Mobile Computing with the Rover Toolkit

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

Abstract

Rover is a software toolkit that supports the construction of both *mobile-transparent* and *mobile-aware* applications. The objective of the mobile-transparent approach is to develop proxies for system services that hide the mobile characteristics of the environment from applications. Since applications can be run without alteration, the mobile-transparent approach is appealing. However, to excel, applications operating in the harsh conditions of a mobile environment must often be aware of and take an active part in mitigating those conditions. The Rover toolkit supports a set of programming and communication abstractions that enable the construction of both mobile-transparent and mobile-aware applications. Using the Rover abstractions, applications obtain increased availability, concurrency, resource allocation efficiency, fault tolerance, consistency, and adaptation. Experimental evaluation of a suite of mobile applications built with the toolkit demonstrates that such application-level control can be obtained with relatively little programming overhead and allows correct operation, increases interactive performance, and dramatically reduces network utilization under intermittently connected conditions.

I. Introduction

The mobile computing environment presents application designers with a unique set of communication and data integrity constraints that are absent in traditional distributed computing settings. For example, although mobile communication infrastructures are becoming more common, network bandwidth in mobile environments is often severely limited or unavailable. Mobile application designers therefore require system facilities that minimize dependence upon continuous network connectivity; provide tools to optimize the utilization of available network bandwidth; minimize dependence on data stored on remote servers; and allow for dynamic division of work between clients and servers. In this paper, we describe the *Rover toolkit*: a set of software tools that both supports applications that operate oblivious to the underlying environment, and enables the construction of applications that use awareness of the mobile environment to isolate themselves from its limitations. We illustrate the effectiveness of the toolkit using a number of distributed applications, each of which runs well over several networks that differ by three orders of magnitude in bandwidth and latency.

To appear in *IEEE Transactions on Computers: Special issue on Mobile Computing,* February 1997.

A. Mobile versus Stationary Environment

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

Computers in a stationary environment are usually very reliable. Relative to their stationary counterparts, mobile computers are quite fragile: a mobile computer may run out of battery power, be damaged in a fall, be lost, or be stolen. Given these threats, primary ownership of data should reside with stationary computers, not mobile computers. Furthermore, application designers should take special precautions to enhance the resilience of the data stored on mobile computers.

Relative to most stationary computers, a mobile computer has fewer computational resources available. However, the available resources may change dynamically (*e.g.,* a "docked" mobile computer that has access to a larger display, graphic or math coprocessor, additional stationary storage, etc.).

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

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

When users modify local copies of global data, consistency becomes an issue. In a mobile environment, optimistic concurrency control [1] is useful because pessimistic methods are inappropriate (a disconnected user cannot grab or release locks), as pointed out by the designers of Coda [2]. However, using an optimistic approach does not come for free: associated with long duration partitions, will be a greater incidence of update-update conflicts than in stationary environments. It is therefore important to use application-specific semantic information to detect when such conflicts are false positives and can be avoided.

B. The Argument for Mobile-Aware Computing

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

Such *mobile-transparent* applications can be used unmodified in mobile environments by having the system shield or hide the differences between the stationary and mobile environments from applications. Coda [2] and Little Work [3] used this approach by providing a file system interface to applications. The systems consist of a local *proxy* for some service (the file system) running on the mobile host and providing the standard service interface to the application, while attempting to mitigate any adverse effects of the mobile environment. The proxy on the mobile host cooperates with a remote server on a well-connected, stationary host.

However, the mobile-transparent approach sacrifices functionality and performance. While the system hides mobility issues from the application, it usually requires manual intervention by the user (*i.e.,* having the user indicate which data to prefetch onto the user's computer). Similarly, conflict resolution is complicated because the interface between the application and its data was designed for a stationary environment. Consider an application writing records into a file shared among stationary and mobile hosts. While disconnected, the application on the mobile host inserts a new record. The local file system proxy records the write in a log. Meanwhile, an application on a stationary host alters another record in the same file. Upon reconnection, the file system can detect that conflicting updates have occurred. However, the file system alone cannot resolve the conflict.

Coda recognizes this limitation and provides for the use of *application-specific resolvers* (ASRs) [4]. However, ASRs alone are insufficient. In the above example, there is no way for the ASR to use the file system interface to determine whether the mobile host inserted a new record or the stationary host deleted an old one. The cause of this confusion is that Coda changes the contract between the application and the file system in order to hide the condition of the underlying network. The read/write interface no longer applies to a single file, but to possibly inconsistent replicas of the file. Therefore, any applications that depend on the standard read/write interface for synchronization and ordering may fail.

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

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

Unlike previous systems, the Rover toolkit is designed to support both mobile-aware and mobile-transparent approaches. For mobile-aware applications, the Rover toolkit provides the components and architecture necessary for operation in a mobile environment. Our hypothesis is that running mobile applications efficiently and correctly often requires making applications and users aware of the environment in which they are running.

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

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

C. Rover: The Toolkit Approach

The Rover toolkit offers applications a distributed object system based on a client/server architecture [6] (see Figure 1). Clients are Rover applications that typically run on mobile hosts, but could run on stationary hosts as well. Servers, which may be replicated, typically run on stationary hosts and hold the long term state of the system. Communication between clients is limited to peer-to-peer interactions within a mobile host (using the local object cache for sharing) and mobile host–server interactions; there is no support for remote peer-to-peer or mobile host–mobile host interactions.

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

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

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

D. Main results

Earlier work on Rover introduced the Rover architecture, including both queued RPC and relocatable dynamic objects [6], [8]. This paper extends the design and implementation of QRPC and RDOs as described in [6] with compressed and batched QRPCs, presents the argument for making applications mobile-aware in greater depth, and explains how applications use that awareness and the Rover toolkit to mitigate the effects of intermittent communication on application performance. We draw four main conclusions from our experimental data and experience developing Rover:

1. QRPC meshes extremely well with intermittently connected environments. Queuing enables RPCs to be scheduled, batched, and compressed for increased network performance. QRPC performance is acceptable even if every RPC is stored in stable logs at clients and servers. For lower-bandwidth networks, the overhead of writing the logs is dwarfed by the underlying communication costs.
2. Use of RDOs allows mobile-aware applications to migrate functionality dynamically to either side of a slow network connection to minimize the amount of data transiting the network. Caching RDOs reduces latency and bandwidth consumption. Interface functionality can run at full speed on a mobile host, while large data manipulations may be performed on the well-connected server.
3. We have implemented several mobile-aware applications (Rover Exmh, Webcal, Irolo, Stock Market Watcher) and several proxies to support mobile-transparent applications (Web and USENET). Our experience indi-

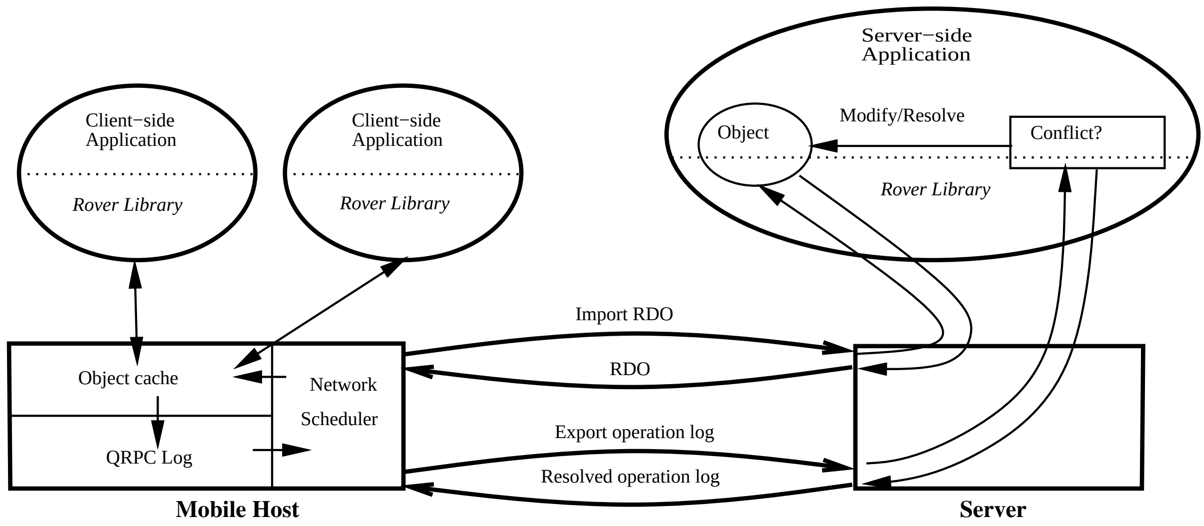

Fig. 1. Rover offers applications a client/server distributed object system with client caching and optimistic concurrency control. Rover applications employ a primary-copy, tentative-update optimistic-consistency-based distributed object model of data sharing: they call into the Rover library to *import* RDOs, and to *export* logs of operations that mutate RDOs. Client-side applications *invoke* operations directly on locally cached RDOs. Server-side applications are responsible for resolving conflicts and notifying clients of resolutions. A network scheduler drains the stable QRPC log, which contains the RPCs that must be performed at the server.

cates that porting applications to Rover generally requires relatively little change to the original application. Building Rover proxies is also easy and has allowed the use of applications (*e.g.,* Netscape and XRN) without modification. Most applications have been made mobile aware with only simple changes (approximately 10% of the original code and as little as three weeks work), while others required several person-months of work.

4. Measurements of end-to-end mobile-application performance show that, by using Rover, mobile-transparent and mobile-aware applications perform significantly better than their original versions. For example, for the mobile-transparent Netscape application, we observe performance improvements of up to 17%. For mobile-aware applications running over slow networks, we observe performance improvements of up to a factor of 7.5 over the original versions.

E. Outline of this paper

In the remainder of this paper we place our research in the context of related work (Section II), present the design of the Rover toolkit (Section III), describe the implementation of the Rover toolkit (Section IV), discuss the implementation of several mobile-transparent and mobile-aware applications (Section V), present experimental results from these applications (Section VI), and finally, offer observations on the benefits and limitations of the Rover approach (Section VII).

II. Related Work

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

Several previous projects have studied building mobile-transparent services for mobile clients. The Coda project pioneered distributed services for mobile clients. In particular, it investigated how to build a mobile-transparent file system proxy for mobile computers by using optimistic concurrency control and prefetching [2], [14]. Coda logs all updates to the file system during disconnection and replays the log on reconnection; automatic conflict resolution mechanisms are provided for directories and files, using Unix file naming semantics to invoke ASRs at the file system level [4]. A manual repair tool is provided for conflicts of either type that cannot be resolved automatically. A newer version of Coda supports low-bandwidth networks as well as intermittent communication [15].

The Ficus file system is also a mobile-transparent file system supporting disconnected operation, but relies on version vectors to detect conflicts [16]. The Little Work project caches files to smooth disconnection from an AFS file system [17]. Conflicts are detected and reported to the user. Little Work is also able to use low-bandwidth networks [18].

The BNU project implements an RPC-driven mobile-transparent application framework on mobile computers. It allows for function shipping by downloading Scheme functions for interpretation [19]. The BNU environment includes mobile-transparent proxies on stationary hosts for hiding the mobility of the system. BNU applications do not dynamically adjust to the environment; nor do they have a concept of tentative or stale data. No additional support for disconnected operation, such as Rover's queued RPC, is included in BNU. A follow-up project, Wit, addresses some of these shortcomings and shares many of the goals of Rover, but employs different solutions [20]. Application designers for BNU noted that the workload characterizing mobile platforms is different from workstation environments and will entail distinct approaches to user interfaces [21].

A number of proposals have been made for various degrees of mobile-awareness in operating system services and application. The Bayou project [12], [22] defines a mobile-aware database architecture for sharing data among mobile users. Bayou supports tentative operation logs and data values [23] and session guarantees for weakly-consistent replicated data [24]. To illustrate these concepts, the authors have built a calendar tool and a bibliographic database. Rover shares the notions of tentative operations and data, session guarantees, and the calendar tool example with the Bayou project. Rover extends this work with an application programming interface, RDOs, and QRPC to deal with intermittent communication, limited bandwidth, and resource-poor clients.

The InfoPad [25], Daedalus [26], GloMop [27] and W4 [28] projects focus on mobile-aware wireless information access. The InfoPad project employs a dumb terminal and offloads all functionality from the client to the server. Daedalus and GloMop use dynamic "transcoding" or "distillation" to reduce the bandwidth consumed by data transmitted to a mobile host. The transcoding technology is completely compatible with Rover's architecture. Applications on the mobile host cooperate with mobile-aware proxies on a stationary host to define the characteristics of the desired network connections. Similarly, W4 applies the technique of dividing application functionality between a small PDA and a powerful, stationary host to Web browsing. Rover is designed for more flexible, dynamic divisions. Depending on the power of the mobile host and available bandwidth, Rover allows mobile-aware browsers to dynamically move functionality between the client and the server.

The BARWAN [29] project supports mobile, "data type aware" applications. The approach relies on strongly typed transmissions. A dynamically extensible type system enables type-specific compression levels and abstraction mechanisms to conserver network usage. User code is itself a transmission type allowing computation relocation. Davies' Adaptive Services [10] similarly takes a protocol-centric approach for exposing information about the mobile environment to the application. A similar approach is taken by the Odyssey project. Odyssey focuses on system support to enable mobile-aware applications to use "data fidelity" to control resource utilization. Data fidelity is defined as the degree to which a copy of data matches the original. [30] Again, Rover is designed to focus on dynamic adaptation of program functionality and data types.

A number of successful commercial mobile-aware applications have been developed for mobile hosts and limited-bandwidth channels. For example, Qualcomm's Eudora is a mail browser that allows efficient remote access over low-bandwidth links. Lotus Notes [31] is a groupware application allowing users to share data in a weakly-connected environment. Notes supports two forms of update operations: append and time-stamped. Conflicts are referred to the user. TimeVision and Meeting Maker are group calendar tools allow a mobile user to download portions of a calendar for off-line use. The Rover toolkit and its applications provide functionality that is similar to these proprietary approaches, but in an application-independent manner. Using the Rover toolkit, standard workstation applications such as *Exmh* and *Ical* can easily be turned into mobile-aware applications.

Gray *et. al.* perform a thorough theoretical analysis of the options for database replication in a mobile environment and conclude that primary copy replication with tentative updates is the most appropriate approach for mobile environments [32].

III. Design of the Rover Toolkit

The Rover toolkit is designed to support the construction of mobile-aware applications and proxies. In this section we describe the key components of the Rover toolkit.

A. *Object Design and QRPC*

As the central structures about which all Rover design decisions revolve, relocatable dynamic objects (RDOs) provide the key point of control in Rover applications. All application code and all application-touched data are written as RDOs. RDOs may execute at either clients or servers. All RDOs have a "home" server that maintains the primary, canonical copy. Clients import secondary copies of RDOs into their local caches and export tentatively updated RDOs back to their home servers.

RDOs may vary in complexity from simple calendar items with a small set of operations to modules that encapsulate a significant part of an application (*e.g.,* the graphical user interface for an e-mail browser). Complex RDOs may create a thread of control when they are imported. The safe execution of RDOs is ensured by authentication and by execution of RDOs in a controlled environment. These safety measures are appropriate for the sharing of objects between mobile hosts and servers in the framework of specific applications. However, there are several safety issues relating to the general use of mobile code that are not addressed by our current implementation. This is an area of active research beyond the scope of this paper.

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

Rover clients use QRPC to lazily fetch RDOs from servers (see Figure 1). When an application issues a QRPC, Rover stores the QRPC in a local stable log and immediately returns control to the application. If the application has registered a *callback* routine, then when the requested RDO has arrived, Rover will invoke the callback to notify the application. Alternatively, applications may simply block to wait for critical data (although this is an undesirable action, especially when the mobile host is disconnected). When the mobile host is connected, the Rover network scheduler drains the log in the background, forwarding any queued QRPCs to the server.

When a Rover application modifies a locally cached RDO, the cached copy is marked *tentatively committed*. Updates are committed by using QRPC to lazily propagate the mutating operations to the Rover server, where they are applied to the canonical copies. In the meantime, the application may choose to use tentatively committed RDOs. This allows the application to continue execution even if the mobile host is disconnected.

B. Communication Scheduling

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

QRPC supports split-phase operation; thus, if a mobile host is disconnected between sending the request and receiving the reply, a Rover server will periodically attempt to contact the mobile host and deliver the reply. The split-phase communication model enables Rover to use different communication channels for the request and the response and to close channels during the intervening period. Several 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 waiting period is long and the client must pay for connection time.

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

C. Computation Relocation

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

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

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

D. Notification

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

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

E. Object Replication and Consistency

An essential component to accomplishing useful work while disconnected is having the necessary information locally available [33]. RDO replica caching is the chief technique available in Rover to achieve high availability, concurrency, and reliability. In this section, we discuss strategies for selecting objects to replicate and for reducing consistency-related-costs.

E.1 Replication

RDO replication is accomplished during periods of network connectivity by filling the mobile host's cache with useful RDOs. Applications should decide which objects to prefetch. We believe that the usability of applications 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 applications can provide prioritized prefetch lists based upon high-level user actions. For example, *Rover Exmh* automatically generates prefetch operations for the user's inbox folder, recently received messages, as well as folders the user visits or selects.

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

E.2 Consistency

When clients are allowed to perform concurrent updates on shared RDOs, most applications require consistency control. The Rover toolkit provides significant flexibility in the choice of mechanism, ranging from application-level locking to application-specific algorithms for resolving uncoordinated updates to a single RDO. Since no single scheme is appropriate for all applications, Rover leaves the selection of consistency scheme to the application. However, only a limited number of methodologies lend themselves naturally to mobile environments. Therefore, Rover provides substantial but not exclusive support for primary-copy, tentative-update optimistic consistency.

We expect many applications will continue to use a variety of approaches, including *ad hoc* approaches such as hand editing or requiring all data replicas to converge to the same values. Certain applications will be structured as a collection of independent atomic actions [34], where the importing action uses application-level locks, version vectors, or dependency-set checks to implement fully-serializable transactions within Rover method calls. Of course, pessimistic concurrency control may cause long blocking periods in the mobile environment.

Rover directly supports primary-copy, tentative-update optimistic consistency control. Since optimistic concurrency control schemes allow updates by any host on any local data, we expect this approach to be widely used. Therefore, we have built into the Rover library support for operation logging, rollback, and replay; log manipulation functions; and automatically maintained RDO consistency vectors. So far, all Rover applications built to date use primary-copy consistency control.

The server is responsible for maintaining the consistent view of the system. Update conflicts are detected and resolved by the server, and the results of reconciliation are always treated by clients as overriding the tentative state stored at the client. Thus, the client only needs to submit tentative operations to the server to reconcile the system state and to assure that any updates are durable.

Rover automatically logs method invocations, rather than only new data values, to increase flexibility in resolving conflicts. For example, a financial account object with debit, credit, and balance methods provides a great deal more semantic information to the application than a simple account file containing only the balance. Debit and credit operations from multiple clients could be arbitrarily interleaved as long as the balance never becomes negative. In contrast, consistently updating a balance value by overwriting the old value would require use of an exclusive lock on the global balance.

When the QRPC for a mutating operation arrives at a server, the server invokes the requested method on the primary copy. Typically a method call first checks whether the RDO has changed since it was imported by a mobile host. The definition of conflicting modifications is strongly application- and data-specific. Therefore, Rover does not try to detect conflicts directly, although it maintains version vectors for each RDO to aid conflict detection.

In the event of an update-update conflict, the conflict must be resolved. Since the submitted operation is tentative and was originally performed at the client on tentative data, the result of performing the operation at the server may not be exactly what the client expected. However, since Rover can employ type-specific concurrency control [35], many conflicts can be avoided. Note that conflict detection may depend not only on the application, but on the data or even the operation involved.

IV. Implementation of the Rover Toolkit

As shown in Figure 2, the Rover toolkit consists of four key components: the access manager, the object cache (client-side only), the operation log, and the network scheduler; we discuss each component in turn.

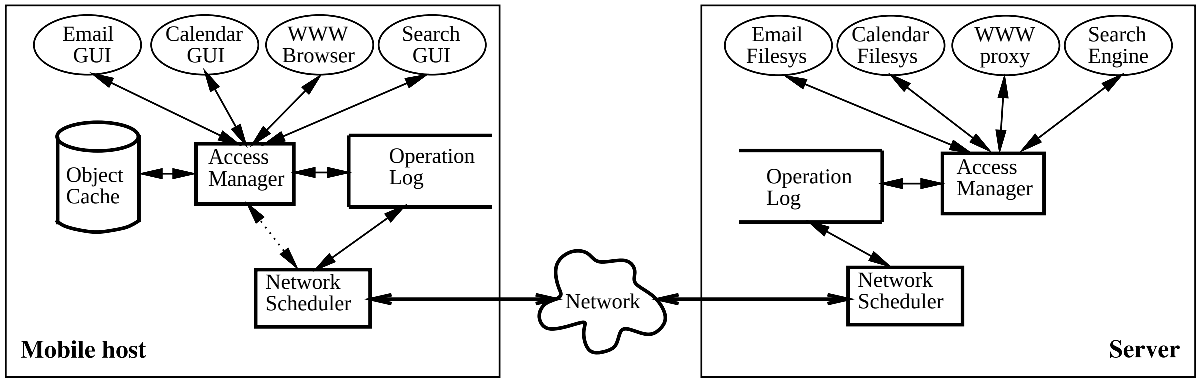

Fig. 2. The Rover architecture consists of four key components at clients and servers: access manager, object cache (client-side only), operation log, and network scheduler.

Each machine has a local Rover *access manager*, which is responsible for handling all interactions between client-side and server-side applications and among client-side applications. The access manager services requests for objects (RDOs), mediates network access, logs modifications to objects, and, at clients, manages the object cache. Client-side applications communicate with the access manager to import objects from servers and cache them locally. Server-side applications are invoked by the access manager to handle requests from client-side applications. Applications invoke the methods provided by the objects and, using the access manager, make changes globally visible by exporting them back to the servers.

Within the access manager, RDOs[1] 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.

Rover starts as a minimal "kernel" that imports functionality on demand. This feature is particularly important for mobile hosts with limited resources: small memory or small screen versions of applications may be loaded by default. However, if the application finds more hardware and network resources available (*e.g.,* if the mobile host is docked) further RDOs may be loaded to handle these cases [11].

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 re-queues any incomplete QRPCs for re-delivery. At-most-once delivery semantics are provided by unique identifiers and the persistent log. One issue that remains an open question is how to handle error responses from resent QRPCs for client-side applications that no longer are running. Our implementation currently ignores such responses.

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

[1] The current implementation of RDOs uses the Tcl and Tk languages [36]. However, since the interface is designed to be language-independent, it will be easy to explore the use of other interpreted or byte-compiled languages (*e.g.,* Java [37]).

Once an object has been imported into the client-side application's local address space, method invocations without side effects are serviced locally by the object. At the application's discretion, method invocations with side effects may also be processed locally, inserting tentative data into the object cache. Operations with side effects also insert a QRPC into a stable *operation log* located at the client. Each insert is a synchronous action.

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

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

One issue Rover addresses with an application-specific approach is operation log growth during disconnected operation. 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-ahead log. The traditional approach is log compaction [33]. Rover takes a different approach by directly involving applications in log compaction. Applications can download procedures into the access manager to manipulate their log records. For example, an application can filter out duplicate requests (*e.g.,* duplicate QRPCs to verify that an object is up-to-date can be reduced to a single QRPC). In addition, applications can apply their own notion of "overwriting" to the operations in the log.

The *network scheduler* groups operations destined for the same server for transmission and selects the appropriate transport protocol and medium over which to send them. Rover is capable of using a variety of network transports. Rover supports both connection-based protocols (*e.g.,* HTTP over TCP/IP networks) and connectionless protocols (*e.g.,* SMTP over IP or non-IP networks) [38], [39]. Different protocols have different strengths. For example, while SMTP has extremely high latency, it is fundamentally a queued background process; it is more appropriate than more interactive protocols for fetching extremely large documents, such as stored video, which require large amounts of time regardless of the protocol. Another advantage is that the IP networks required for HTTP or TCP are not always available, whereas SMTP often reaches even the most obscure locations.

The network scheduler leverages the queuing of QRPCs performed by the log to gain transmission efficiency. The result is a potentially significant reduction in per-operation transmission overhead and an increase in connection efficiency through amortization of connection setup and teardown across multiple requests and responses. This amortization is especially important when connection setup is expensive (either in terms of added latency or dollar cost). For example, the latency for a null RPC over a 9.6 Kbit/s Cellular CSLIP link is 2.23 seconds; batching offers a substantial performance benefit.

Our original network scheduler sent a request as soon as it was received from a client application. The new extended scheduler uses the following heuristic to batch requests that are destined for the same server: when a request is received from one client application, the scheduler uses the access manager to check all the client applications (including the one that sent the original request) to see if any are in the process of sending a request (this is the dashed arrow in Figure 2). If there are additional requests pending, the scheduler delays sending the original request. Upon receipt of the next request, the scheduler repeats the pending request check. When there are no pending requests from client applications, the scheduler batches the requests and sends them on the same connection; the results are also received on the same connection.

This heuristic imposes a small delay on requests (the time for the access manager to check each client application for pending requests and to receive the requests). However, this is a small penalty to pay relative that incurred by a high roundtrip time, since it allows the scheduler to automatically batch requests. Thus, an application that issues several requests in a series will have the requests automatically batched and sent to the server using a single connection. We are also investigating alternatives that rely upon applications to specify the set of requests that should be batched together.

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

V. Mobile Computing Using Rover

In this section, we discuss the steps involved with implementing mobile-aware applications (or porting existing applications to a mobile-aware environment), the programming interface provided by the Rover toolkit, and the set of sample applications that we constructed using the toolkit.

A. Using Objects Instead of Files

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

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

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

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

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

The next step is to add support for interacting with the environment. For example, in the e-mail example, one of the important pieces of message metadata that a folder object contains is the message's size and the size of any attachments. This information can be used by the application and conveyed to the user to allow useful decisions to be made. Support for prefetching is another environment interaction issue. Also, the application developer must decide which mechanisms to use for notifying users of the status of displayed data.

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

B. Toolkit Programming Interface

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

To import an object, an application calls *import* and provides the object's unique identifier, the session identifier, a callback, and arguments. In addition, the application specifies a priority that is used by the network scheduler to reorder QRPCs. The *import* function immediately returns a promise [40] to the application. The application can then wait on this promise or continue execution. Rover transparently queues QRPCs for each import operation in the stable log. When the requested object is received by the access manager, the access manager updates the promise with the returned information. In addition, if a callback was specified, the access manager invokes it.

The current implementation also has a *load* operation that is an *import* combined with a call to create a process. Applications use the *load* operation to import RDOs that need a separate thread of control. When the access manager receives an RDO that was requested by a *load*, it creates a separate process and executes the RDO. The reason for a separate *load* operation is historical. At the time that the prototype was implemented, the underlying target operating systems (the UNIX-based Linux and SunOS operating systems) did not support multiple threads per address space and only provided limited support for dynamic linking. In a future implementation, *load* may be directly incorporated within *import*.

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

TABLE I

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

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

C. Rover Application Suite

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

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

This application suite was chosen to test several hypotheses about the ability to reasonably meet users' expectations in a mobile, intermittently-connected environment. These applications represent a set of applications that mobile users are likely to use. Because RDOs affect the structure of applications, it is important to qualitatively test the ideas contained in the Rover toolkit with complete applications in addition to using standard quantitative techniques.

As can been seen in Table II, porting these file system-based workstation applications to a mobile-aware Rover applications requires varying amounts of work. Some applications were written/ported in a few weeks, while others

Rover Program	Base code	New Rover client code	New Rover server code
Rover Exmh	24,000 Tcl/Tk	1,700 Tcl/Tk 220 C	140 Tcl/Tk 2,700 C
Webcal	26,000 C++ and Tcl/Tk	2,600 C++ and Tcl/Tk	1,300 C++ and Tcl/Tk
Rover HTTP Proxy	none	250 Tcl/Tk 1,500 C	740 C
Rover Irolo	470 Tcl/Tk	400 Tcl/Tk 220 C	280 Tcl/Tk
Rover NNTP Proxy	none	525 Tcl/Tk 476 C	350 C
Rover Stock Watcher	none	84 Tcl/Tk 220 C	260 Perl 65 Tcl/Tk

TABLE II

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

required several person-months of work. For example, porting *Exmh* and *Ical* to Rover required simple changes to approximately 10% of the lines of code. Most of these changes came from replacing file system calls with object invocations; these modifications in *Rover Exmh* and *Rover Webcal* were made almost independently of the rest of the code.

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

C.1 Mobile-Transparent Applications

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

Rover HTTP proxy. This is a unique application that interoperates with most of the popular Web browsers. It allows users of existing Web browsers to "click ahead" of the arrived data by requesting multiple new documents before earlier requests have been satisfied. The proxy intercepts all web requests and, if the requested item is not locally cached, returns a null response to the browser and enqueues the request in the operation log. When a connection becomes available, the page is automatically requested. In the meantime, the user can continue to browse already available pages and issue additional requests for pages without waiting. The granularity of RDOs is individual pages and images.

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

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

The proxy can also directly control NCSA's *Mosaic* [41] and NCC's *Netscape Navigator* [42] browsers using their remote control interfaces.

C.2 Mobile-Aware Applications

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

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

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

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

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

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

Rover Stock Market Watcher. This application uses both computation migration and fault-tolerance techniques [8]. The client constructs RDOs for stocks that are to be monitored and sends them to the server. The server uses fault-tolerant techniques to store the real-time information retrieved from stock ticker services.

VI. Experiments

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

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

Network options that we have experimented with include 10 Mbit/s switched Ethernet, 2 Mbit/s wireless AT&T WaveLAN, 128 Kbit/s and 64 Kbit/s Integrated Digital Services Network (ISDN) links, and Serial Line IP with Van Jacobson TCP/IP header compression (CSLIP) [43] over 19.2 Kbit/s *V.32terbo* wired and 9.6 Kbit/s *Enhanced Throughput Cellular* (ETC) cellular dial-up links[2].

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

To minimize the effects of unrelated network traffic on the experiments, the switched Ethernet was configured such that the server, the ThinkPad Ethernet adapter, and the WaveLAN base station were the only machines on the Ethernet segment and were all on the same switch port. However, network traffic over the wired, cellular, and ISDN links used shared public resources and traversed shared links; thus, there is increased variability in the experimental results for those network transports. To reduce the effects of the variations on the experiments, each experiment was executed multiple times and the results averaged. It is important to note that ordinary TCP/IP was used on the wireless networks. While Rover applications might benefit from the use of a specialized TCP/IP implementation, it is not necessary; this is an advantage of using Rover. Since a Rover application sends less data than an unmodified application, it is less sensitive to errors on wireless links.

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

1. Using QRPC instead of RPC significantly improves performance by enabling batching and compression of multiple requests and responses.
2. Mobile-transparent applications benefit from using the Rover toolkit.
3. Mobile-aware applications offer significant performance advantages over existing versions.

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

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

TABLE III

The Rover experimental environment. Latencies are in milliseconds, throughput is in Mbit/s. Null RPC latency is a ping-pong over TCP sockets; TCP throughput is the time to send 1 Mbyte of compressible ASCII data (14.4:1 using GNU's *gzip -6*) similar to Rover Tcl-based RDOs; and QRPC latency is the time to perform a null QRPC. The ISDN and Wired and Cellular CSLIP links perform hardware compression. Note that the cellular times reflect the overhead of the ETC protocol and a non-error-free wireless link.

A. Null QRPC Performance

To establish the baseline performance for QRPC, we repeated the latency and bandwidth measurement experiments from [6], but extended them to include several additional network technologies and the use of Flash RAM for stable storage. The results are summarized in Table III.

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

To understand the effects of batching and compression, we measured the performance of QRPC with asynchronous logging. Figure 3 shows the effects of batching and compression (using the heuristic from Section IV) on the per-request cost when performing a series of 60 QRPCs. In each set of bars, the leftmost bar (compressed batched) shows the performance when both compression and batching are applied. For this test, the compression ratio was approximately twelve to one and the batch size was an average of seven requests per message. The second bar (compressed single) shows the performance with compression and only a single request outstanding. The compression ratio was 2.5 to one. The third bar (uncompressed overlapped) shows the performance without compression or batching, but with multiple outstanding requests. The rightmost bar (uncompressed single)shows the performance without compression or batching and with only a single request outstanding.

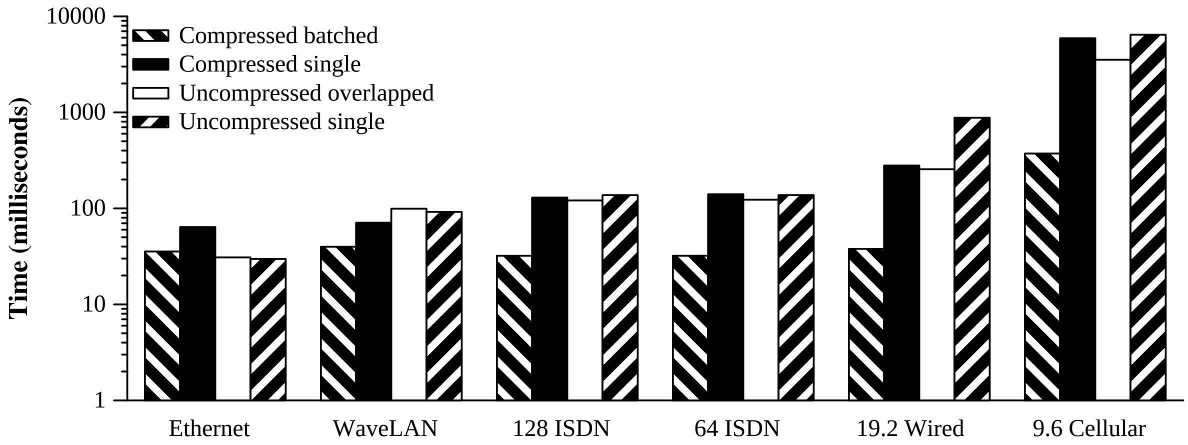

Fig. 3. Average time in milliseconds for one null QRPC when using compressing and batching a series of 60 QRPCs with asynchronous log record flushing. The y-axis uses a logarithmic scale.

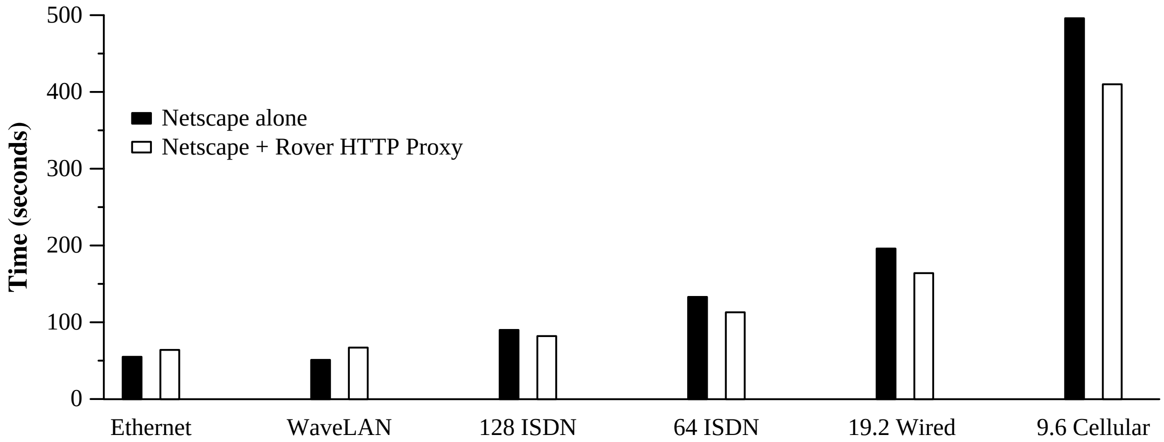

Fig. 4. Time in seconds to fetch/display 10 WWW pages using Netscape alone and with the Rover HTTP proxy.

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

B. Mobile-Transparent Application Performance

We compared the performance of Netscape, using a mobile-transparent Rover HTTP proxy against the same application executing independently. We measured the time to fetch and display 10 WWW pages using a variety of networks. Figure 4 provides the results of the experiment and shows that performance when using the Rover HTTP proxy is comparable for faster networks and up to 17% faster for the slower networks. The total data transmitted to the client was 239 Kbytes of compressed data representing 286 Kbytes of uncompressed data. The HTML portion of the pages accounted for 44.5 Kbytes and had a compression ratio of 2.6:1. The majority of the data consisted of images, which were far less compressible using the default compression. We plan to explore the use of application-specific image compression [27]. It is important to note that the experiments do not reflect the

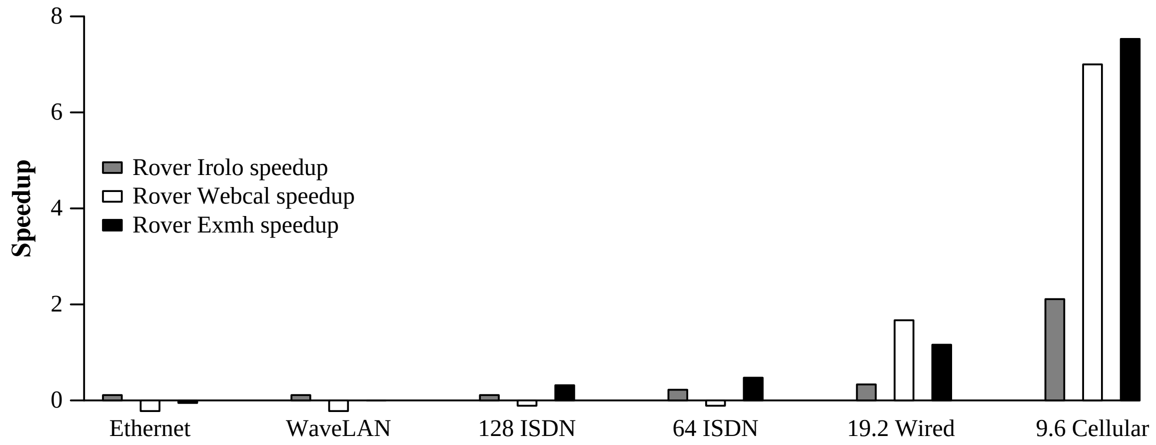

Fig. 5. Speedup (or slowdown) of Rover mobile-aware versions of applications over the original X11-based applications. The tasks were: reading eight MIME e-mail messages, viewing one week's appointments from a medium-sized calendar, and browsing fifty rolodex entries.

"click-ahead" nature of the Netscape+Rover HTTP proxy application, which allows the user to browse the loaded pages while waiting for additional pages to load.

C. Mobile-Aware Application Performance

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

To measure the performance benefits of the complete Rover system for mobile-aware applications, we compare the performance of Rover Webcal, Rover Exmh, and Rover Irolo against their unmodified X11-based counterparts, Ical, Exmh, and Irolo. For each application, we designed a workload representative of a typical user's actions and measured the time to perform the complete task. To keep the measurements representative, we did not measure the cost of starting the application and loading the data required for the task. This is typical of how the system is used, where the application is started and the data are loaded over a fast network and then the application is used repeatedly over a slow network (or without any network connectivity). Each task was repeated on each of the six network options.

Figure 5 presents the speedup (or slowdown) of the Rover version of each application over the original X11-based application. In general, the results show that, for fast networks (Ethernet, WaveLAN, and ISDN), the performance when using Rover is comparable (a slight speedup for Irolo, equal for Exmh, and a slight slowdown for Ical). Over slower networks (wired and cellular dial-up links), Rover application performance is consistently better (ranging from a 33% performance gain on wired dial-up to a factor of 7.5 on cellular dial-up). The results for these two networking technologies are especially encouraging, since they represent the target environment for Rover.

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

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

VII. Conclusions

We have shown that the integration of relocatable dynamic objects and queued remote procedure calls in the Rover toolkit provides a powerful basis for building mobile-transparent and mobile-aware applications. We have found it quite easy to adapt applications to use these Rover facilities, resulting in applications that are far less dependent on high-performance communication connectivity. For example, one might conjecture that it would be difficult to build a mobile version of Netscape that provides a useful service in the absence of network connectivity. In practice, we find the combination of the Rover cache, relocatable dynamic objects for interactive support, and queued remote procedure calls results in a surprisingly useful system.

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

In addition, measurements of end-to-end mobile application performance shows that mobile-transparent and mobile-aware applications perform significantly better than their stationary counterparts. For example, for the mobile-transparent Netscape application, we observe a performance improvement of 17%. For mobile-aware applications, we observe performance improvements of up to a factor of 7.5 over slow networks.

VIII. Acknowledgments

We thank David Gifford for his efforts in the early design stages of the Rover toolkit, in particular his idea of QRPC. We also thank Greg Ganger, Eddie Kohler, and the anonymous reviewers for their careful readings of earlier versions of this paper. We would also like to thank the rest of the Rover project team: George Candea, Constantine Cristakos, Alan F. deLespinasse, and Michael Shurpik for helping with the development of Rover.

References

[1] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, vol. 6, no. 2, pp. 213–226, June 1981.

[2] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 3–25, Feb. 1992.

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

[4] P. Kumar, *Mitigating the Effects of Optimistic Replication in a Distributed File System*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Dec. 1994.

[5] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–28, Nov. 1984.

[6] A. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and F. Kaashoek, "Rover: A toolkit for mobile information access," in *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, CO, Dec. 1995, pp. 156–171.

[7] A.D. Birrell and B.J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, Feb. 1984.

[8] A. Joseph and F. Kaashoek, "Building reliable mobile-aware applications using the rover toolkit," in *Proc. of the Second International Conference on Mobile Computing and Networking (MOBICOM '96)*, Rye, NY, Nov. 1996, pp. 117–129.

[9] R. H. Katz, "Adaptation and mobility in wireless information systems," *IEEE Personal Communications*, vol. 1, pp. 6–17, 1994.

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

[11] F. Kaashoek, T. Pinckney, and J. A. Tauber, "Dynamic documents: Mobile wireless access to the WWW," in *Proc. of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994, pp. 179–184.

[12] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch, "The Bayou architecture: Support for data sharing among mobile users," in *Proc. of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994, pp. 2–7.

[13] M.G. Baker, "Changing communication environments in MosquitoNet," in *Proc. of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994, pp. 64–68.

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

[15] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan, "Exploiting weak connectivity for mobile file access," in *Proc. of the Fifteenth Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, CO, Dec. 1995, pp. 143–155.

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

[17] L. B. Huston and P. Honeyman, "Disconnected operation for AFS," in *Proc. of the First USENIX Symposium on Mobile & Location-Independent Computing*, Cambridge, MA, Aug. 1993, pp. 1–10.

[18] L. Huston and P. Honeyman, "Partially connected operation," in *Proc. of the Second USENIX Symposium on Mobile & Location-Independent Computing*, Ann Arbor, MI, Apr. 1995, pp. 91–97.

[19] T. Watson and B. Bershad, "Local area mobile computing on stock hardware and mostly stock software," in *Proc. of the First USENIX Symposium on Mobile & Location-Independent Computing*, Cambridge, MA, Aug. 1993, pp. 109–116.

[20] T. Watson, "Application design for wireless computing," in *Proc. of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994, pp. 91–94.

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

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

[23] 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*, Santa Cruz, CA, Dec. 1994, pp. 192–195.

[24] 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*, Sept. 1994, pp. 140–149.

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

[26] S. Narayanaswamy, *et. al.*, "Application and network support for InfoPad," *IEEE Personal Communications*, vol. 3, no. 2, pp. 4–17, Apr. 1996.

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

[28] J. Bartlett, "W4—the Wireless World-Wide Web," in *Proc. of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994, pp. 176–178.

[29] R. Katz *et. al.*, "The Bay Area Research Wireless Access Network (BARWAN)," in *Proc. of the Spring COMPCON Conference*, Feb. 1996.

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

[31] 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, Sept. 1988.

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

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

[34] D. K. Gifford and J. E. Donahue, "Coordinating independent atomic actions," in *Proc. of the Spring COMPCON Conference*, San Francisco, CA, Feb. 1985, pp. 92–92.

[35] W. Weihl and B. Liskov, "Implementation of Resilient, Atomic Data Types," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, pp. 244–269, Apr. 1985.

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

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

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

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

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

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