with object invocations; these modifications in *Rover Exmh* and *Rover Webcal* were made largely independently of the rest of the code.

TABLE 2  
LINES OF CODE CHANGED OR ADDED IN PORTING EXMH  
AND WEBCAL TO ROVER AND USING ROVER TO IMPLEMENT  
THE HTTP AND NNTP PROXIES, IROLO,  
AND THE STOCK WATCHER APPLICATIONS

Rover Application	Base code	New Rover client code	New Rover server code
Exmh	24,000 Tcl/Tk	1,700 Tcl/Tk 220 C	140 Tcl/Tk 2,700 C
Webcal	26,000 C++ and Tcl/Tk	2,600 C++ and Tcl/Tk	1,300 C++ and Tcl/Tk
HTTP Proxy	none	250 Tcl/Tk 1,500 C	740 C
Irolo	470 Tcl/Tk	400 Tcl/Tk 220 C	280 Tcl/Tk
NNTP Proxy	none	530 Tcl/Tk 480 C	350 C
Stock Watcher	none	84 Tcl/Tk 220 C	260 Perl 65 Tcl/Tk

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

#### 5.3.1 Mobile-Transparent Applications

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

**Rover HTTP proxy.** This application is unique in that it interoperates with most of the popular Web browsers. It allows users of existing Web browsers to "click ahead" of the arrived data by requesting multiple new documents before earlier requests have been satisfied. The proxy intercepts all HTTP requests and, if the requested item is not

locally cached, returns a null response to the browser and enqueues the request in the operation log. When a connection becomes available, the page is automatically requested. In the meantime, the user can continue to browse already available pages and issue additional requests for pages without waiting. The granularity of RDOs is individual pages and images.

The client and server cooperate in prefetching. The client specifies the depth of prefetching for pages, while the server automatically prefetches in-lined images.

The proxy uses a separate nonbrowser window to display the status of a page (loaded or pending). If an uncached file is requested and the network is unavailable, an entry is added to the window. As pages arrive, the window is updated to reflect the changes. This window exposes the object cache and operations log directly to the user and allows the user limited control over them.

### 5.3.2 Mobile-Aware Applications

The mobile-aware applications all use the same technique for handling conflict detection and avoidance: a log of changes to RDOs. This log allows the server to detect and either avoid or resolve a conflict. The user is notified of any unresolvable conflicts and may then retry the original operation with or without modifying it or abort it. Also, most of the applications use color coding to indicate that an operation is tentative.

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

*Rover Exmh* gives the user some control over the migration of computation to servers. For example, instead of performing a search of mail folders locally at the client (and thus having to import the index across a potentially low-bandwidth link), the user can instruct the client to construct a query request RDO and send it to the server.

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

**Rover Irolo.** This graphical Rolodex application uses two types of data RDOs, *entries* and *indices* (lists of entries). The GUI displays the last time an entry was updated and indicates whether the item is committed or tentative. For comparison purposes, we also implemented a file system-based version of this application.

**Rover Stock Market Watcher.** This is a simple financial stock tracking application. The client portion of this application constructs and sends an RDO to the server. The RDO executes at the server and watches the attributes of a user-specified stock for changes that exceed a user-specified threshold (e.g., price, volume, or changes). When the threshold is exceeded, the server notifies the client. The server portion of the application uses fault-tolerant techniques to store the real-time information retrieved from stock ticker services [8].

## 6 EXPERIMENTS

*Rover* runs on several platforms: IBM ThinkPad 701C (25/75Mhz i80486DX4) laptops running Linux 1.2.8; Intel Advanced/EV (120 Mhz Pentium) workstations running Linux 1.3.74; DECstation 5000 workstations running Ultrix 4.3; and SPARCstation 5 and 10 workstations running SunOS 4.1.3\_U1. The primary mode of operation is to use the laptops as clients of the workstations. However, workstations can also be used as clients of other workstations.

The *Rover* server executes either as a Common Gateway Interface (CGI) plugin to NCSA's *httpd* 1.5a server (running on Ultrix and SunOS in the non-forking, pool of servers mode), or as a standalone TCP/IP server. The standalone server yields significant performance advantages over the CGI version, as it avoids the fork and exec overheads incurred on each invocation of the CGI version. In addition, because a new copy of the CGI server is started to satisfy each incoming request, any persistent state across connections must be stored in the file system and re-read for each connection, a penalty that the standalone server avoids.

Network options that we have experimented with include 10 Mbps switched Ethernet, 2 Mbps wireless AT&T WaveLAN, 128 Kbps and 64 Kbps Integrated Digital Services Network (ISDN) links, and Serial Line IP with Van Jacobson TCP/IP header compression (CSLIP) over 19.2 Kbps *V.32terbo* wired and 9.6 Kbps *Enhanced Throughput Cellular* (ETC) cellular dial-up links. The configuration used for the cellular experiments was the one suggested by our cellular provider and the cellular modem manufacturer: 9.6 Kbps ETC. The client connected to our laboratory's terminal server modem pool through the cellular service provider's pool of ETC cellular modems. This imposes a substantial added latency of approximately 600 ms, but also yields significantly better resilience to errors. Other choices are 14.4 Kbps ETC and directly connecting to the terminal server modem pool using 14.4 Kbps *V.32bis*. However, both choices suffer from significantly higher error rates, especially when the mobile host is in motion. Also, *V.32bis* is significantly less tolerant of the communication interruptions introduced by the in-band signaling used by cellular phones for cell switching and power level change requests.

The test environment consisted of a single server and multiple clients. The server machine was an Intel Advanced/EV workstation running the standalone TCP/IP server. The clients were IBM ThinkPad 701C laptops. All of the machines were otherwise idle during the tests.

TABLE 3  
THE ROVER EXPERIMENTAL ENVIROMENT

Transport	TCP		QRPC Latency		
	Throughput 1 Mbyte	Latency null RPC	No Logging	Flash RAM Logging	Disk Logging
Ethernet	4.45	8	22	77	97
WaveLAN	1.09	20	34	79	116
128 ISDN	0.57	74	116	168	189
64 ISDN	0.32	87	117	184	191
19.2 Wired CSLIP	0.027	430	738	769	789
9.6 Cellular CSLIP	0.008	2230	3540	3670	3800

*Latencies are in milliseconds, throughput is in Mbps.*

To minimize the effects of unrelated network traffic on the 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. However, network traffic over the wired, cellular, and ISDN links used shared public resources and traversed shared links; thus, there is increased variability in the experimental results for those network transports. To reduce the effects of the variations on the experiments, each experiment was executed multiple times and the results averaged. The standard deviations for our measurements were within 10% of the mean values. It is important to note that ordinary TCP/IP was used on the wireless networks. While Rover applications might benefit from the use of a specialized TCP/IP implementation, it is not necessary. Since a Rover application sends less data than an unmodified application, it is less sensitive to errors on wireless links.

The following experiments are designed to explore the performance characteristics of the Rover toolkit. In particular, the experiments test the following hypotheses:

- 1) The overhead imposed by logging QRPCs in stable logs at both the client and server is negligible for the target environments (i.e., dial-up links).
- 2) The batching and compression of multiple requests that is enabled by using QRPC instead of RPC significantly improves the average request latency and throughput.
- 3) The Rover toolkit substantially improves the performance and usability of mobile-transparent applications.
- 4) Using mobile-aware versions of applications offers significant performance and usability advantages over unmodified versions.

### 6.1 QRPC Performance

To establish the baseline performance for QRPC, we repeated the latency and bandwidth measurement experiments from [6], but extended them to include several additional network technologies and the use of Flash RAM for the client and server stable logs. The results are summarized in Table 3. Null RPC latency is a ping-pong over TCP sockets; TCP throughput is the time to send 1 Mbyte of compressible (14.4:1 using GNU's *gzip -6*) ASCII data similar to Rover Tcl-based RDOs; and QRPC latency is the time to perform a null 200-byte QRPC (200 bytes is the average size of a QRPC from our experience with Rover applications). The ISDN, Wired CSLIP, and Cellular CSLIP links perform hardware compression. The cellular times reflect

the overhead of the ETC protocol and a non-error-free wireless link.

The cost of a QRPC has several primary components: the transport cost (the base null TCP cost from Table 3 plus the per-byte network transmission cost); the stable client and server logging costs; and the execution cost of the QRPC itself. By using stable logging at clients, Rover can guarantee the delivery of requests from clients to servers. The use of server-side stable logging allows Rover to avoid having to retransmit a request from a client (which might be disconnected) after a server failure [8]. The results show that the relative impact of logging is a function of the transport media. Since we expect that Rover users will often be connected via slower links (e.g., wired or cellular dial-up), the cost of stable logging will be a minor component of overall performance (e.g., less than 1% for cellular links when using Flash RAM). Thus, we believe it is acceptable to pay the additional cost for client and server logging of QRPCs.

To understand the effects of batching and compression, we used a synthetic workload and measured the performance of QRPC with asynchronous logging. Using asynchronous logging allows the cost of logging to be amortized across multiple QRPCs. The synthetic work load consisted of a issuing a series of 60 200-byte null QRPCs. For the batching and multiple outstanding requests experiments, there were no interarrival pauses.

The network scheduler's batching algorithm yields the following behavior:

- 1) When the network scheduler receives each of the first few requests, the queue of pending requests from the test client is empty; so each request is sent in a separate message. No delay is imposed on these messages.
- 2) As the access manager starts spending additional time processing outgoing messages and incoming responses, the queue of pending requests from the test client grows in size.
- 3) At this point, the network scheduler constructs large batches of requests. Early messages in each batch incur some delay, while later messages in the same batch incur significantly less delay.

Fig. 3 shows the effects of batching and compression (using the heuristic from Section 4) on the per-request cost using the synthetic workload. In each set of bars, the leftmost bar (compression and batching) shows the performance when both compression and batching are applied. For this test, the compression ratio was approximately 12 to one and the batch size was an average of seven requests per

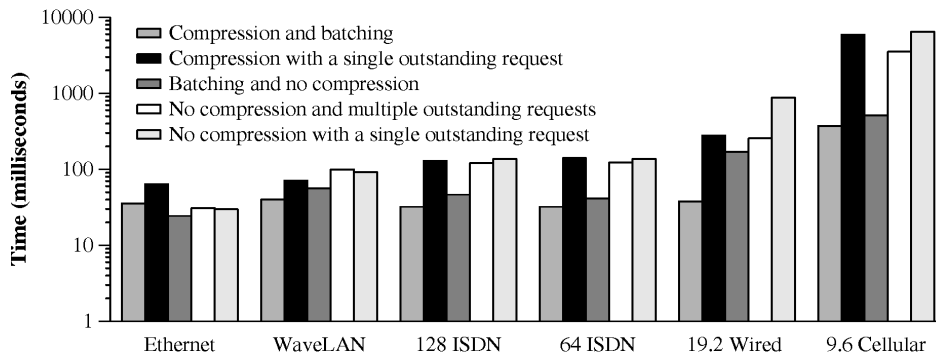


Fig. 3. Average time in milliseconds for one null QRPC when using compression and batching of a synthetic workload consisting of null QRPCs with asynchronous log record flushing. The y-axis uses a logarithmic scale.

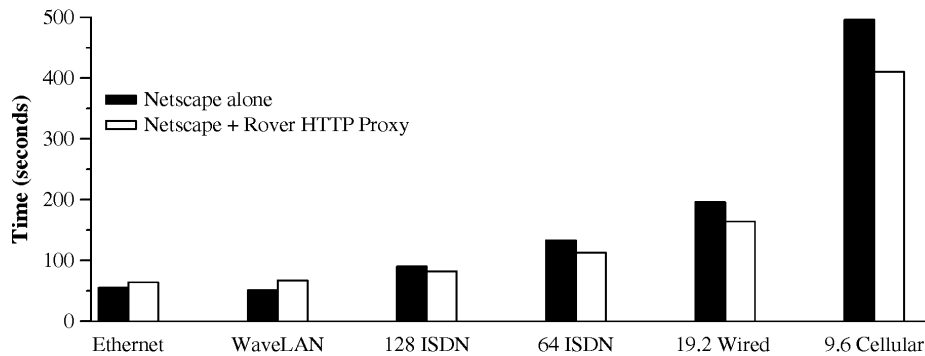


Fig. 4. Time in seconds to fetch and display 10 WWW pages using Netscape alone and with the Rover HTTP proxy.

message with individual batches ranging in size from one to 55 requests, depending upon the speed of the network link.

The second bar (compression with a single outstanding request) shows the performance with compression and only a single request outstanding. The compression ratio was 2.5 to one. The third bar (batching and no compression) shows the performance with batching and no compression. Batches contained an average of seven requests per message with individual batches ranging in size from one to 55 requests, depending upon the speed of the network link.

The fourth bar (no compression and multiple outstanding requests) shows the performance without compression or batching, but with multiple outstanding requests. The rightmost bar (No compression with a single outstanding request) shows the performance without compression or batching and with only a single request outstanding.

The results show that together compression and batching offer performance gains for all networks with the largest gains occurring for the slowest networks. The main reason for the batching performance gain is the elimination of multiple roundtrip latencies. Compression offers a significant benefit only when used with batching because it is able to compress multiple QRPC headers within a batch.

## 6.2 Mobile-Transparent Application Performance

We compared the performance of Netscape using a mobile-transparent Rover HTTP proxy against the same application executing independently. We measured the time to fetch and display 10 WWW pages using a variety of networks. Fig. 4 provides the results of the experiment and

shows that performance when using the Rover HTTP proxy is comparable for faster networks and up to 17% faster for the slower networks. The performance gain is a result of the elimination of multiple roundtrip latencies to open connections for inlined objects and the compression of the data. The total data transmitted to the client was 239 Kbytes of compressed data equivalent to 286 Kbytes of uncompressed data. The HTML portion of the pages accounted for 44.5 Kbytes and had a compression ratio of 2.6:1. The majority of the data consisted of images, which were far less compressible using the default compression. We plan to explore the use of application-specific image compression [23]. It is important to note that the experiments do not reflect the "click-ahead" nature of the Netscape+Rover HTTP proxy application, which allows the user to browse the loaded pages while waiting for additional pages to load.

## 6.3 Mobile-Aware Application Performance

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

To measure the performance benefits of the complete Rover system for mobile-aware applications, we compare the performance of Rover Webcal, Rover Exmh, and Rover Irolo to their unmodified X11-based counterparts, Ical, Exmh, and Irolo. For each application, we designed a workload representative of a typical user's actions and measured the time to perform the complete task. To keep the measurements representative, we did not measure the cost of

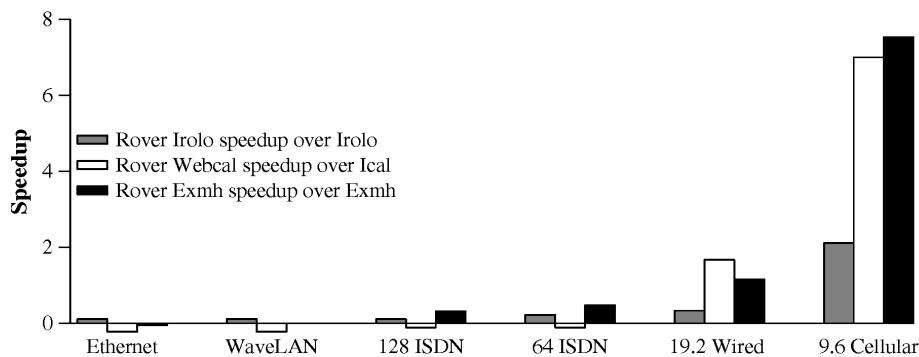


Fig. 5. Speedup (or slowdown) of Rover mobile-aware versions of applications over the original X11-based applications while performing a sample set of tasks.

starting the application and loading the data required for the task. This operation is typical of how most people use mobile computers, where the application is started and frequently used data is loaded over a fast network (e.g., at the office) and then the application is used repeatedly over a slow network or without any network connectivity (e.g., on a plane). Each task was repeated on each of the six network options.

Fig. 5 presents the speedup of the Rover version of each application over the original X11-based application while performing a sample set of tasks. The tasks were: reading eight MIME e-mail messages, viewing one week's appointments from a calendar with 50 items in it (the approximate median size of shared and private user calendars on our systems), and browsing fifty Rolodex entries. In general, the results show that, for fast networks (Ethernet, WaveLAN, and ISDN), the performance when using Rover is comparable (a slight speedup for Irolo, equal for Exmh, and a slight slowdown for Ical). Over slower networks (wired and cellular dial-up links), Rover application performance is consistently better (ranging from a 33% performance gain on wired dial-up to a factor of 7.5 on cellular dial-up). The results for these two networking technologies are especially encouraging, since they represent the target environment for Rover.

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

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

## 7 CONCLUSIONS

We have shown that the integration of relocatable dynamic objects and queued remote procedure calls in the Rover toolkit provides a powerful basis for building mobile-transparent and mobile-aware applications. We have found it quite easy to adapt applications to use these Rover facilities, resulting in applications that are far less dependent on

high-performance communication connectivity. For example, one might conjecture that it would be difficult to build a mobile version of Netscape that provides useful service in the absence of network connectivity. In practice, we find the combination of the Rover cache, relocatable dynamic objects, and queued remote procedure calls results in a surprisingly useful system. Furthermore, it was much easier to build a proxy using the toolkit than with an ad-hoc approach.

RDOs and QRPCs allow application developers to decouple many user-observable delays from network latencies. The result is excellent graphical user interface performance over network technologies that vary by three orders of magnitude in bandwidth and latency. In addition, measurements of end-to-end mobile application performance shows that mobile-transparent and mobile-aware applications perform significantly better than their stationary counterparts. For example, for the mobile-transparent Netscape application, we observe a performance improvement of 17%. For mobile-aware applications, we observe performance improvements of up to a factor of 7.5 over slow networks.

## ACKNOWLEDGMENTS

We thank David Gifford for his efforts in the early design stages of the Rover toolkit, in particular his idea of QRPC. We also thank Atul Adya, Greg Ganger, Eddie Kohler, Massimiliano Poletto, and the anonymous reviewers for their careful readings of earlier versions of this paper. We would also like to thank the rest of the Rover project team: George Candea, Constantine Cristakos, Alan F. deLepinasse, and Michael Shurpik for helping with the development of Rover. This work was supported in part by the Advanced Research Projects Agency under contract DABT63-95-C-005, by a U.S. National Science Foundation Young Investigator Award, by an Intel Graduate Fellowship, and by grants from AT&T, IBM, and Intel.

## REFERENCES

- [1] J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Trans. Computer Systems*, vol. 10, pp. 3-25, Feb. 1992.
- [2] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Systems*, vol. 6, no. 2, pp. 213-226, June 1981.

- [3] L. Huston and P. Honeyman, "Partially Connected Operation," *Proc. Second USENIX Symp. Mobile and Location-Independent Computing*, pp. 91-97, Ann Arbor, Mich., Apr. 1995.
- [4] P. Kumar, "Mitigating the Effects of Optimistic Replication in a Distributed File System," PhD thesis, School of Computer Science, Carnegie Mellon Univ., Dec. 1994.
- [5] J.H. Saltzer, D.P. Reed, and D.D. Clark, "End-to-End Arguments in System Design," *ACM Trans. Computer Systems*, vol. 2, no. 4, pp. 277-288, Nov. 1984.
- [6] A. Joseph, A.F. deLepinasse, J.A. Tauber, D.K. Gifford, and F. Kaashoek, "Rover: A Toolkit for Mobile Information Access," *Proc. 15th Symp. Operating Systems Principles (SOSP)*, pp. 156-171, Copper Mountain Resort, Colo., Dec. 1995.
- [7] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, vol. 2, no. 1, pp. 39-59, Feb. 1984.
- [8] A. Joseph and F. Kaashoek, "Building Reliable Mobile-Aware Applications Using the Rover Toolkit," *Proc. Second Int'l Conf. Mobile Computing and Networking (MOBICOM '96)*, pp. 117-129, Rye, N.Y., Nov. 1996.
- [9] R.H. Katz, "Adaptation and Mobility in Wireless Information Systems," *IEEE Personal Comm.*, vol. 1, no. 1, pp. 6-17, First Quarter 1994.
- [10] N. Davies, G. Blair, K. Cheverst, and A. Friday, "Supporting Adaptive Services in a Heterogeneous Mobile Environment," *Proc. Workshop Mobile Computing Systems and Applications*, pp. 153-157, Santa Cruz, Calif., Dec. 1994.
- [11] F. Kaashoek, T. Pinckney, and J.A. Tauber, "Dynamic Documents: Mobile Wireless Access to the WWW," *Proc. Workshop Mobile Computing Systems and Applications*, pp. 179-184, Santa Cruz, Calif., Dec. 1994.
- [12] 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," *Proc. 15th Symp. Operating Systems Principles (SOSP)*, pp. 172-183, Copper Mountain Resort, Colo., Dec. 1995.
- [13] M.G. Baker, "Changing Communication Environments in MosquitoNet," *Proc. Workshop Mobile Computing Systems and Applications*, pp. 64-68, Santa Cruz, Calif., Dec. 1994.
- [14] L.B. Mummert, M.R. Ebling, and M. Satyanarayanan, "Exploiting Weak Connectivity for Mobile File Access," *Proc. 15th Symp. Operating Systems Principles (SOSP)*, pp. 143-155, Copper Mountain Resort, Colo., Dec. 1995.
- [15] B.D. Noble, M. Price, and M. Satyanarayanan, "A Programming Interface for Application-Aware Adaptation in Mobile Computing," *Proc. Second USENIX Symp. Mobile and Location-Independent Computing*, pp. 57-66, Ann Arbor, Mich., Apr. 1995.
- [16] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G.J. Popek, "Resolving File Conflicts in the Ficus File System," *Proc. USENIX Summer 1994 Technical Conf.*, pp. 183-195, Boston, 1994.
- [17] T. Watson and B. Bershad, "Local Area Mobile Computing on Stock Hardware and Mostly Stock Software," *Proc. First USENIX Symp. Mobile and Location-Independent Computing*, pp. 109-116, Cambridge, Mass., Aug. 1993.
- [18] T. Watson, "Application Design for Wireless Computing," *Proc. Workshop Mobile Computing Systems and Applications*, pp. 91-94, Santa Cruz, Calif., Dec. 1994.
- [19] 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," *Proc. 1994 Symp. Parallel and Distributed Information Systems*, pp. 140-149, Sept. 1994.
- [20] B.R. Badrinath and G. Welling, "Event Delivery Abstractions for Mobile Computing," Technical Report LCSR-TR-242, Dept. of Computer Science, Rutgers Univ., 1996.
- [21] M.T. Le, F. Burghardt, S. Seshan, and J. Rabacy, "InfoNet: The Networking Infrastructure of InfoPad," *Proc. Spring COMPCON Conf.*, pp. 163-168, San Francisco, Mar. 1995.
- [22] S. Narayanaswamy, S. Seshan, E. Amir, E. Brodersen, F. Bughardt, A. Burstein, T. Chang, A. Fox, J. Gilbert, R. Han, R. Katz, A. Long, D. Messerschmitt, and J. Rabacy, "Application and Network Support for InfoPad," *IEEE Personal Comm.*, vol. 3, no. 2, pp. 4-17, Apr. 1996.
- [23] A. Fox, S.D. Gribble, E. Brewer, and E. Amir, "Adapting to Network and Client Variability via On-Demand Dynamic Distillation," *Proc. Seventh Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 160-173, Cambridge, Mass., Oct. 1996.
- [24] J. Bartlett, "W4—The Wireless World-Wide Web," *Proc. Workshop Mobile Computing Systems and Applications*, pp. 176-178, Santa Cruz, Calif., Dec. 1994.
- [25] M.H. Brown and R.A. Schillner, "DeckScape: An Experimental Web Browser," Technical Report 135a, Digital Equipment Corp. Systems Research Center, Mar. 1995.
- [26] R. Katz, E.A. Brewer, E. Amir, H. Balakrishnan, A. Fox, S. Gribble, T. Hodes, D. Jiang, G. Thanh Nguyen, V. Padmanabhan, and M. Stemm, "The Bay Area Research Wireless Access Network (BARWAN)," *Proc. Spring COMPCON Conf.*, Santa Clara, Calif., Feb. 1996.
- [27] L. Kowell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif, "Replicated Document Management in a Group Communication System," *Proc. Second Conf. Computer-Supported Cooperative Work*, Portland, Ore., Sept. 1988.
- [28] J. Gray, P. Helland, P. O'Neill, and D. Shasha, "The Dangers of Replication and a Solution," *Proc. 1996 SIGMOD Conf.*, pp. 173-182, Montreal, Quebec, Canada, June 1996.
- [29] D.K. Gifford and J.E. Donahue, "Coordinating Independent Atomic Actions," *Proc. Spring COMPCON Conf.*, p. 92, San Francisco, Feb. 1985.
- [30] W. Weihl and B. Liskov, "Implementation of Resilient, Atomic Data Types," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 2, pp. 244-269, Apr. 1985.
- [31] J.K. Ousterhout, *Tcl and the Tk Toolkit*. Reading, Mass.: Addison-Wesley, 1994.
- [32] K. Arnold and J. Gosling, *The Java Programming Language*. Reading, Mass.: Addison-Wesley, 1996.
- [33] B. Liskov and L. Shira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls," *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, pp. 260-267, Atlanta, June 1988.



**Anthony D. Joseph** received an SB in computer science and engineering and an SM in electrical engineering and computer science from the Massachusetts Institute of Technology in 1988. He is currently a PhD candidate at MIT and a member of the Laboratory for Computer Science. His research interests include the design, implementation, and use of parallel and distributed systems, with a special focus on mobile computing. He has worked on a variety of projects, including the Proteus parallel simulator, the Prelude language, and distributed garbage collection.



**Joshua A. Tauber** received a BS in computer science and a BA in government from Cornell University in 1991. He received an SM in electrical engineering and computer science from the Massachusetts Institute of Technology in 1996. He is currently a PhD candidate at MIT and a member of the Laboratory for Computer Science. His principal research interest is in distributed computing systems. His work focuses on mobile code and system software for mobile computers. Tauber investigated mobile computing and innovative user interfaces at the Matsushita Information Technology Laboratory in Princeton, New Jersey.



**M. Frans Kaashoek** received a PhD from the Vrije Universiteit, Amsterdam, The Netherlands, in 1992, for his work on group communication in distributed computer systems. Currently, he is a Jamieson Career Development associate professor in the Electrical and Electronics Engineering Department at the Massachusetts Institute of Technology and a member of the Laboratory for Computer Science. At Vrije Universiteit, he was one of the principal designers of the Amoeba distributed operating system and the Orca distributed shared memory system. At MIT, he is working on the exokernel operating system architecture, the Rover mobile toolkit, the C language, and the Fugu scalable workstation. His principal field of interest is computer systems: operating systems, networking, programming languages, compilers, and computer architecture for distributed systems, mobile systems, and parallel systems.