Rover: A Toolkit for Mobile Information Access

Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek
M.I.T. Laboratory for Computer Science
Cambridge, MA 02139, U.S.A.
{adj, aldel, josh, gifford, kaashoek}@lcs.mit.edu

Abstract

The Rover toolkit combines *relocatable dynamic objects* and *queued remote procedure calls* to provide unique services for "roving" mobile applications. A relocatable dynamic object is an object 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 non-blocking remote procedure call requests even when a host is disconnected, with requests and responses being exchanged upon network reconnection. The challenges of mobile environments include intermittent connectivity, limited bandwidth, and channel-use optimization. Experimental results from a Rover-based mail reader, calendar program, and two non-blocking versions of World-Wide Web browsers show that Rover's services are a good match to these challenges. The Rover toolkit also offers advantages for workstation applications by providing a uniform distributed object architecture for code shipping, object caching, and asynchronous object invocation.

1 Introduction

Application designers for mobile "roving" computers face a unique set of communication and power constraints that are absent in traditional workstation settings. For example, although mobile communication infrastructures are becoming more common, network bandwidth in mobile environments will often be limited, and at times, unavailable. In addition, mobile hosts tend to have limited, but variable hardware resources. A mobile host may be completely disconnected, plugged into AC power, plugged into a wired network, or docked. Docking stations can provide a wide range of additional resources including co-processors, stable storage, and user interface devices. As a mobile host gains access to new hardware resources, client-server computation allocation trade-offs may change. Therefore, mobile application designers have a common need for system facilities that minimize dependence upon continuous connectivity, that provide tools to optimize the utilization of network bandwidth, and that allow for dynamic division of work between client and server.

The Rover toolkit provides mobile application developers with a set of tools to isolate mobile applications from the limitations of mobile communication systems. The Rover toolkit provides mobile communication support based on two ideas: *relocatable dynamic objects* (RDOs) and *queued remote procedure call* (QRPC). A relocatable dynamic object is an object 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 non-blocking remote procedure calls [6] even when a host is disconnected: requests and responses are exchanged upon network reconnection.

The Rover toolkit offers applications a uniform distributed object system based on a client/server architecture. Rover applications employ a check-in, check-out model of data sharing: they **import** RDOs into their address spaces, **invoke** methods provided by the RDOs, and **export** the RDOs back to servers. An RDO might be as simple as a calendar item with its associated operations or as complex as a module that encapsulates part of an application (*e.g.,* the user interface of a calendar tool). More complex RDOs may create threads of control when they are imported. The safe execution of RDOs is ensured by executing them in a controlled environment.

Rover permits disconnected hosts to update shared objects. Object consistency is provided by application-level locking or by using application-specific algorithms to resolve uncoordinated updates to a single object. In Rover, every object has a "home" server. A mobile host imports objects into its local cache and exports updated objects back to their home servers. Update conflicts are detected at the server, where Rover attempts to reconcile them. Because Rover can employ type-specific concurrency control [62], we expect that many conflicts can be resolved automatically. In addition, we expect that certain applications will be structured as a collection of independent atomic actions [16], where the importing action sets an appropriate application-level lock.

Our initial implementation of Rover permits applications full access to the objects of the World-Wide Web (WWW) [4]. Objects are named using Universal Resource Names (URNs) [52] and our implementation is fully compatible with the HyperText Transport Protocol (HTTP) [5]. Our research prototype is modular, and favors ease of implementation and experimentation over performance. The Rover toolkit supports several transport protocols (*e.g.,* HTTP and Simple Mail Transport Protocol (SMTP) [45]) over various communication media (*e.g.,* Ethernet, WaveLAN, and phone lines). SMTP allows Rover to exploit E-mail for queued communication. We have developed three Rover-based applications: an E-mail reader, a calendar program, and a WWW browser proxy that provides two non-blocking versions of common WWW browsers. These applications are representative of the set that mobile users are likely to

This work was supported in part by the Advanced Research Projects Agency under contract DABT63-95-C-005, by an NSF National Young Investigator Award, and by grants from IBM and AT&T.

Appears in *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

use. In these applications, RDOs are downloaded into the client and executed to reduce communication requirements, and server requests are queued for asynchronous processing.

Experience with applications developed using Rover shows that it is possible to build interactive applications that isolate a user from the loss of network connectivity and from limited network bandwidth. For example, in the case of NCSA's *Mosaic* WWW browser [41], Rover delivers information immediately if it is available in the local Rover cache; in the case of a cache miss, it queues a request and returns immediately. The user is later notified when the information arrives. Thus, a user can "click ahead" of the communication channel, and expect a stream of responses as they become available. In addition, forms and other interactive components that are typically implemented with server scripts can be downloaded for local execution. Submitting a form is also implemented asynchronously, and a user can proceed even if the mobile host is disconnected. Form responses are delivered as they are received.

Certain aspects of Rover are also attractive for workstation applications. For example, Rover provides distributed workstation applications with a uniform object architecture. These applications can use RDOs to offload servers by caching objects and downloading functionality from servers to clients, or vice versa. RDOs can also be used to dynamically extend clients; for example, a WWW browser can be dynamically extended to include a particular payment scheme. QRPCs can be used to send RPCs to remote hosts (with SMTP as a transport protocol), which can permit servers to batch process requests, and can shift low priority inquiries to off hours. Responses to these RPCs are transported by SMTP to a nearby mail host, which can stage the responses for rapid retrieval by the client. Rover provides all these features in a well-defined object framework, which simplifies the construction of high-performance distributed applications.

We draw four main conclusions from our experimental data:

1. Applications written using a file system model for sharing and storing data can be relatively easily ported to Rover's object model. Porting Brent Welch's *Exmh* and Sanjay Ghemawat's *Ical* to Rover required simple changes to 10% and 15% of the lines of code, respectively.

2. QRPC performance is acceptable even if every RPC is stored in a stable log, or if SMTP is used as a transport protocol. For lower-bandwidth networks the overhead of writing the log is dwarfed by the underlying communication costs.

3. Caching RDOs reduces latency and bandwidth consumption. A local invocation on an RDO is 56 times faster than sending an RPC over a TCP/CSLIP14.4 [25] connection.

4. Migrating RDOs provides Rover applications with excellent performance over moderate bandwidth links (*e.g.,* 14.4 Kbit/s dial-up lines) and in disconnected operation. *Exmh*, a Tcl/Tk based E-mail browser, running over NCD's PC-Xware 2.0's *Xremote* on a 14.4 Kbit/s line is 8.3 times slower than *Rover Exmh*, our port of *Exmh*. Using RDOs also yields significant improvements in Graphical User Interface (GUI) responsiveness when using low bandwidth network links.

In the remainder of this paper we present the context of our work (Section 2), the Rover programming model (Section 3), the Rover architecture (Section 4), discuss the Rover implementation and HTTP compatibility (Section 5), describe the current Rover applications (Section 6), review experimental results from the applications (Section 7), and conclude with observations on the benefits and limitations of the Rover approach (Section 8).

2 Related Work

To our knowledge, no one has studied an architecture like Rover's, which provides both queued RPC and relocatable dynamic objects. Queued RPC is unique in that it provides support for asynchronous fetching of information, as well as for lazily queuing updates. The use of relocatable dynamic objects for dealing with the constraints of mobile computing—intermittent communication, varying bandwidth, and resource poor clients—is also unique to the Rover architecture.

The Coda project pioneered the provision of distributed services for mobile clients. In particular, it investigated how to make file systems run well on mobile computers by using optimistic concurrency control and prefetching [30, 50]. Coda logs all updates to the file system during disconnection and replays the log on reconnection. Coda provides automatic conflict resolution mechanisms for directories and files, and uses Unix file naming semantics to invoke application-specific conflict resolution programs at the file system level [31]. A manual repair tool is provided for conflicts of either type that cannot be resolved automatically. A newer version of Coda also supports low bandwidth networks, as well as intermittent communication [39].

The Ficus file system also supports disconnected operation, but relies on version vectors to detect conflicts [48]. The Little Work Project caches files to smooth disconnection from an AFS file system [23]. Conflicts are detected and reported to the user. Little Work is also able to use low bandwidth networks [22].

The Bayou project [11, 54] defines an architecture for sharing data among mobile users. Bayou addresses the issues of tentative data values [55] and session guarantees for weakly-consistent replicated data [53]. To illustrate these concepts, the authors have built a calendar tool and a bibliographic database. Rover borrows the notions of tentative data, session guarantees, and the calendar tool example from the Bayou project. Rover extends this work with RDOs and QRPC to deal with intermittent communication, limited bandwidth, and resource poor clients.

An alternative to the Rover object model is the Thor object model [36]. In Thor, objects are updated within transactions that execute entirely within a client cache. However, Thor does not support disconnected operation: clients have to be connected to the server before they can commit. An extension for disconnected operation in Thor has been proposed by Gruber and others [19], but it has not been implemented. Furthermore, it does not provide a mechanism for non-blocking communication, and their proposed object model does not support method execution at the servers.

The BNU project implements an RPC-driven application framework on mobile computers. It allows for function shipping by downloading Scheme functions for interpretation [61]. Application designers for BNU noted that the workload characterizing mobile platforms is different from workstation environments and will entail distinct approaches to user interfaces [33]. The BNU environment includes proxies on stationary hosts for hiding the mobility of the system. No additional support for disconnected operation, such as Rover's queued RPC, is included in BNU. A follow-up project, Wit, addresses some of these shortcomings and shares many of the goals of Rover, but employs different solutions [60].

RDOs can be viewed as simple Agents [49] or as a light-weight form of process migration [14, 46, 51, 56]. Other forms of code shipping include Display Postscript [1], Safe-Tcl [7], Active Pages [21], Dynamic Documents [26], and LISP Hypermedia [38]. RDOs are probably closest to Telescript [63], Ousterhout's Tcl agents [44], and Java [18]. Most differences between RDOs and these other forms of code shipping are immaterial because the particular form of code shipping is orthogonal to the Rover architecture. The key difference between Rover and other code shipping systems is that Rover provides RDOs with a well-defined object-based execution environment that provides a uniform naming scheme, an application-specific replication model, and QRPC.

The InfoPad project [34] and W4 [3] focus on mobile wireless information access. The Infopad project employs a dumb terminal, and offloads all functionality from the client to the server. W4 employs a similar approach for accessing the Web from a small PDA. Rover, is designed to be more flexible. Depending on the power of the mobile host and the available bandwidth, Rover dynamically adapts and moves functionality between the client and the server.

A number of proposals have been made for dealing with the limited communication environments for mobile computers. Katz surveys many of the challenges [27]. Baker describes MosquitoNet, which shares similar goals with Rover, but has not been implemented yet [2]. Oracle recently released a product for mobile computers that provides asynchronous communication [15]; unfortunately, details and performance analysis are not available.

A number of successful commercial applications have been developed for mobile hosts and limited-bandwidth channels. For example, Qualcomm's Eudora is a mail browser that allows efficient remote access over low-bandwidth links. Lotus Notes [28] is a groupware application that allows users to share data in a weakly-connected environment. Notes supports conflict detection, but reflects all conflicts to the user for resolution. The Rover toolkit and its applications provide functionality that is similar to these proprietary approaches and it does this in an application-independent manner. Using the Rover toolkit, standard workstation applications, such as *Exmh* and *Ical*, can be easily turned into "roving" mobile applications.

The DeckScape WWW browser [8] is a "click-ahead" browser that was developed simultaneously with our web browser proxy. However, their approach was to implement a browser from scratch; as such, their approach is not compatible with existing browsers.

Several systems use E-mail messages as a transport medium, and obtain similar benefits as we obtain by using QRPC. The Active Message Processing project [57] has developed various applications, including a distributed calendar, which use E-mail messages as a transport medium. In another project, researchers at DEC SRC used E-mail messages as the transport layer of a project that coordinated more than a thousand independently administered and geographically dispersed nodes to factor integers of more than 100 digits [35]. The application is a centralized client-server system with one server at DEC SRC that automatically dispatches tasks and collects results.

Our research borrows from early work on replication for non-mobile distributed systems. In particular, we borrow from Locus [59] (type-specific conflict resolving) and Cedar [17] (check-in, check-out model of data sharing).

3 Rover Programming Model

In this section, we discuss the Rover programming model, QRPC, RDOs, user interface issues, and Rover application development.

3.1 Overview

The Rover toolkit offers applications a distributed object system based on a client/server architecture. Clients are Rover applications that typically run on mobile hosts, but could run on stationary hosts as well. Servers, which may be replicated, typically run on stationary hosts and hold the long term state of the system. Rover applications employ a check-in, check-out model of data sharing: they **import** objects into a local object cache, **invoke** methods provided by the objects, and **export** the objects back to servers. This process is reflected in Figure 1.

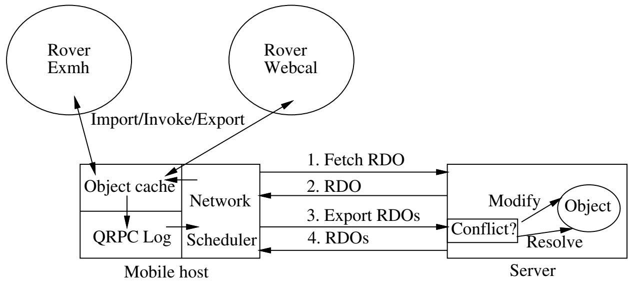

Figure 1: Rover offers applications a client/server distributed object system with client caching and optimistic concurrency control. Rover applications employ a check-in, check-out model of data sharing by calling **import**, **invoke**, and **export** operations. A network scheduler drains the stable QRPC log, which contains the RPCs that must be performed at the server.

Rover objects are named by unique object identifiers, which are URNs in our prototype. Servers store RDOs, which are objects with well-defined interfaces that can be dynamically relocated from the server to the client, or vice versa. An RDO might be as simple as a calendar item with its associated operations or as complex as a module that encapsulates part of an application (*e.g.,* the user interface of a calendar tool). These more complex RDOs may create a thread of control when they are imported. The safe execution of RDOs is ensured by executing them in a controlled environment.

Rover applications can choose the degree of consistency used for replicating objects. Rover caches objects on mobile hosts in a cache that is shared by all applications running on that host. Cached objects are secondary copies of objects; the exporting server retains the primary copy. When Rover invokes a method on an object, it first checks the object cache. If the object is resident, Rover modifies the object without contacting the server. These updates to the cached copy are marked *tentatively committed*. Each update will be marked *committed* when the update is performed upon the server's copy of the object. By using tentatively committed objects, applications can continue computation even if the mobile host is disconnected. Rover applications typically reflect tentatively-committed objects to users (*e.g.,* by displaying them in a different color) so that users can tell that their updates have not yet been committed.

If an object is not present, Rover lazily fetches it from the server using a QRPC. Rover stores the QRPC in a stable log at the mobile host and returns control to the application. The application can register a *callback* routine with Rover, which will be called by Rover to notify the application that the object has arrived. If the mobile host is connected, the Rover network scheduler drains the log in the background and forwards queued QRPCs to the server. The Rover network scheduler may deliver QRPCs in a different order from their enqueued order (*i.e.,* non-FIFO), depending upon associated priorities and the dollar costs for communicating with the server.

Upon arrival of a fetch request, the server fetches the requested object and sends it back to the mobile host. If a mobile host is disconnected between sending the request and receiving the reply, Rover will replay the request from its stable log upon reconnection. Upon receiving a fetch reply, Rover inserts the object returned into the cache and deletes the QRPC from the stable log. In addition, if a callback routine is registered, Rover will perform the callback to inform the application that the object has arrived. The application can then invoke methods on the local copy.

When an invoked method modifies a cached object, Rover lazily updates the primary copy at the server by sending the method call in a QRPC to the server, and returns control to the application. When the QRPC arrives at the server, the server invokes the requested method on the primary copy. Typically a method call first checks whether the object has changed since it was imported by a mobile host. Rover maintains version vectors for each object so that methods can easily detect such changes. If the object has not changed, the method modifies the primary copy and sends a reply back to the mobile host. Upon arrival of the reply, Rover changes the object from tentatively committed to committed, deletes the QRPC from the stable log, and invokes a callback (if one is registered).

If a method call at the server detects an update-update conflict, then the conflict is resolved in an application-specific manner. The Rover toolkit itself does not enforce any specific concurrency control mechanism or consistency guarantees on copies of objects. Instead, it provides a mechanism for detecting conflicts and leaves it up to applications to reconcile them. For example, the *Rover Exmh* E-mail browser exploits semantic knowledge about folders and messages to determine whether a conflict violates consistency. For example, concurrently deleting two different messages in the same folder does not result in a conflict. If there is a conflict that cannot be reconciled, the method returns with an error. These errors are reflected to the user so that he or she can resolve the conflict.

The interval between the time an object is imported to an application and the time it is exported back to the server represents the time during which conflicting updates may occur. Applications can eliminate this window by using application-specific locks or can reduce this window through the use of periodic polling or server callbacks.

The Rover shared object model is different from Coda's shared file model [30]. In the Rover model, consistency is provided by application-level locking or by using application-specific algorithms to resolve uncoordinated updates to a single object. In the Coda model, concurrency is provided by open/close operations on files, extended with file-specific conflict resolvers [32]. The disadvantage of using a file model is that it provides only coarse-grained conflict resolution, since it does not allow for type-specific concurrency control and replication. By storing each datum in a separate file some of the disadvantages of the Coda model can perhaps be avoided; however, the Coda resolvers cannot distinguish between two files of the same type. In addition, because Coda can only use read/write operations, semantic information about operations is lost; without the semantic information, it may not be possible to commute two writes.

Rover provides cryptographic authentication for client-server interactions. Clients authenticate themselves and exported changes to servers; likewise, servers authenticate themselves to clients.

3.2 Queued Remote Procedure Call

Queued remote procedure call (QRPC) is a communication system that permits applications to continue to make RPC requests even when a host is disconnected, with requests and responses being exchanged upon network reconnection. For example, when a client imports an object from a server, Rover appends the **import** operation to a stable log and returns to the application. If the mobile computer is connected, the network scheduler will send the request to the server. When the reply arrives, Rover delivers a notification to the application. If the mobile host is disconnected (or communication is too expensive) the RPC remains in the operation log until the client is reconnected (or communication becomes less expensive); at that point all enqueued QRPCs are sent. This asynchronous communication model allows applications to decouple themselves from the underlying communication infrastructure. During disconnected operation, the network simply appears to be very slow.

Unlike simple message passing, QRPC incorporates stub generation, marshaling and unmarshaling of arguments, and at-most-once delivery semantics. QRPC differs from traditional asynchronous RPC in its failure semantics. A traditional RPC fails when a network link is unavailable or when a host crashes. QRPCs are stored in a stable log so that if links become unavailable or the sender or receiver crashes, they can be replayed upon recovery. They are deleted from the log only after a response has been received from the server.

The Rover network scheduler is responsible for draining the log and forwarding QRPCs to servers. It can transmit a group of related operations together in a single message to improve transmission efficiency. It supports prioritization of QRPCs, both within an application and between applications (see Section 4.4). It exploits the stable log and the application-supplied priorities to reorder transmission of QRPCs. Reordering is important to usability in an environment with intermittent connectivity, as it allows the user (through applications) to identify the operations that are most important. For example, a user who has been disconnected for several days may have a significant amount of work stored in the log. By allowing transmission reordering, Rover allows users to choose to send important information over a slow or expensive link, and delay other sends until faster or less expensive communication is available.

The split-phase communication model offered by QRPC provides several key benefits, including the ability to use separate or different communication channels for the request and response and allowing the communication channel to be closed during the intervening period. Several existing and new wireless technologies offer asymmetric communication options, such as 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 expectation is that the waiting period will be fairly long (*e.g.,* due to server load or operation complexity) and that the client is charged for use of the channel based upon connection time.

Another potential advantage of QRPC is that it can be used to stage messages near their destination. This addresses the problem of a disconnected client, saturated server, or failed link. It is useful because it allows for faster data transmission once the condition preventing data progress is removed. For example, staging data near the next expected contact point of a mobile computer will reduce the round-trip time penalties incurred during transmission over a large distance.

3.3 Relocatable Dynamic Objects

Rover applications can employ RDOs in at least four ways:

1. Graphical User Interfaces (GUIs) can be downloaded and customized for a mobile host. Using GUIs over wide-area or low bandwidth networks can be an exercise in frustration. By migrating a GUI to the client, input events can be processed locally, even if the client becomes disconnected. In addition, these interactive programs can change their appearance depending upon the available display resources and available bandwidth for fetching large images.

2. A server can send a decompression procedure along with compressed data in order to obtain application-specific compression, which can reduce network bandwidth consumption.

3. Applications can implement their own consistency protocols. For example, like other calendar tools, our distributed calendar application schedules a meeting by starting with a set of choices and verifying the availability of a time slot in each attendee's calendar. It performs the verification by examining each calendar for a matching free time period. After it has found such a period, it marks the time period as *tentatively* scheduled on each calendar. If, after marking all calendars, no irreconcilable conflicts have occurred, it marks the meeting as scheduled. Otherwise, it deletes the temporary time period and repeats the scheduling attempt with an alternative time period. Eventually, if no satisfactory time is found, the operation aborts and returns an indication of its failure to the client. This two-phase commit process is an example of how the Rover architecture allows application designers to build applications with the desired degree of consistency for application-level operations.

4. Clients can export computation to the server. 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. In the case of a dynamic data stream, without an RDO executing at the server, the application would either have to poll or rely upon server callbacks. While this might be acceptable during connected operation, it is not acceptable during disconnected operation or with intermittent connectivity. Furthermore, every change to the data would have to be returned to the client for processing. With RDOs, the desired filtering or processing can be performed at the server, with only the processed results returned to the client. For example, a financial client making decisions based upon a set of stock prices could construct an RDO that would watch prices at the server and report back only significant changes. This would significantly reduce the amount of information transmitted from the server to the client.

An added benefit of RDOs is that the Rover toolkit can be dynamically extended. Rover starts as a minimalistic "kernel" that imports functionality on demand. This feature is particularly important for mobile hosts with limited resources.

Implementing RDOs involves three somewhat conflicting goals: (1) safe execution, (2) portability, and (3) efficiency. These goals can be achieved by using approaches such as code inspection [12] and sandboxing [58], pointer-safe languages [18, 47], or code interpretation with limited environments (*e.g.,* Safe-Tcl [43]). In our initial implementation we use interpreted Tcl.

3.4 User Notification

Because the mobile environment may rapidly change from moment to moment, it is important to present the user with information about its current state. The Rover toolkit provides applications with information about the environment for presentation to the user. Applications may use either polling or callback models to determine the state of the mobile environment.

The environment consists of the state of imported RDOs and the state of the network. The RDO life cycle consists of being imported but not yet present, being present in the local environment, being modified locally by method invocations, being exported to the server, being reconciled or committed, and being evicted from the environment. A network interface may be present or absent, and, if present, is characterized by cost and quality of service: latency and bandwidth. Applications can register methods to be invoked for each change in the state of an RDO or of the network.

Applications reflect these notifications to users. For example, in an E-mail application, messages that have been imported but have not yet arrived locally may be "grayed" or marked "loading." A calendar application may mark tentatively scheduled events with a particular color. When a network interface becomes available after a period of disconnection, the user may be queried as to whether to fetch E-mail. Alternatively, a user may instruct an E-mail reader to fetch new messages at the next opportunity.

3.5 Programming with Rover

To create an application with Rover, a programmer defines objects for the data types manipulated by the application. Methods that update an object should include code for conflict detection and resolution. In addition, a programmer defines an object to encapsulate the return value of an export operation. The return value may be another object that may be created at runtime.

The interface between Rover and its applications contains four primary functions: **create session**, **import**, **invoke**, and **export**. **Create session** is called once to set up a connection with the local object-cache manager and return a session identifier. It also provides authentication information, which is used by the cache manager to authenticate client requests that are sent to a Rover server.

To import an object, an application calls **import** and provides the unique identifier for the object, the session identifier, a callback, and arguments. In addition, the application specifies a priority that is used by the network scheduler to reorder QRPCs. **Import** returns a promise [37]. Applications can wait on this promise or continue computation. The callback will be invoked upon arrival of the imported object.

The current implementation also has a **load** operation that is an import combined with a call to create a process. Using load, applications can import RDOs that need a separate thread of control. Upon their arrival, Rover creates a separate process that runs the RDO. The application and the RDO communicate through the Rover object cache. The reason for a separate load operation is that the underlying operating system (Linux, a UNIX operating system) does not support multiple threads per address space and limited dynamic linking. In a future implementation, **load** will be directly incorporated within **import**.

Once an object is imported, applications **invoke** methods on it. These methods read and modify objects in Rover's cache. Rover transparently queues RPCs for each update in the log when an object is exported.

To export local changes to an object, an application calls **export**, and provides the unique identifier for the object, the session identifier, a callback, and arguments. Like **import**, **export** returns a promise. Upon arrival of the responses to the requests, callbacks are invoked. In addition, the promise contains a pointer to the list of return values. The application traverses this list of response objects, marking objects as committed, updating the display, and checking for unresolved conflicts.

Porting an existing UNIX application (such as those described in Section 6) requires replacing the file model with the Rover object model and dividing the application into a client part and a server part. As discussed in Section 6, porting applications to Rover is relatively straightforward.

4 Rover Architecture

Rover's architecture addresses six important issues: (1) allowing useful work to be done in the presence of network disconnections by permitting applications to be loaded into mobile hosts and by queuing fetches and updates as appropriate; (2) providing a means for varying an application's workload between servers and clients, both to address computationally under-powered clients and overloaded servers; (3) efficiently using available network connectivity from both quality of service and cost standpoints; (4) taking advantage of available network infrastructure to optimize the staging of information; (5) exposing network connectivity to applications and permitting applications to be involved in connectivity related decisions; (6) providing an efficient application development environment. The Rover architecture is structured as three layers and consists of four components: the access manager, the object cache, the operation log, and the network scheduler (see Figure 2); we discuss each component in turn.

4.1 Access Manager

Each client machine has a local Rover *access manager*, which is responsible for handling all interactions between client applications and servers and among client applications. The access manager services requests for objects, mediates network access, manages the object cache, and logs modifications to objects. Client applications use the access manager to import objects from servers and cache them locally. Applications invoke the methods provided by the objects and, using the access manager, make changes globally visible by exporting the changes to the objects back to the servers.

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

The access manager also handles connection requests from servers. This is particularly useful when establishing a network connection is a costly operation. For example, an E-mail service could send a pager message to a disconnected mobile host when incoming E-mail is received. The mobile host would then initiate a network connection (*e.g.,* by placing a cellular call) and retrieve the messages. The key advantage is that the mobile host does not have to perform an expensive polling operation to receive timely updates.

Failure recovery is also handled by the access manager. This task is eased somewhat by our use of both a persistent cache and an operation log. After a failure, the access manager requeues 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. Since the original application that issued the QRPC is no longer running, there is no available delivery target. Restarting the application does not solve the problem, as the appropriate action may depend upon lost state. One solution we are considering is extending the persistence model to incorporate portions of client application state. However, this solution would have a significant impact on application performance.

4.2 Object Cache

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 applications do not usually directly interact with the object cache. The Rover toolkit automatically maps **import** and **export** operations onto objects cached both within the application's address space and the access manager's address space. When a client application issues an **import** or **export** operation, the toolkit satisfies the request based upon whether the object is found in a local cache and the consistency option specified for the object.

Applications control the consistency model for the objects they import. The object cache offers several options for maintaining the consistency of an object relative to its primary copy. The options are: *uncacheable*, *immutable*, *verify-before-use*, *verify-if-service-is-accessible*, *expires-after-date* (*i.e.,* a lease), and *service-callback*. Uncacheable objects are objects that cannot be cached (*e.g.,* the result of an **export** operation). Immutable objects are those that are guaranteed not to change. Verify options control whether Rover will verify that the object is up-to-date relative to the service's primary copy before invoking a method on it. With verify-before-use Rover always verifies the object, while with verify-if-service-is-accessible Rover verifies only if the service is currently reachable under the application's requirements for communication costs. With expires-after-date, Rover automatically removes the object from its cache when the specified date has passed. With service-callback the Rover server will attempt to notify the Rover client if the object changes.

Cached copies of objects are updated in one of three ways: by importing fresh copies from the home service, by applying method invocations provided by the home service to reflect updates by other clients, and by applying method invocations from local applications. Updates performed by local applications allow applications on a single host to share data without requiring network communication. The cache distinguishes between updates from the home service and local applications. Since the home service has the primary copy of an object, information from it represents the "permanent" state of the object (*i.e.,* the data is not dependent upon a pending operation). Updates from local applications, however, are marked as tentative, until they are propagated to and accepted by the home service. Applications can specify whether they will accept tentative data when importing an object.

An essential component to accomplishing useful work while disconnected is having the necessary information locally available. This goal is usually accomplished during periods of network connectivity by filling the cache with useful information [29]. There are two issues here: when and what to prefetch.

Applications decide which objects to prefetch. The usability of Rover will be critically dependent upon simple user interface metaphors for indicating collections of objects to be prefetched. Requiring users to directly list the names of objects that they wish to prefetch is inherently confusing and error-prone. Instead, Rover client applications provide the access manager with prioritized prefetch lists based upon high-level user actions. For example, *Rover Exmh* automatically generates prefetch operations for the user's inbox folder and recently received messages. It also prefetches additional folders and messages based upon observed user behavior or user selection.

The network scheduler decides when to prefetch objects based upon network activity, available bandwidth, and transmission cost. For example, while waiting for a server response on an idle communication channel, the scheduler will send prefetch requests based upon their priority and the channel's communication costs. Clearly, overly aggressive prefetching will impose an unacceptable overhead.

For systems that provide for disconnected operation, prefetching is a requirement. During disconnected operation, Rover appli-

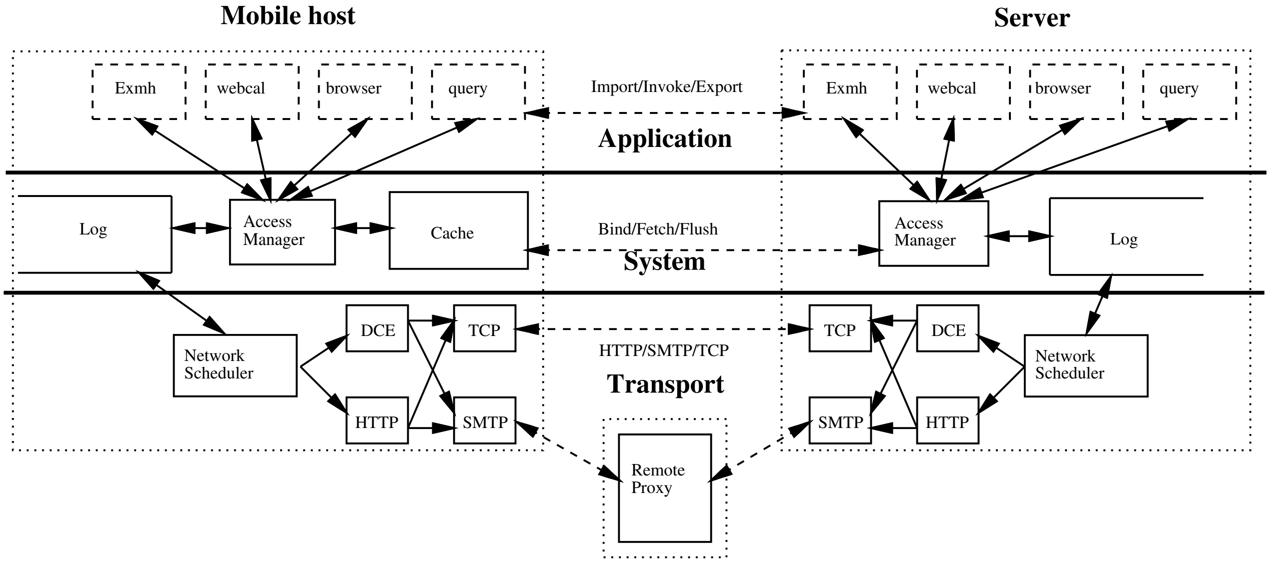

Figure 2: The Rover architecture consists of three layers (application, system, and transport) and four key components (access manager, object cache, operation log, and network scheduler).

cations will not block on an **import**, nevertheless, it may not be possible for a user to complete a desired activity (*e.g.,* viewing). Wireless connectivity may provide a fallback mode of operation for any objects that were not prefetched during a previous period of connectivity. However, there will always be instances where connectivity is unavailable or expensive. Thus, it is critical that prefetching during periods of connectivity is sufficiently aggressive.

4.3 Operation Log

Once an object has been imported into the client application's local address space, method invocations without side effects are serviced locally by the object. At the discretion of the application, method invocations with side effects may also be processed locally, inserting tentative data into the object cache. Side-effecting operations also insert a QRPC into a stable operation log located at the client. Upon server reception, an operation is performed on the server. Log flushing is asynchronous with application activity.

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

The price of local updates is that the client's and server's copies of an object will diverge over time. At some point, network connectivity will be restored and modifications to exported objects will have to be reconciled with any changes to the server's copies.

In most cases, we expect that changes will have been made only to the server's copy or only to the client's copy, but not to both. The likelihood of conflicting modifications is strongly application- and data-specific. For example, it is unlikely that a user's private data would be modified by another server or client. On the other hand, it is quite likely that a public calendar of interesting talks would be modified by others. Our experience with Rover applications is that handling independent updates is a rather simple task. While conflicting updates are more difficult to handle, we believe that our approach of allowing applications to specify conflict resolution procedures on a per-operation basis will help in two ways. First, it will help reduce the granularity of updates below that found in traditional filesystem-based approaches, which will reduce the likelihood of a conflict and provide more information about the conflict. Second, per-operation conflict resolution offers more flexibility in handling a conflict.

We have already seen benefits from both effects in our use of a distributed calendar application. The pre-Rover version is based upon shared calendars stored in files. Updates to the calendars have a window of vulnerability when two or more users try to update the same calendar. The Rover version of the calendar uses two techniques to reduce the likelihood of a conflict: it reduces the granularity of conflict resolution to the individual appointment and notice level and it handles certain combinations of simultaneous changes (*e.g.,* one user changing the time of a meeting while another changes the topic).

There will be cases where conflicts cannot be resolved. To provide for such occurrences, applications can include application-specific information in each operation that can be used by the application to provide the user with a clear indication of the cause of a conflict resolution failure.

Another issue that Rover addresses with an application-specific approach is the growth in size of the operation log during disconnected operation. The ability to convey application-level semantics directly to servers is an important functional advantage, especially in the presence of intermittent connectivity. However, it may lead to an operation log that grows in size at a rate exceeding that of a simple write log. To address this potential problem, Rover directly involves applications in log compaction. Applications can download procedures into the access manager for manipulating their log records. For example, an application can filter out duplicate requests (*e.g.,* duplicate QRPCs to verify that an object is up-to-date can be reduced to a single QRPC). In addition, applications can apply their own notion of "overwriting" operations to the operations in the log.

4.4 Network Scheduler

The network scheduler groups related operations together for transmission. We leverage off the queuing performed by the log to gain transmission efficiency. The result is a reduction in per-operation transmission overhead and an increase in connection efficiency by amortizing connection setup and teardown across multiple requests and responses; this amortization is especially important when connection setup is expensive (either in terms of added latency or dollar cost).

The network scheduler may reorder logged requests based upon two primary criteria: the session's consistency requirements and application-specified operation priorities. Using these criteria, the scheduler provides an ordering for flushing operations from the log.

Rover applications prioritize requests by using a "quality of service" model to specify delivery importance. This model closely matches other familiar service models, like postal delivery. A person requiring long-term postal delivery uses a 32 cent stamp. They recognize that the mail may be delivered anytime within a day or two or perhaps more. Imposing a two day delivery deadline increases the cost to 3 dollars. Delivery overnight costs 10 dollars; same-day delivery is 100 dollars. This same "pay more for faster delivery" model can be applied directly to message delivery between clients and servers. The user's desire for delivery speed can be directly balanced against transmission costs, especially considering that the cost of communication is considerably higher for wide-area wireless communication.

Quality of service is also a factor in the choice of delivery transport. For example, the requested quality of service would influence a choice between SMTP and TCP.

Client applications that import objects are not the only source of network traffic. The object cache generates traffic for revalidation of cached objects. In addition, client applications may issue prefetch requests. Such background traffic can be used by the network scheduler to fill gaps in primary network traffic and optimize link utilization of connection-based networks.

The Rover network scheduler controls when and which communication interfaces are opened and what should be sent over the interface. The scheduler performs a matching function between the quality of service desired for a request and the available network transport options (based upon information provided by the communication layer). Likewise, objects sent to servers also include cost information which is used to determine when to send results back to the client. Thus, returning to the previous financial application example, a user can specify the importance of receiving timely information, which can then be conveyed in the object sent to the server.

We are continuing to investigate when to open a communication interface. Knowledge of the user's willingness to pay a given amount is not sufficient. What is needed is knowledge about expected future communication options. For example, consider a person traveling to a remote destination via an airplane. While on the plane, the person performs several operations, generating a log of changes. The person assigns a moderate level of delivery funding per operation, a level that is sufficient that the aggregate sum of funding is above the threshold for use of an expensive wireless link. However, at the end of the flight, significantly cheaper links will be available. Without such future knowledge, a naive scheduler will yield less than optimal cost efficiency.

Function	Operation
Rover_AddWrite	Mutate an RDO
Rover_Export	Export a modified RDO
Rover_Flush	Flush a cached RDO
Rover_GetDV	Get an RDO's dependency vector
Rover_GetPid	Get an RDO's process ID
Rover_Import	Import an RDO
Rover_LoadApplication	Import an RDO and execute
Rover_MarkPermanent	Mark an operation permanent
Rover_NewSession	Create a new session
Rover_PendingWrites	Get a count of pending writes for an RDO
Rover_PromiseClaim	Claim a promise
Rover_QRPC	Issue a non-blocking QRPC
Rover_RPC	Issue a blocking RPC
Rover_Shutdown	Shut down a client application
Rover_Update	Update the Rover RDO cache from a local RDO

Table 1: Rover library functions.

5 Rover Implementation

Since Rover is a research platform, we have chosen a layered, modular approach to implementation, which favors flexibility over performance. Our intention is to use the prototype as a testbed for a wide range of ideas.

Rover is implemented on IBM ThinkPad 350C and 701C laptops running Linux 1.2.8, DECstation 5000 workstations running Ultrix 4.3, and SPARCstation 5 and 10 workstations running SunOS 4.1.3_U1. The Rover server can execute either as a Common Gateway Interface (CGI) application of NCSA's *httpd* 1.4.2 server (running on Ultrix and SunOS in the non-forking, pool of servers mode), or as a standalone TCP/IP server. Rover server code consists of approximately 1500 lines of C code.

The Rover client code relies upon CERN's pre-release multithreaded version of the Web Common Library (version 3.0pre3) for HTTP support. The client code consists of approximately 10000 lines of new C code plus an additional 200 of modifications to the Web Common Library.

Network connectivity is provided by 10 Mbit/s switched Ethernet, wireless 2 Mbit/s WaveLAN, and dial-up lines. In addition to HTTP over TCP/IP, we have implementations that uses SMTP over TCP/IP and raw TCP/IP.

Our primary mode of operation is to use the laptops as clients of the workstations. However, we also use the workstations as clients of other workstations.

The Rover implementation is composed of three primary layers: server and client applications, system support, and transport. We discuss each layer in the following sections.

5.1 Client and Server Application Layer

The highest layer in Rover is the client and server application layer. Rover applications consist of Tcl/Tk scripts [43] and binary applications. The Tcl/Tk scripts are interpreted using a Tcl/Tk interpreter environment that has some simple C extensions to support RDOs (see Table 1) and linked with a Rover library. The library provides functions for communicating with the Rover access manager. Using the C extensions, server applications construct Tcl/Tk RDOs in response to client requests; the RDOs are then transported by Rover to client applications. Likewise, client applications construct modification operations, which are kept by Rover in a per-object data structure. When an application calls **export**, these modifications operations are turned into QRPCs and stored in the stable operation log. Binary applications are directly linked with the Rover communication functions.

We chose this implementation approach because it greatly simplified development of our initial prototype. However, we recognize that this is an interim solution. In our prototype, we are not using a secure or safe language. Instead, limited security is provided by executing some RDOs in a separate address space, by interpreting RDOs written in Tcl, and cryptographic authentication of all messages exchanged between the access manager and Rover servers.

5.2 System Support Layer

System support in Rover consists of a set of client-side and server-side modules. The server modules exist mainly as library routines invoked by incoming requests from clients.

The Rover server is a secure setuid application that authenticates requests from client applications, mediates access to RDOs, and provides a Tcl/Tk execution environment for RDOs from client applications.

We provide two implementations of Rover servers. One is compatible with the Common Gateway Interface (CGI) [40] of standard, unmodified HTTP compliant servers (*e.g.,* CERN or NCSA's httpd servers). The other implementation is a standalone TCP/IP server which provides a very restricted subset of HTTP. Both servers offer identical functionality and communication interfaces to Rover client applications.

The HTTP server forks and executes a new image of the CGI implementation of the Rover server. As a result, any state that needs to be persistent across connections must be re-read for each connection. In addition, there is a significant amount of overhead associated with using the CGI interface (*e.g.*, the cost for the `fork` and `execve` required to start the application). These were the motivating reasons behind our choice to also implement the standalone TCP/IP server.

On the client side, we use a local client-server model: each Rover application executes in a separate address space and communicates via Local Remote Procedure Call (LRPC) with the local Rover access manager. To improve efficiency, a copy of each imported object and RDO is cached within a client application's address space. This copy is unavailable to other applications on the mobile host, but, if desired by the application, will be kept consistent with the global Rover object cache.

The access managers are multithreaded with non-preemptive servicing of client application requests and incoming responses from servers. In addition, they have several background housekeeping threads that perform operations such as log flushing, cache cleaning, and revalidating cache entries.

The client-side implementation is designed to be used in a bootstrapping mode, where the access manager loads application scripts and binaries from a Rover server and starts executing them.

The Rover client cache is implemented using ordinary UNIX files: one file contains a list with the cache's contents, and one file is used for each cached object. The content list is stored in memory in a hash table. When an object is entered into the cache, it is first stored in a memory-resident table. A background process periodically flushes new or modified entries to disk and updates the disk copy of the content list. When the access manager receives an **import** request, it queries the cache; the access manager first searches the memory-resident table before searching the content list. For cache maintenance purposes, the cache records several useful attributes for each entry: when the object was originally loaded, the cost to load the object (currently, this is the time measured from when the request was sent to a server to when the response was received), and the last time an application referenced the object.

The Rover client log is also implemented as an ordinary UNIX file. Rover uses the file to save log entries, track the log entries that have been sent to servers, and record the log entries that have been deleted (either because a response has been received or because an application deleted the request). When an application calls **export**, a set of QRPCs is entered into the log and Rover performs both a flush and a synchronize operation to force the new QRPCs to the disk. Thus, the flush is on the critical path for message sending. Our prototype implementation favors simplicity over performance: it does not perform any compression on the log [29] and it does not employ efficient techniques for implementing stable storage (*e.g.,* Flash RAM [13] or group commit [20]).

5.3 Transport Layer

The transport layer is the lowest layer of Rover, and is itself split into two levels. The upper portion of Rover's transport layer converts Rover URNs into an HTTP POST message with a URL specifying the server application (*e.g.,* http://www.pdos.lcs.mit.edu/cgi-bin/rover/e-mail) and a message body containing the RDO request method and an authenticator. The server application verifies the authentication information and, for **import** requests, generates a copy of the desired RDO and returns it in the HTTP response. For **export** requests, the server constructs an interpreter and interprets the operation log provided in the data part of the request. URNs have the advantage that they add a layer of indirection to the resource location problem. Thus, we can move resources based upon varying requirements (*e.g.,* server load or availability) without exposing such changes to end users.

The lower layer consists of a network scheduler and communication protocols. Messages can be sent over both connection-based protocols (*e.g.,* TCP/IP) and connectionless protocols (*e.g.,* SMTP) [9, 24] and IP or non-IP based networks. The choice is handled by the network scheduler and is based in part upon the requested quality of service. The implementation of the network scheduler has several queues for different priorities and it chooses a network interface based on availability and quality. Our current implementation of the network scheduler does not rely upon an economic model for making reordering decisions.

The implementation assumes that networks can reorder, drop, or arbitrarily delay messages; this is especially important for SMTP delivery. The transport layer uses unique identifiers and retransmission to provide at-most-once delivery semantics.

With TCP, the network interface directly communicates with the server. For each QRPC, it opens a TCP connection, sends the data, and then closes the connection. We chose this implementation instead of an implementation that amortizes the cost for setting up the connection over multiple QRPCs, since this what the HTTP server currently does.

With SMTP, it communicates with a local mail delivery agent, which will forward the request on to remote proxy. The remote proxy is implemented as a mail filter program. For ordinary Rover requests, it opens a TCP connection to the requested server, sends the requests, receives the response, and uses SMTP to send the response back to the original sender where another mail filter delivers the response.

We chose to use SMTP as one of the communication protocols because it allows us to experiment easily with split-phase communication and staging data close to the destination. The use of SMTP has its drawbacks. In particular, mail delivery can be plagued with intermittent delivery problems, mostly relating to variable delivery latencies or intermediate gateways modifying message content (*e.g.,* truncating messages and lines, reformatting content, or mutating lines containing special sequences). This complicates the Rover implementation as we must encode any data being sent and perform extra checks to ensure at-most-once delivery. For these reasons, we view SMTP as an interim solution that is most useful because it is widely deployed. However, our architecture is structured in a manner that will allow us to seamlessly incorporate support for any future alternatives.

6 Rover Applications

To demonstrate our ideas and the Rover toolkit, we have implemented three applications: a distributed calendar, an E-mail browser, and a Web browser proxy. The suite was chosen to test several hypotheses about our ability to reasonably meet users' expectations in a mobile, partially-connected environment. These applications represent a set of applications that mobile users are likely to use. Because RDOs affect the structure of applications, we felt it was important to test our ideas with complete applications in addition to using standard quantitative techniques.

6.1 Rover Exmh

To investigate the application design space of Rover, we have developed *Rover Exmh*, a port of Brent Welch's *Exmh* Tcl/Tk-based E-mail browser. Parts of the browser itself and messages are loaded from a remote server in the form of RDOs. These RDOs create the browser interface and load the user's E-mail messages. In addition, RDOs are used to implement application-specific consistency and prefetching strategies. This was the first application we implemented after implementing the core Rover client and server functionality. It took us approximately three person-weeks to complete the initial port of *Exmh* to Rover.

Rover Exmh is organized as follows. The core of the browser is unchanged from *Exmh* and is implemented as Tcl/Tk scripts. *Exmh*'s file system-based interface to the E-mail message handling system, *mh*, was replaced with an object-based interface, which is contained in RDOs retrieved from an HTTP server when the user first loads the browser. The browser imports the user's inbox (as an RDO) and list of recently visited folders (also as an RDO). After the list RDO is imported, it automatically issues prefetching **import** operations for each of the folders on the list. When the user selects a message for viewing, if it is not already present, it is imported from the server and marked as loading in the folder display. The messages' states are updated as the messages arrive and become ready for viewing. The browser's interface runs entirely on the client, so that button clicks and typing result only in local method invocations, and are not exported to the server until after the user is done. This effectively demonstrates the value of RDOs for reducing client dependence on connectivity and bandwidth.

For the *Rover Exmh* server, we added a thin veneer of Tcl/Tk commands so that RDOs could invoke the appropriate E-mail message handling commands. We also added support for detecting and resolving conflicting modifications to a user's folders and generating updates for informing clients of changes.

In terms of GUI issues, we modified *Exmh*'s display functions to show the sizes of mail messages and to indicate (using color) which messages were locally cached. We used RDOs to provide this dynamically updated functionality.

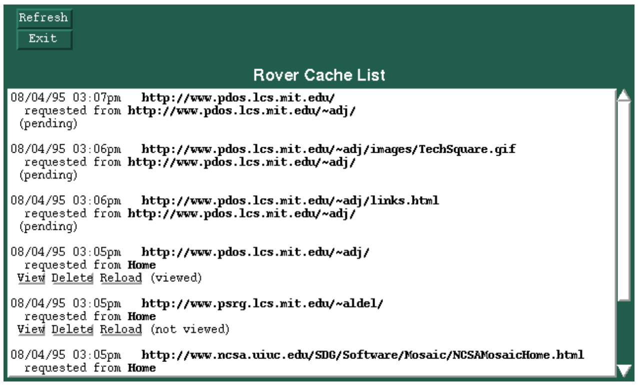

Figure 3: The Rover Web Browser Proxy log display with a number of QRPCs and their status. Entries can be deleted, viewed, or reloaded.

We considered alternative approaches, such as using a POP server. However, POP only provides message delivery and temporary message storage. It does not work well in an environment where there is no shared message storage. For example, in order for a user to access messages from both a mobile host and a fixed host, the user must resort to temporarily storing messages in the POP server when accessing them from the mobile host. Using Rover, the Rover server is the repository for the user's mail and ensures that the user sees a consistent view from any client.

6.2 Rover Webcal

Rover Webcal is a port of *Ical*, a publicly available Tcl/Tk and C++ based calendar and scheduling program written by Sanjay Ghemawat. *Ical* provides an X interface for maintaining a calendar, which contains a set of items. An item is either an appointment or a notice. An appointment starts and ends at particular times of day, while a notice does not have any starting or ending time. Notices are useful for marking certain days as special; for example, a calendar may contain a notice for April 15th indicating that taxes are due.

Under *Ical*, users sharing a calendar must share a file system. Consistency is provided on a per file (calendar) basis. *Ical* periodically checks the modification date of calendars so that it can display the most recent information. Conflicts may occur when two or more users attempt to simultaneously update the same calendar. When a conflict occurs, the user is given the choice of overwriting the changed calendar or flushing his or her own changes. The likelihood of the race condition being a problem is a function of how often clients modify calendars and how often they check calendars for changes. In practice, on a distributed file system, there has not been a problem. In a disconnected environment, on the other hand, we expected that this form of conflict would be extremely likely. As such, we decided to move the consistency granularity to the individual item level.

In *Rover Webcal*, each individual item is an RDO and is used as the unit of consistency control. *Rover Webcal* uses RDOs in place of items and calendars. Users can schedule events (*e.g.,* modify or create new items or calendars), which will then be marked as tentative. Once the changes have been accepted by the server, they will be marked as permanent.

In choosing the item as unit of consistency we address two issues: update conflicts in the form of multiple changes to the same

Rover Program	Base client code	Rover client code	Server code
Exmh	24000 Tcl/Tk	250 Tcl/Tk 120 C	2600 C
Webcal	33000 C++ and Tcl/Tk	4400 C++ and Tcl/Tk	5600 C++ (from client) 660 C and Tcl/Tk
Web Proxy	none	250 Tcl/Tk 2800 C	none

Table 2: Lines of code changed or added in porting *Exmh* and *Webcal* to Rover and implementing the *Rover Web Browser Proxy*.

calendar, and propagation of changes to general interest calendar entries (*e.g.,* meetings and talks) to mobile hosts.

Update conflicts are avoided in many cases because most calendar changes are to independent items. However, we can also handle certain classes of changes to the same item (*e.g.,* one user changes the item's time, while another user changes the content). In addition, we have flexibility in the choice of action to take in the event of an irreconcilable conflict. For example, when scheduling meetings, the head of a department could mark their changes as always overriding any other changes made by others in the department.

The second issue addressed by changing the granularity of consistency control is the timely propagation of changes to affected users. This issue is addressed both through the use of server callbacks and by the need for the server to only send RDOs for changed attributes (and not the complete item or calendar). This significantly reduces network traffic and network bandwidth requirements.

To port *Ical* to Rover, we split *Ical* into a client and server by placing the GUI, calendar, and item handling code in the client, along with an interface to Rover. We created the server by adding a thin veneer of Tcl/Tk commands to *Ical*'s file reading, parsing, and writing code. We also added conflict detection and resolution code and code to generate update information for clients registered for callbacks.

We are also exploring the use of RDOs to perform group scheduling functions as we consider this to be an good example of where a variable division of labor is useful. When a mobile host is well connected, it could handle the scheduling function; likewise, when it is only partially connected, the scheduling function could be off-loaded to a server.

We do not have an accurate measurement of the time to port *Ical* as the porting was done concurrently with the initial client and server implementations.

6.3 Rover Web Browser Proxy

The *Rover Web Browser Proxy* [10] is a unique application. It will interoperate with most of the popular Web browsers. Using it enabled us to rapidly produce one of the first full-function browsers that allows users to "click ahead" of the arrived data by requesting multiple new documents before earlier requests have been satisfied. The proxy intercepts all web requests and if the requested item is not locally cached, it 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 proxy will also directly control NCSA's *Mosaic* [41] and NCC's *Netscape Navigator* [42] browsers using their remote control interfaces. Cached Web documents are used whenever possible, to allow for fast access in the absence of a network connection. If an uncached file is requested and the network is unavailable, an entry is created in a displayed list of outstanding and satisfied requests. The list is actually created by an RDO (Figure 3 shows an example display). The display exposes the object cache and operations log directly to the user and allows the user limited control. For now, the cache is mostly managed explicitly by the user (hence the "Delete" and "Reload" buttons); this seems to be a fairly reasonable solution, especially in the case of results from form submissions, in which only the user can decide for how long the requested data is useful. One automated feature is prefetching: the delay between the user's request and its arrival at the remote proxy is used as an estimate of the network latency. If the delay is above a user-specified threshold, documents that are directly accessible from the one requested are prefetched.

The design of the proxy is general enough that it will allows us to experiment with using RDOs to dynamically generate web pages (*e.g.,* to ship Dynamic Documents [26] or fragments of databases to mobile hosts).

Our *Rover Web Browser Proxy* has been used to interact with Web resources through the SMTP-based transport layer, and has been found to work reasonably well even on slow or very infrequently connected networks, as well as on non-IP networks that do not permit HTTP connections. A certain amount of adaptation is required on the user's part. Since documents may arrive in an order different from that in which they were requested, the user must be able to discover which resources are available at a given time. The dynamically updated display list appears to do a good job of allowing this.

Development of the proxy also occurred during initial system development, making it difficult to determine the amount of time spent implementing it. However, an initial prototype was implemented within a few person-weeks.

6.4 Discussion

The three applications illustrate how RDOs can be used to adapt workstation applications to a mobile environment. The web browser uses RDOs to create the GUI. *Rover Exmh* uses RDOs to implement the user interface, resolve conflicts on updates of folders at the server, and prefetching. *Rover Webcal* uses RDOs to provide the GUI, to perform conflict resolution at the server, and to propagate changes back to user. A generalization of *Rover Webcal* that is under development uses RDOs to schedule appointments at the server. All applications use QRPCs for lazily fetching and updating objects and RDOs.

As can been seen in Table 2, the three applications also illustrate that porting file system-based workstation applications to an object-based Rover model is relatively easy. For example, porting *Exmh* and *Ical* to Rover required simple changes to 10% and 15% of the lines of code respectively. Most of these changes came from replacing the file systems call with object invocations; these modifications in *Rover Exmh* and *Rover Webcal* were made almost independently of the rest of the code.

The implementation of the *Rover Web Browser Proxy* provided a unique demonstration of how easily we could construct Rover applications that easily interoperate with existing applications. The proxy was initially developed for Mosaic; however, adding support for Netscape was a simple operation.

Server:SS 5/70 Client	Transport	TCP Latency null RPC	TCP Throughput 1 MByte
SS 5/70	Ethernet	5	5.2
TP 701C/75	Ethernet	8	5.0
	WaveLAN	11	1.03
	CSLIP14.4	380	0.025
	CSLIP2.4	2100	0.002

Table 3: The Rover experimental environment. Latencies are in milliseconds, throughput is in Mbit/s.

7 Results

We designed a set of experiments to validate our ideas and to measure how effectively the Rover toolkit meets our goals. In particular, the experiments test the following hypotheses:

1. QRPC performance is acceptable even if every RPC is stored in a stable log, or if SMTP is used as the transport protocol.
2. Caching RDOs reduces latency and bandwidth consumption.
3. Migrating RDOs provides Rover applications with excellent performance over moderate bandwidth links (*e.g.,* 14.4 Kbit/s dial-up lines) and in disconnected operation.

In this section, we first provide details on our experimental methodology and establish the baseline for QRPC performance. Then, we test each hypothesis in turn.

7.1 Baseline Performance

Our test environment consisted of a single server and multiple clients. The server was a SPARCstation 5 (70Mhz microSPARC-II) workstation running SunOS 4.1.3_U1 as the server. The Rover server ran as a standalone TCP server. The clients were IBM ThinkPad 701C laptops (25/75Mhz i80486DX4) running Linux 1.2.8. All of the machines were idle during the tests. The network options consisted of switched 10 Mbit/s Ethernet, 2 Mbit/s wireless AT&T WaveLAN, and Serial Line IP with Van Jacobson TCP/IP header compression (CSLIP) [25] over 14.4 Kbit/s and 2.4 Kbit/s dial-up links. To minimize the effects of network traffic on our experiments, we configured the switched Ethernet such that the server, the ThinkPad Ethernet adapter, and the WaveLAN base station were the only machines on the Ethernet segment and were all on the same switch port. The standard deviations for our measurements were within 10% of the mean values.

To establish the baseline performance for QRPC, we measured the latency and bandwidth of various representative network technologies. The results are summarized in Table 3. The table shows the latency for null ping-pong and the throughput for sending 1 Mbyte using TCP sockets over a number of networking technologies. The throughput over CSLIP is high, because the compression that is performed by the modem on ASCII data. The 1 Mbyte of ASCII data is very compressible (GNU's *gzip -6* yields a 14.4:1 compression ratio); since Rover is sending Tcl scripts (ASCII), we expect that Rover application will also observe the benefits of compression.

Protocol	Transport	Latency	Throughput
TCP	Ethernet	47	0.74
	WaveLAN	61	0.48
	CSLIP14.4	500	0.02
	CSLIP2.4	3600	0.001
SMTP	Ethernet	5600	0.02
	WaveLAN	5800	0.02
	CSLIP14.4	11000	0.007
	CSLIP2.4	43000	0.001

Table 4: Time in milliseconds for a null QRPC (a 290-byte request with a 5-byte reply). Throughput measured in Mbit/s for 16-Kbyte QRPC requests.

Transport	Latency	Throughput
Ethernet	8	3.6
WaveLAN	13	0.92
CSLIP14.4	420	0.022
CSLIP2.4	3100	0.001

Table 5: Time in milliseconds to open a TCP connection, send 290 bytes, receive a 5 byte response, and close the connection. Throughput measured in Mbit/s for 16-Kbyte requests.

7.2 The Performance of QRPC

To obtain insight into the performance of QRPC, we measured the costs for performing various sizes of QRPC operations with synchronous logging and compared these costs to the overhead of corresponding TCP operations. Table 4 displays the times for performing null QRPCs over various network technologies. For a null QRPC, approximately 290 bytes are sent and 5 bytes are received. We measured the throughput of Rover sending 16-Kbyte QRPC requests. We have chosen these data sizes because they reflect the sizes of small and large RDOs. The overhead of performing a QRPC includes the time for the access manager to synchronously log the RPC to stable storage, transmit the RPC, receive a null response from the TCP-based Rover server, and lazily delete the RPC from the log. For comparison, Table 5 gives the latency and throughput for TCP using the same data sizes as in the QRPC experiments.

We make two observations about the numbers for QRPCs on slower networks. By comparing the QRPC numbers to the TCP numbers, we see that the overhead imposed by QRPC is not significantly greater than the minimum overhead imposed by TCP, both for latency and throughput. Second, the cost of synchronously logging data is dwarfed by the underlying communications cost.

We also observe that latency and throughput on fast networks is sub-optimal. The difference between TCP and QRPC on these networks is due to the overhead of writing the log entry, which takes about 37 milliseconds. (In the throughput experiments, Rover writes the log using 16-Kbyte entries at 1.9 Mbit/s.) Most of this overhead can be simply eliminated by employing a more sophisticated implementation of the operation log: the current simplistic approach to implementing the log accounts for almost all the additional overhead over base TCP.

The performance when using SMTP is much worse than we initially expected. There are several reasons for the high overhead, including: the times to encode and decode the HTTP request and response, the time spent sending and receiving the request and response using *sendmail* (twice on each end), the time spent in

Type	Null RDO	16KB RDO
Tcl/Tk	0.06	0.06
Client	3.2	3.2
LRPC	7.4	20

Table 6: Time for an **invoke** on a local RDO invocation in milliseconds.

the remote proxy processing the request, and the time spent in the local mail filter processing the response. In addition, several of the applications require process forks and must be loaded into memory. We are investigating methods for reducing the overhead. However, it is important to remember that Rover's architecture moves the QRPC off the critical path for an application.

While the extra overhead of using SMTP gateways is obvious, there are benefits which may not be so evident. One is that SMTP is fundamentally a queued background process; it is more appropriate than the interactive HTTP protocol for fetching extremely large documents, such as video, which require large amounts of time regardless of the protocol. Indeed, we see from Table 4 that the differences in throughput are not nearly as pronounced for larger QRPCs on slower networks. Another advantage is that the IP networks required for HTTP are not always available, whereas SMTP often reaches even the most obscure locations. Of course, for small documents, HTTP is still highly preferable when it is available.

7.3 The Benefits of Caching RDOs

To obtain insight into the benefits (reduction in latency and bandwidth consumption) of caching RDOs, we compare the time for a local invocation with the time to export an RDO. First, we measured Rover's performance for three different kinds of local null invocations: Tcl/Tk, client, and LRPC. A local invocation occurs when the target RDO is located on the mobile host, either in the client application's cache or in the shared cache. A null RDO consists of 45 bytes of Tcl/Tk code and no data, while a 16KB RDO consists of 45 bytes of code plus an additional 16 Kbytes of data.

Table 6 lists the times for performing local null invocations. The Tcl/Tk measurement is the time for a null procedure call within the Tcl/Tk interpreter. The client measurement is the time to export an RDO from the local client application cache into the Tcl/Tk interpreter and perform a null procedure call on the RDO. The LRPC cost adds the additional step of performing a local RPC to the access manager to retrieve the RDO before exporting it and performing a null procedure call.

Our cost for crossing address spaces (LRPC) is approximately an order of magnitude greater than the underlying system cost (approximately 400 microseconds for sending and receiving a 4-byte message across a pipe). We believe that this is due to the simplistic approach we used to implement LRPC: it dynamically allocates buffers, copies the data several times, uses an expensive protocol for transferring messages across a pipe, and uses a naive marshaling scheme. All of these overheads can easily be eliminated.

We can now compare the local invocation time to the time to export an RDO. (We measure the time for an **export** instead of an **import**, because an **export** has to write the full RDO into the log.) Exporting an RDO consists of performing an LRPC from the application to the access manager with the updated RDO. The access manager constructs a QRPC for the updated RDO and queues the request with the RDO in the stable log. The network scheduler dequeues the request and issues the update to a Rover server. The server then constructs a small reply RDO and returns it to the mobile host. Finally, the access manager removes the QRPC from the log and notifies the client application.

Protocol	Transport	Latency	Throughput
TCP	Ethernet	59	0.36
	WaveLAN	75	0.28
	CSLIP14.4	555	0.02
	CSLIP2.4	4100	0.001
SMTP	Ethernet	5600	0.02
	WaveLAN	5800	0.02
	CSLIP14.4	11000	0.007
	CSLIP2.4	44000	0.001

Table 7: Time for an **export** of a 57-byte RDO in milliseconds. Throughput measured in Mbit/s for 16-Kbyte RDOs.

Table 7 provides the results for exporting small (47 code bytes and 10 data bytes) and large (50 code bytes and 16K data bytes) RDOs. Most of the overhead for exporting a remote RDO can be attributed to the cost for performing an LRPC and a QRPC. The remaining time is the communication transport time, the time to create the RDO, and the time to process the request within the access manager and the Rover server.

By comparing the time for an LRPC (7.4 milliseconds) with the time for the export of a small remote RDO (59 milliseconds), we conclude that LRPC is only a small fraction of the total cost of exporting a remote RDO. Therefore, caching RDOs can substantially reduce the latency of method invocations. Similarly, we conclude that caching can substantially reduce bandwidth consumption.

More interesting is the comparison between the time for a null RPC over TCP and the time for LRPC, since the TCP numbers give a lower bound on how efficient QRPC could be. Using the fastest transport protocol, a null RPC over TCP takes 8 milliseconds (see Table 5), which is already almost one millisecond slower than our inefficient implementation of LRPC. We therefore conclude that caching is worthwhile even when using the fastest transport protocols. With the slower networks, the gains from using caching become much larger: for a 14.4 dial-up line an LRPC is 56 times faster (7.4 versus 420 milliseconds).

All measurements were done using the standalone Rover server. Using the CGI-based Rover server with Ethernet/TCP adds approximately 110 milliseconds to the cost of exporting a small RDO. About half of the cost is from internal HTTP server processing, while the other half is the cost to fork and execute the Rover server. Although the price for backwards compatibility with HTTP servers is considerably high, it allows Rover to leverage off the extensive available WWW infrastructure.

7.4 The Benefits of Migrating RDOs

To understand the performance benefits of dynamically migrating RDOs in an environment with moderate bandwidth links and disconnected operation, we performed two system-level experiments. In the first experiment, we measured the time to perform a simple task using *Exmh* and *Rover Exmh*: starting the application and reading all of the messages in a user's inbox (eight messages, ranging in size from 0.79 Kbytes to 32.7 Kbytes, totaling 65.4 Kbytes). For the second experiment, we measured the time to perform a simple task using *Ical* and *Rover Webcal*: starting the application and viewing the appointments for a week's activities (the user's base calendar included 16 other calendars, ranging in size from 16 bytes to 20 Kbytes, totaling 79.5 Kbytes).

Environment	Transport	Time
X11R6	Ethernet	0:55
X11R6 (NFS)	CSLIP14.4	2:36
Xremote (Windows 3.1)	14.4 Kbit/s	9:08
Rover/App cached	CSLIP14.4	1:34
Rover/App cached	CSLIP2.4	7:56
Rover/Full cache	CSLIP14.4	1:06
Rover/Full cache	CSLIP2.4	1:37
Rover/Full cache	none	1:02

Table 8: Time to start *Exmh* and *Rover Exmh*, and read all messages in user's inbox. Times are minutes:seconds.

Environment	Transport	Time
X11R6	Ethernet	0:15
X11R6 (NFS)	CSLIP14.4	1:02
Xremote (Windows 3.1)	14.4 Kbit/s	3:20
Rover/App cached	CSLIP14.4	1:09
Rover/App cached	CSLIP2.4	10:16
Rover/Full cache	CSLIP14.4	0:33
Rover/Full cache	CSLIP2.4	1:11
Rover/Full cache	none	0:29

Table 9: Time to start *Ical* and *Rover Webcal*, and check a week's appointments and reminders. Times are minutes:seconds.

The results for the system-level experiments are summarized in Tables 8 and 9 respectively. As a baseline, we used the unmodified version of each application running on the server and using X over Ethernet to provide the user interface to the mobile host. For limited bandwidth connectivity, we used NCD's PC-Xware 2.0's *Xremote* running under Microsoft Windows 3.1. We also tested each unmodified application using NFS to access the server. In this test we ran the application locally on the mobile host, which used NFS to obtain and cache data (mail folder and calendar files).

For the Rover tests, we used Rover versions of each application running locally on the mobile host. We tested the performance when the application binary and supporting RDOs were locally cached. Thus, for *Rover Exmh*, we measured the time to retrieve the RDOs representing the inbox folder and the messages contained within it, while for *Rover Webcal*, we measured the time to retrieve the RDOs representing the user's calendar and the calendar items contained within it. The "full cache" numbers measure the time when, in addition to caching the application, all the RDOs encapsulating the data were locally cached. However, the network was used to validate that the RDOs, including the application code, were up-to-date. Finally, the fully disconnected case measured performance when all information was locally cached and validation requests were enqueued and logged to stable storage for later delivery.

We first compare the Rover applications in disconnected mode with the unmodified version of the application running on the server and using X over Ethernet to provide the user interface to the mobile host. When considering this comparison, it is important to recall the relative performance differences between the server (SPARCstation 5) and the mobile host (ThinkPad 701C), and that Rover logs validation requests. We see that the Rover applications performance is competitive with the unmodified applications (1:02 versus 0:55 for *Exmh* and 0:15 versus 0:29 for *Rover Webcal*). From this experiment we conclude that Rover delivers excellent performance in disconnected operation.

If we compare the unmodified applications running over Xremote with the Rover applications over a 14.4 dial-up line, we see that the Rover applications perform substantially better than the unmodified applications over Xremote (1:06 versus 9:08 for *Exmh* and 0:33 versus 3:20 for *Rover Webcal*). Even if the Rover cache is cold, the Rover applications still perform substantially better (1:34 for *Exmh* and 1:09 for *Rover Webcal*). What the numbers do not show is the extreme sluggishness of the user interface when using Xremote. Scrolling and refreshing operations are extremely slow. Clicking and selecting operations are very difficult to perform because of the lag between mouse clicks and display updates. Because of this, we could not run Xremote effectively over a 2.4 dial-up line, while the Rover applications ran fine over 2.4. From this experiment we conclude that dynamically migrating RDOs (in this case the GUI) delivers substantial performance benefits.

If we compare X11R6 (NFS) with Rover, we obtain a rough idea of how systems with caching but without migrating RDOs might perform. We see that the Rover applications run substantially faster than the unmodified applications on lower bandwidth links. With a 14.4 dialup-line the unmodified *Exmh* runs in 2:36, while *Rover Exmh* runs in 1:06. Similarly, unmodified *Ical* runs in 1:02, while *Rover Webcal* runs in 0:33. Again, most of the performance benefits are coming from Rover's capability to dynamically migrate the GUI using an RDO.

8 Conclusions

We have found that the integration of relocatable dynamic objects and queued remote procedure calls in the Rover toolkit provides a powerful basis for building mobile applications. We have been pleased by how easy it has been to adapt applications to use these Rover facilities to create 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 Mosaic that provides a useful service in the absence of network connectivity. In practice, we have found that the combination of the Rover cache, relocatable dynamic objects for interactive support, and queued remote procedure calls results in a surprisingly useful system.

The largest, most important, drawback of the Rover approach is that application designers must think carefully about how application functions should be divided between a client and a server. For example, to permit a disconnected client to interact with server facilities, the application architect must create appropriate relocatable dynamic objects and make sure they are loaded into the client's cache. If a disconnected client is permitted to make updates to server-based objects, application-specific consistency constraints must be implemented. Queued remote procedure calls may require user interface adaptations that correspond to the idea of delayed responses to requests. However, we have found that certain applications, including Mosaic and Netscape, can use queued remote procedure calls with the creative re-engineering of existing protocols.

It is possible that relocatable dynamic objects and queued remote procedure calls will find application in workstation environments as well as mobile environments. For example, relocatable dynamic objects can be used to off-load scripting tasks from Web servers, and queued remote procedure calls permit the background delivery of large objects, such as video segments.

Mobile computing has the potential to make a dramatic impact on people's lives, and we are encouraged by the wide range of applications that are possible in the absence of continuous, high-bandwidth connectivity. The next steps in our research program will be to further understand how to accommodate an increasing range of mobile applications that reflect the full capability of the mobile environment to the end user.

Acknowledgments

We thank the anonymous reviewers (including the Stanford students from Mary Baker's distributed systems course), Mitchell Charity, Greg Ganger, Wilson Hsieh, Kirk Johnson, Chanathip Namprempre, Peter Szilagyi, Bienvenido Vélez, Carl Waldspurger, Deborah Wallach, and Ron Weiss for their careful readings of earlier versions of this paper and insightful discussions. We would also like to thank Michael Shurpik and Sanjay Ghemawat for helping with the development of Rover and Webcal.

References

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

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

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

[4] T. Berners-Lee, R. Caillau, A. Luotonen, H. Frystyk, and A. Secret. The world-wide web. *Communications of the ACM*, 37(8):76–82, August 1994.

[5] T. Berners-Lee, R. T. Fielding, and H. Frystyk. *HyperText Transfer Protocol – HTTP/1.0.* IETF HTTP Working Group Draft 02, Best Current Practice, August 1995.

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

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

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

[9] D. H. Crocker. Standard for the format of ARPA internet text messages. *RFC 822*, Aug 1982.

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

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

[12] P. Deutsch and C.A. Grant. A flexible measurement tool for software systems. *Information Processing 71*, 1971.

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

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

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

[16] D. K. Gifford and J. E. Donahue. Coordinating independent atomic actions. In *Spring Compcon '85*, pages 92–92, San Francisco, CA, February 1985.

[17] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *CACM*, 31(3):288–298, March 1988.

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

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

[20] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc of the Eleventh Symposium on Operating Systems Principles (SOSP)*, December 1987.

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

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

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

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

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

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

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

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

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

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

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

[32] P. Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Proc. of the Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, pages 66–70, Napa, CA, 1993.

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

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

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

[36] B. Liskov, M. Day, and L. Shrira. Distributed object management in Thor. In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, 1993.

[37] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls. In *Proc. SIGPLAN 88 Conf. on Progr. Lang. Design and Impl.*, pages 260–267, Atlanta, GA, June 1988.

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

[39] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, CO, 1995.

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

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

[42] Netscape Communications Corporation. *Netscape Navigator*. Mountain View, CA, 1995.

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

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

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

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

[47] J. Rees and W. Clinger. The revised^3 report on the algorithmic language Scheme. AI Memo 848a, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, September 1986.

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

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

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

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

[52] K. Sollins and L. Masinter. *Functional Requirements for Uniform Resource Names*. Internet RFC1737, December 1994.

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

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

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

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

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

[58] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. of the Fourteenth Symposium on Operating Systems Principles (SOSP)*, pages 203–216, Asheville, NC, 1993.

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

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

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

[62] W. Weihl and B. Liskov. Implementation of Resilient, Atomic Data Types. *ACM Trans. Prog. Lang. Syst.*, 7(2):244–269, April 1985.

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