Building Reliable Mobile-Aware Applications using the Rover Toolkit

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

DRAFT October 2, 1996 **DRAFT**

Abstract

This paper discusses extensions to the Rover toolkit for constructing reliable mobile-aware applications. The extensions improve upon the existing failure model, which addressed client or communication failures and guaranteed reliable message delivery from clients to server, but did not address server failures (*e.g.,* the loss of an incoming message due to server failure) [16]. Due to the unpredictable, intermittent communication connectivity typically found in mobile client environments, it is inappropriate to make clients responsible for guaranteeing request completion at servers. The extensions discussed in this paper provide both system- and language-level support for reliable operation in the form of stable logging of each message received by a server, per-application stable variables, programmer-supplied failure recovery procedures, server process failure detection, and automatic server process restart. The design and implementation of fault-tolerance support is optimized for high performance in the normal case (network connectivity provided by a high latency, low bandwidth, wireless link): measurements show a best-case overhead of less than 7% for a reliable null RPC over wired and cellular dialup links. Experimental results from both micro-benchmarks and applications, such as the Rover Web Browser proxy, show that support for reliable operation can be provided at an overhead of only a few percent of execution time during normal operation.

1 Introduction

The Rover toolkit provides mobile application developers with a client/server distributed object model that isolates mobile applications from the limitations of mobile communication systems [16]. This paper discusses extensions to the Rover toolkit for the construction of *reliable* mobile-aware applications. These extensions improve upon the existing Rover failure model, which addressed client or communication failures and guaranteed reliable message delivery from clients to server, but did not address server failures. The extensions described provide the tools for handling a specific class of faults: transient, recoverable faults. These faults are typically caused by environmental circumstances (*e.g.,* power glitches, communication link errors or failures, resource exhaustion due to high system load, etc.) or software errors in rarely used code paths. The extensions do not address repeatable or non-recoverable failures (*e.g.,* those due to critical design or implementation errors). Although transient and recoverable failures are a restricted class of failures, they represent a significant fraction of actual system and application failures [11]. Furthermore, in a mobile environment, the incidence of communications faults (*e.g.,* dropped phone calls, lost carriers, etc.) will be significantly higher than for fixed workstations connected by wired networks.

The reliabilty extensions leverage functionality already provided by the Rover toolkit: stable logging of each message sent by a client and message retransmission after communication failures. While the use of

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, by an Intel Graduate Fellowship, and by grants from AT&T, IBM, and Intel.

stable logging at the client provides reliable delivery of a message to a server, it does not handle failures at the server. This limitation can be particularly problematic for long-running operations at servers. To address this deficiency, the extensions add stable logging of each message received by a server, per-application stable variables, programmer-supplied failure recovery procedures, server process failure detection, and automatic server process restart. This additional functionality provides the necessary building blocks for constructing efficient, reliable, mobile-aware applications.

Logging messages at both clients and servers might appear to be an inappropriate and expensive solution. An alternative solution, where the client is responsible for guaranteeing both delivery of messages and completion of operations at servers, could easily be implemented using retransmission timers. This solution would be appropriate for a low latency, high bandwidth wired environment. However, there are several reasons why it is completely inappropriate for a mobile environment, including higher latency, lower bandwidth, intermittent connectivity or periods of extended disconnection, and retransmission cost. For these reasons, it is more appropriate to add logging at both ends of the link.

Our approach is not based upon transactions; it is an application-involved approach. In keeping with the design philosophy of the Rover toolkit, application designers are allowed to build their own transactional models.

The techniques used by the extensions are not novel; for example, other systems using similar approaches are Clouds [1], Tacoma [15], and Argus [17] (for more details see Section 6). However, the techniques are applied in a unique way to the problems of mobile computing, specifically the movement of computation from a client to a server in an environment in which intermittent connectivity and limited bandwidth are the norm and not the exception.

Performance is one of our primary concerns. The extensions to the Rover toolkit are designed so that the performance of applications (especially those not using fault-tolerant features) is only minimally affected during normal operation. The experimental results presented in Section 5 show that application performance is acceptable (an overhead of only a few percent for cellular links) even when providing very reliable operation by stably logging client and server messages.

Several Rover-based applications demonstrate the benefits of supporting fault-tolerant operation: the Rover Web Browser proxy, a simple file search utility, and a financial stock tracking application. For example, in the case of the file search application, Rover uses stable variables to store information about the state of the search (*i.e.,* the remaining files and directories to be searched and the matches found so far). Thus, after a failure, the search resumes with only a small amount of lost work.

Our experience with applications developed using the fault-tolerant extensions to the Rover toolkit shows that it is easy to build applications that tolerate transient faults. We also observe that, due to the overhead associated with stable variables, there should be a clear differentiation between stable and volatile variables.

These results are directly applicable to other mobile application environments. The tools and techniques provided by the Rover toolkit could be applied towards building reliable agents, fault-tolerant Java applets [2], or other reliable mobile code environments.

In the remainder of this paper we present a brief synopsis of the original Rover toolkit and its failure model (Section 2), describe the extensions to the toolkit to provide the fault-tolerant programming model (Section 3), present the Rover implementation (Section 4), review experimental results from the sample applications (Section 5), discuss related work (Section 6), and conclude with observations on the benefits and limitations of the extensions (Section 7).

2 Rover Toolkit

This section presents a brief overview of the design and implementation of the original Rover toolkit and its failure model.

2.1 Rover Overview

The Rover toolkit [16] offers applications a distributed object system based on a client/server architecture. Client applications typically run on mobile hosts, but could run on stationary hosts as well. Server applications typically run on stationary hosts and hold the long-term state of the system. The toolkit is designed

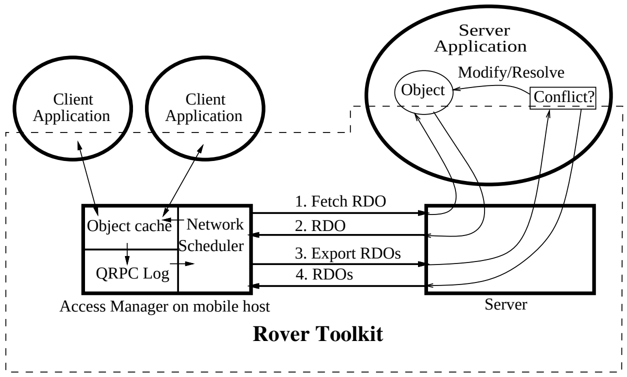

Figure 1: The Rover toolkit client/server distributed object model. Rover offers applications client caching and optimistic concurrency control based upon a check-in, check-out model of data sharing. Client applications use QRPCs to import RDOs from servers (steps 1 and 2) and to export changed RDOs back to servers (steps 3 and 4).

to simplify the construction of applications that allow useful operation in the presence of limited or nonexistent communication bandwidth and with varying computational resources. Support for this flexibility is based upon 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 loaded dynamically 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 [5] even when a host is disconnected: requests and responses are exchanged upon network reconnection. In the meantime, they are stored in a stable client log. QRPC supports split-phase communications: different connections or communications transports may be used for sending and receiving responses.

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 graphical user interface of a calendar tool). These more complex RDOs may create threads of control when they are imported.

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 with 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, options that are not possible during disconnected operation or are potentially expensive when intermittently connected.

Figure 1 shows the model of computation used by the Rover toolkit. RDOs at mobile hosts are stored in a shared cache managed by a local Rover Access Manager (AM). Rover lazily fetches RDOs from servers using QRPC. The QRPCs are stored in a stable log that is drained by the network scheduler whenever the mobile host is connected.

In response to the arrival of a fetch request, the Rover server sends the requested RDO to the mobile host. Upon receiving a reply, Rover inserts the RDO returned into the cache, deletes the QRPC from the stable client log, and notifies the requesting clients. If a mobile host becomes disconnected between sending the request and receiving the reply, it will retry the original operation. The server is responsible for detecting and handling duplicate requests. This may involve repeating the work performed by the original operation.

Updates made by applications to RDOs proceed in 2 stages. The mutating operations are first lazily propagated to the Rover server using QRPC and then applied to the canonical copy of the RDOs.

When the QRPC for a mutating operation arrives at a server, the server passes it to the appropriate application. The application uses the RDO's version vector (maintained by the toolkit) to check whether the RDO has changed since it was imported by a mobile host. Because Rover can employ type-specific concurrency control [24], many conflicts can be resolved automatically. Rover provides the mechanisms for detecting conflicts and leaves it up to applications to reconcile them [21]. Any results are sent back to the client.

2.2 Rover Fault Model

The fault model for the Rover toolkit addresses transient client software/hardware and communication link failures. The model balances the need to hide hardware and software faults with the need to avoid lowering performance in the normal case.

Hardware faults occur mostly in the form of communications link failures (*e.g.,* a dropped dialup link or a packet dropped by a congested router). Additional hardware faults are power losses, failed components (in particular, hard disk drives), and total loss situations (theft, fire, etc.). The Rover toolkit only addresses transient or recoverable failures. For most applications, providing a system that can tolerate hard failures would impose an unnecessary burden on programmers and performance.

Software faults occur mostly in the form of transient failures or *Heisenbugs*. These faults are usually benign failures caused by programming errors in rarely executed code paths (*e.g.,* race conditions), resource exhaustion, or transient hardware errors. The faults can usually be cured by reexecuting the failed operation. Hard failures or *Bohrbugs* are faults that are not transient; they recur when the operation is reexecuted (*e.g.,* due to programming error). In addition, they may corrupt system state, leaving the system unrecoverable even after a restart. As with permanent hardware faults, there are many research techniques that can be applied to hard faults [3], but they do not concern us here. Past studies have shown [11] that most hardware and software errors are transient, recoverable failures. The best recovery mechanism for these failures is to restart the application or system.

2.3 Existing Rover Support for Reliable Applications

The existing Rover toolkit handles only client software/hardware failures and failures of the communication link while sending QRPCs. For example, QRPCs received at servers and the responses sent by those servers are not stored in a stable log. Thus, while it is an infrequent occurrence, a server software or hardware failure is an unhandled weak point.

The toolkit uses a a separate process model and a stable outgoing QRPC log to guarantee the delivery of an operation from a client application to a server in the presence of transient communication link, client, or Access Manager failures.

2.3.1 Client application failures

Rover clients use a local client-server model: each Rover application executes in a separate address space and communicates via Inter-Process Communication (IPC) with the local Rover Access Manager. By using a separate address space, the failure of an individual client has no effect on other clients or the Access Manager.

To improve the efficiency of the client-server model, 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 design of the Rover toolkit only addresses access manager failures and not failures of client applications. The toolkit does not currently provide fault-tolerant support for client applications (*i.e.,* a client application that fails will not be automatically restarted and all application state, except for that stored in exported RDOs, is lost).

2.3.2 Client hardware and software failures

The Rover toolkit includes limited support for transient hardware and software failures of clients.

Before a QRPC is sent to a server, a record containing the QRPC is appended to a stable client log. When a response to a QRPC is received from a server, another record is appended to the stable client log, indicating that the QRPC is completed.

The stable client log is used to guarantee at-least-once delivery of QRPCs to servers in the presence of transient software or hardware failures. The stable client log is implemented as an ordinary UNIX file located on a hard disk, on a battery-backed Static RAM PCMCIA card, or Flash RAM PCMCIA card; media failures for these devices are sufficiently rare that we ignore them for this prototype. When an action is entered into

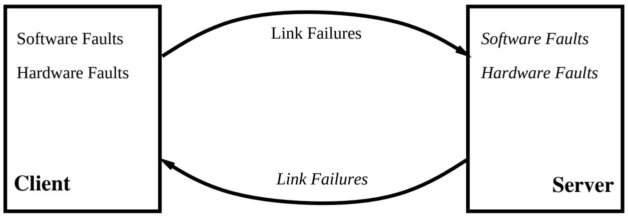

Figure 2: Extended Rover fault model. The faults listed in *italics* are the ones that are addressed by the extensions to the Rover toolkit discussed in this paper.

the log, the Access Manager performs both a flush and (optionally) a file synchronize operation to force the action to the log[1]. Thus, a flush is on the critical path for any logged action.

The Access Manager attempts to resend QRPCs in response to delivery failures (due to a communications problem, link failure, or server failure). The Access Manager first reevaluates the condition of available communications links. If no link is available, it leaves the QRPC in the to-be-sent queue. Otherwise, the Access Manager marks the QRPC as a retry and immediately attempts to resend the QRPC. If this second attempt fails, the server is marked as down. The Access Manager will then periodically reevaluate the server's condition and attempt to resend the QRPC.

During client failure recovery, the Access Manager scans the stable log, looking for any pending or incomplete QRPCs. It marks each pending QRPC as a retry and resends it to the server using the same process described above.

3 Design Extensions for Reliable Mobile Applications

In this section, we discuss the system design of the three extensions added to the Rover toolkit to provide reliable execution of RDOs at servers: stable server logging of incoming QRPCs, stable server logging of outgoing responses, and programming language support for reliable operation.

With the extensions, Rover guarantees that a mutating operation performed by a client application will be delivered to and executed at a server and that the results will be delivered back to the client application in the presence of transient communication link, server, and client Access Manager failures.

The faults addressed by both the original and extended Rover fault model are listed in Figure 2. We retain the same overhead/performance balancing goal as the original toolkit. The changes (shown in Figure 3) consist of three extensions:

1. *Server logging of incoming QRPCs to a stable server log.* This addresses the previously unhandled problem of a server crashing after acknowledging successful receipt of a message.

2. *Server logging of intermediate server application state to a stable server log and programmer-supplied application recovery procedures.* This reduces the amount of work that is repeated after a server failure and allows applications to be directly involved in the recovery process.

3. *Server logging of responses to a stable server log before transmission.* This reduces the amount of work that is repeated after a server failure.

3.1 Server logging of incoming QRPCs

When the server receives a QRPC from a client, it immediately appends *message reception* record containing the QRPC to its stable log. This is important because the order in which QRPCs will be executed may not match the order in which they were received (*e.g.,* the messages containing the QRPCs may have been arbitrarily reordered by the communications channel). If the server receives a QRPC out of order, it saves

[1]To increase the robustness of the log when the file synchronize operation is not immediately performed, the Access Manager installs signal handlers for segmentation faults and bus errors (in addition to other signals). When a fault occurs, the Access Manager performs flush and synchronize operations on the log file.

the QRPC in its stable server log and places it in an incoming QRPC queue. After processing each QRPC, the server checks the queue to see if any of the messages in it are now acceptable. The message reception records in the stable log provide a total ordering over messages received from multiple clients.

Before starting execution of the QRPC, the server also appends a *start-of-execution* record for the QRPC to its stable log; thus, there are two logging actions on the critical path of QRPC execution. The execution record is necessary because it used during failure recovery to determine whether the QRPC started execution before the failure.

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

During failure recovery, the start and completion records are used to determine the status of each QRPC in the log: never executed (no start or completion record), executed and incomplete (start record only), executed and completed (start and completion records).

3.2 Server logging of intermediate values

Application support for intermediate value logging is provided by extending Tcl, the programming language used to construct RDOs, by adding some fault-tolerant features: support for stable variables and support for failure recovery procedures.

Stable variables allow simple, long-running applications to stably store information that is potentially expensive to reconstruct.

Programmers can declare stable variables anywhere within an application. Stable variables are identical to ordinary global variables. The difference is that, for stable variables, the Rover toolkit records in the stable server log the stable variable's existence and initial value, if supplied. Using Tcl's variable tracing features, the toolkit also traces writes to the variables and then records the changes in the stable server log. The recorded values are used during recovery as a form of REDO log [10].

We decided to require that programmers explicitly declare stable variables because of the high overhead associated with both tracing writes to a stable variable (approximately 2 times the cost of a write to a volatile variable) and recording changes in a disk-based stable server log (approximately 100 times the cost of a write to a volatile variable). Section 5 provides a detailed discussion of the costs associated with using stable variables.

Programmers can specify Tcl and C/C++ procedures that will be invoked during failure recovery. The procedures are responsible for performing the following actions:

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

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

3. Restoring any stable variables by invoking a toolkit function.

4. Invoking the Tcl recovery procedure, if one was specified. When the Tcl recovery procedure is invoked, it should resume the computation that was in progress at the time of the fault.

Each application is responsible for determining the state of an RDO at the time of a failure. The application can use stable variables to record incremental changes that are made to or by an RDO. Alternatively, an application could record changes in a log belonging to the RDO (*e.g.,* a separate stable log for the application). In keeping with the Rover design philosophy, the choice is left to the application programmer.

An alternative approach would be to periodically checkpoint the application's state. The frequency of checkpointing affects amount of lost work after a failure. However, this would not give application designers fine grain control over which data is stably logged and when it is stably logged. Stable variables provided application designers with fine grain control.

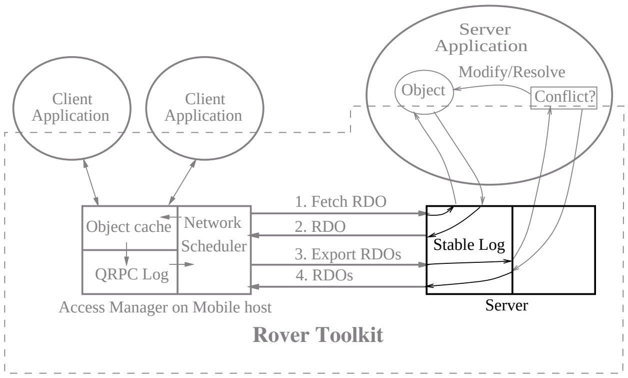

Figure 3: The extensions to the Rover toolkit in support of reliable execution of RDOs at servers: logging of incoming QRPCs, stable variables, and logging of outgoing responses.

3.3 Server logging of outgoing responses

The Rover toolkit allows application designers to specify that the server should append a record containing the application's outgoing response to the stable server log. Response logging is very useful when the generated response is expensive to recreate after a failure.

The response log records are used during the stable server log scanning phase of failure recovery. For each incomplete or undelivered QRPC, if the log contains an outgoing response record for that QRPC, the server sends it instead of reexecuting the QRPC.

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

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

3.4 Server failure recovery

During recovery from a server failure, the server's stable log is used as a REDO log. For each record of an incoming QRPC, the recovery process looks in the stable server log for a record of the QRPC's completion. If the server encounters an incoming QRPC record without a corresponding completion record, it takes one of the following actions:

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

2. If the stable server log contains a record of the start of the QRPC's execution and there is an application-provided recovery procedure, the server invokes the recovery procedure.

3. Otherwise, the server simply reexecutes the QRPC using the record of the original QRPC stored in the log.

Reliable applications can reduce the amount of work that is redone after a failure by using the programming language support to record intermediate values in the stable server log.

4 Server-side Implementation

We have extended the Rover Toolkit implementation to add server-side support for reliable operation. On the server, we provide a stable log for incoming QRPCs, some intermediate values, and outgoing responses. In addition, we have implemented server failure recovery procedures and server failure detection. In this section, we discuss the implementation of the various components of server-side support for reliable operation.

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

(a). Volatile Tcl counter

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

(b). Stable Tcl counter

Figure 4: Volatile and stable Tcl counters.

4.1 Rover server

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

There are two implementations of Rover servers. One is a standalone TCP/IP server that provides a very restricted subset of HTTP/1.0. A single standalone server provides service to multiple clients. The other implementation is compatible with the Common Gateway Interface (CGI) [18] of standard, unmodified HTTP compliant servers (*e.g.,* CERN or NCSA's httpd servers). A new copy of the CGI-based server is executed for each connection from a client; this makes it expensive to maintain shared state. Because of the state sharing restrictions of the CGI-based server, the extensions are only applicable to the standalone server; as such, in this paper, we discuss only the standalone server in detail. For more information about the CGI server, see [16].

4.2 Stable Server Log

Crucial to the implementation of reliable operation is a stable log of the server's actions. Like the Rover clients, the Rover server uses ordinary UNIX files (located either on a hard disk, a battery-backed Static RAM PCMCIA card, or on a Flash RAM PCMCIA card) to implement a stable server log of incoming QRPCs. Thus, log flushing is on the critical path for message reception and processing.

During execution of the QRPC, the application may use the language primitives described below to log intermediate values to the stable server log. As discussed earlier, to avoid having to reexecute completed QRPCs whose results were not delivered, applications have the option of specifying that the server record the application's output in the stable server log.

During failure recovery, the Rover server scans its stable server log looking for any QRPCs that were received but not processed or that were processed but whose results were not delivered to the client. For incomplete or unprocessed QRPCs, the server follows the recovery procedure outlined above.

Rover provides Tcl language support for reliable operation of RDOs executing at the server. Programmers declare stable or non-volatile variables using the `Rover_stable` *variableName [initialValue]* declaration to specify the name and, optionally, the initial value for a stable variable. The server saves the declaration in the stable server log file and sets a Tcl variable trace on the variable. The Tcl variable trace procedure registers an application that will be invoked when the specified variable is modified. Rover uses this procedure to detect changes to stable variables; the changes are then logged to the stable server log.

Programmers use `Rover_setRecoveryProc` to specify the name of a special Tcl procedure to be used during recovery; the server will save the procedure name in the stable server log.

Each server application can specify a C recovery procedure that will be invoked during the recovery process. The toolkit provides routines that application designers can use to perform the necessary steps.

During failure recovery, for each QRPC that needs to be reexecuted, the server invokes the application's C recovery procedure. The C recovery procedure should do the following: restore the application's C/Tcl environment, restore the RDO's stable variables by calling the server's `restoreStableVars` function, and, if specified, invoke the RDO's Tcl recovery procedure. If no Tcl recovery procedure is specified, the recovery procedure should reexecute the QRPC. During failure recovery, the values of stable variables in the stable server log take precedence over the initial values specified by `Rover_stable` declarations. This is because the logged values represent later states of the variable.

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

4.3 Process Pair-Based Fault Detection

The standalone Rover server operates as a pair of processes. The server is automatically started at system startup time and immediately forks a copy of itself. It then performs a wait operation on the child process. Meanwhile, the child server initializes its state, performs recovery (if necessary), and prepares to receive and process requests. The child server installs signal handlers for all the UNIX signals that can be caught. If the child server encounters an unrecoverable error or a signal that cannot be handled, it terminates operation. The parent process will detect this and start a new child server. Because the parent server's operation is extremely simple (wait and restart a process), it should not be vulnerable to many software faults.

4.4 Server Response Redelivery

As mentioned earlier, the Rover toolkit supports split-phase communications. For efficiency, the toolkit normally sends a request-response pair over the same connection. However, the connection may be lost as a result of communication failures, in which case, the server will retry the send using a new connection.

The extensions to the server allow it to retry sending a response if the connection back to the client is lost. The client-side support for this reuses functionality added to the Access Manager to support the Rover Web Browser proxy application [16]. The Access Manager provides an httpd-like interface port that allows unmodified client WWW browsers to operate in an offline manner by submitting HTTP requests directly to the Access Manager (instead of a remote WWW server).

Each QRPC from the client includes the client Access Manager's IP address and httpd port number. In the event of a communication failure, the server will periodically attempt to open a connection to the Access Manager.

5 Results

We designed a set of experiments to measure how effectively the Rover toolkit extensions meet our goals. In particular, the experiments test the following hypotheses:

1. The overhead associated with addition of server logging of QRPCs is acceptably small.
2. It is important to have a clear demarcation between stable and volatile variables due to the overhead associated with stable variables.
3. End-to-end performance of non-fault-tolerant applications is minimally affected by the support for building reliable applications.
4. End-to-end performance of fault-tolerant applications is good.

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

5.1 Experimental Environment

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, the workstations are also used as clients of other workstations.

Server: Pentium 120	Transport	TCP	
		Latency null RPC	Throughput send 1 MByte
Client: TP 701C/75	Ethernet	8	4.45
	WaveLAN	35	1.09
	128 ISDN	74	0.57
	64 ISDN	87	0.32
	19.2 Wired CSLIP	430	0.027
	9.6 Cellular CSLIP	2230	0.008

Table 1: The Rover experimental environment. Latencies are in milliseconds, throughput is in Mbit/s.

The network options consist of switched 10 Mbit/s 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) [14] 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 a Intel Advanced/EV workstation running the standalone TCP server. The clients were IBM ThinkPad 701C laptops. All of the machines were otherwise idle during the tests.

To minimize the effects of 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, the 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.

5.2 Baseline Performance

The addition of stable logging to the Rover server changes the overhead associated with using QRPC as a transport. To examine the effects of the extensions, we repeated the QRPC cost experiments from [16].

The cost of a QRPC has several primary components:

1. Transport cost. This is the time to transmit the request and receive the reply. The cost should not have changed from the original implementation.

2. Stable logging cost. The time to log the outgoing QRPC at the client and log the incoming QRPC at the server. Relative to the original implementation, this cost will increase by the cost of server-side logging.

3. Execution cost. The time to process the QRPC at the server. This cost has been reduced due to other unrelated changes to the server.

5.2.1 Transport costs

The transport costs for QRPC are dependent upon the latency and bandwidth of the network technology. We measured various representative networks; the results are summarized in Table 1. The table shows the latency for null ping-pong and the throughput for sending 1 Mbyte of ASCII data using TCP sockets over a

[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 imposed a substantial added latency of approximately 600 ms, but also yielded 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).

Media	Capacity (megabytes)	Seek time (ms)	Rot. Latency (ms)	RPM	Max. transfer rate (MByte/s)	Interface	Read time (ns)	Avg. 16-bit write time (ms)
IBM 2590 disk	2458	9.8	6.95	4316	3.0	SCSI-II	—	—
IBM 2540 disk	540	13	7.5	4000	11.1	IDE	—	—
Intel Flash Card	40	—	—	—	10.0	PCMCIA	150	0.006

Table 2: Attributes of the media used for stable logging (from manufacturer's specification sheets).

Machine	Latency		
	Disk	Flash RAM	Simulated Static RAM
Pentium server	30.0	16.9	1.0 – 3.0
ThinkPad client	37.5	24.8	4.0 – 13

Table 3: Time in milliseconds to log 290 bytes to stable storage, including a synchronous flush of the file buffers for the log to the storage media. Linux uses a linked list to store the blocks containing the RAM disk data; thus, the times for the simulated Static RAM are a function of the log file size (1000–4000 operations were performed for the measurement).

number of networking technologies. The throughput over wireless CSLIP is lower than expected (8.2 Kbit/s instead of 9.6 Kbit/s), because of the overhead of the ETC protocol and retransmissions due to errors on the wireless links.

On the other hand, the throughput over wired CSLIP is higher than expected (28 Kbit/s versus 19.2 Kbit/s), because of the compression that is performed by the modem on ASCII data. Likewise, the ISDN routers also performed a significant amount of compression (584 Kbit/s versus 128 Kbit/s and 328 Kbit/s versus 64 Kbit/s). Both links benefit because the 1 Mbyte of ASCII data used for the test is very compressible (GNU's *gzip -6* yields a 14.4:1 compression ratio). Since RDOs are constructed of Tcl scripts (ASCII), it is reasonable to expect that Rover applications will also observe similar compression benefits when using links with hardware compression.

5.2.2 Stable logging costs

We measured the performance of the stable logging of QRPC to disk, Flash RAM, and simulated Static RAM. The server has a PCI bus-based Adaptec AHA-2940 Ultra Fast SCSI-II interface to an IBM 2590 disk. The clients have IBM DBOA-2540 disks. Both clients and servers are equipped with Intel Series 2+ Flash Memory PCMCIA cards. We used RAM disks to simulate the performance of battery-backed Static RAM PCMCIA cards. All of the devices used Linux's ext2 filesystem [6] with 1024 byte block and fragment sizes. The attributes of the various storage media are listed in Table 2.

Flash RAM was chosen as an alternate stable storage media to disks because it offers the advantage of significantly lower power utilization (0.06 milliwatts to 1 watt, 0.6 milliwatts average) [12], versus the DBOA-2540 disk drive (0.1 to 4.65 watts, 2.3 watts average) [13]. However, the reduction in power utilization comes at a price. One important consideration with Flash RAM is the highly variable write time. While the time to read a 16-bit word is a constant 250 nanoseconds, the time to write a 16-bit word varies from 6 microseconds to 3 milliseconds. Furthermore, the time to write one 512-byte block (the size of erase units for the Series 2+ cards), varies from 0.4 to 2.1 seconds.

In addition to the power benefits relative to disks, Static RAM also offers the substantial advantage over Flash of constant read and write times. However, Static RAM devices have two significant drawbacks: they require a battery (a potential failure point) because they are not truly non-volatile and they are significantly more expensive than comparable size Flash RAM devices.

The Flash RAM can be accessed using either a character or block device interface; the experiments used the disk-like block device interface. The block interface is implemented by a Flash Translation Layer (FTL), which also provides automatic block copying, asynchronous erasing, and wear leveling functions. Using the ext2 filesystem on top of the FTL, each synchronous flush of data to the Flash RAM required writing two

Storage media	Volatile	Tracing only	Stable logging	
			Async.	**Sync.**
None	0.17	0.17	—	—
Disk	—	—	0.34	24
Flash RAM	—	—	0.34	0.40
Static RAM	—	—	0.32	1.5

Table 4: Time in milliseconds to execute each iteration of the example counter code in Figure 4 at the server.

Flash RAM blocks for each updated data block and two Flash RAM blocks for the inode. In addition, the write of each Flash RAM block by the FTL required three 32-bit writes to update the FTL block map. The combination of slow writes and large numbers of writes per block (resulting in many block copies and erasures) can cause the performance of the Flash RAM to be significantly lower than expected. The use of an alternative filesystem or a Flash RAM coupled with a small battery-backed SRAM cache could offer substantial performance benefits [9].

The results of the stable logging experiments are presented in Table 3. The faster processor and better disk subsystem of the server yield significantly better logging performance than the client. The Flash RAM also performed significantly better than the disk. Linux uses a linked list to store the blocks containing the RAM disk; thus, the times for the simulated Static RAM are a function of the log file size. During the experiments, the log file ranged in size from 0 to 290,000 bytes (1,000 log writes) and 0 to 1,160,000 bytes (4,000 log writes).

5.2.3 Execution costs

The measured cost of processing a null QRPC at the server is 1.6 milliseconds. The processing time only included the time to parse, dispatch, and execute the QRPC.

5.3 Null QRPC Performance

Figure 5 summarizes the results of combining the transport, logging, and execution times for a null QRPC. For a null QRPC, approximately 290 bytes are sent and 5 bytes are received. The horizontal line for each network is the time for asynchronous flushing of log buffers to disk. During the experiments, the server was not processing any other messages and the interarrival rate of QRPCs was sufficiently low enough to allow buffer flushing to occur while the server was idle.

Some observations:

1. The performance of asynchronous log flushing is close to the performance when no logging is performed at the client or the server.
2. The performance when logging using a RAM disk at both the client and server is close to the performance when no logging is performed at the client or the server.
3. The cost of adding synchronous client-side logging matches the expected costs from Table 3.
4. Synchronous server-side logging has an associated cost that is approximately twice the cost of adding client-side logging. This is because the server synchronously logs both the incoming QRPC and the start-of-execution record for the QRPC.

Overall, the results show that the relative impact of logging is a function of the transport media. Since we expect that users will often be connected via slower links (*e.g.,* wired or cellular dialup), the cost of stable logging when using a disk will be a minor component of overall performance (*e.g.,* less than 7% for wired and cellular dialup links). Given that, in a mobile environment, the time to retransmit a QRPC from a client to a server after a failure is likely to be significant (*e.g.,* a client disconnected for an extended period of time or connected over a high latency link), we believe it is acceptable to pay a small performance penalty for server logging of incoming QRPCs. Furthermore, by using asynchronous log flushing, a small reduction in reliability is traded for a significant performance gain.

For highly connected, relatively faultless environments, like Ethernet and WaveLAN, clients and servers can determine when and how to use the reliability extensions (*e.g.,* by either logging less data or asynchronously flushing log buffers). The high cost of fault-tolerance need be paid only if necessary. In addition, when possible, the Access Manager's network scheduler batches requests to a server. Each request is still individually logged, however, this approach increase overall throughput by avoiding the roundtrip delays of earlier messages.

5.4 Stable Variables

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

At the server, the extra cost to trace a write to a Tcl variable is negligible. Adding stable variable logging with asynchronous log buffer flushing doubles the cost versus a volatile variable. Using synchronous logging to a disk, the cost per iteration is two orders of magnitude higher than when a volatile variable is used. When Flash RAM is used as the stable media, the cost per iteration is one order of magnitude higher than when a volatile variable is used. However, when SRAM is used as the stable media, the cost is only 2.5 times the cost of using a volatile variables.

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

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

5.5 Application Development and Performance

To measure the effect of the fault-tolerant extensions on end-to-end application performance, we constructed two applications: a stock market tracker and a file search tool. We also modified the server portion of the Rover Web Browser proxy. The applications were chosen to evaluate the ease of constructing reliable applications or modifying existing applications and to measure the performance of reliable applications.

Creating a new application or modifying an existing one involved two steps: identifying objects that should be made stable and providing the necessary recovery procedures.

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

For the second step, the Rover toolkit provides application designers with help in creating the necessary recovery procedures. The toolkit provides procedures for performing some of the recovery steps, including procedures for restoring stable variables and invoking the Tcl recovery procedure.

5.5.1 Stock tracker

As an exercise to gauge the added difficulty in constructing reliable applications, we constructed a simple financial stock tracking application. The application constructs and sends an RDO to the server. The RDO executes at the server and watches the attributes of a user-specified stock for changes that exceed a user-specified threshold (*e.g.,* price, volume, or changes). When the threshold is exceeded, the server notifies the client.

The RDO uses stable variables to store the stock's current attributes. Using stable variables instead of storing the information in a file greatly simplified the construction of the application (*e.g.,* using stable variables avoided the construction of marshalling and unmarshalling procedures).

Stable storage	None	Volatile only	Stable logging	
			Async.	Sync.
None	327	336	—	—
Disk	—	—	418	420
Flash RAM	—	—	354	429
Static RAM	—	—	327	417

Table 5: Execution time in milliseconds for the server portion of the Rover Web Browser proxy. The experiment measured the time to fetch a copy of the Rover project home page and its inlined images, *http://www.rover.lcs.mit.edu/* .

5.5.2 Rover Web Browser proxy

The Rover Web Browser proxy is a client-server application consisting of approximately 1000 lines of server code and 2900 lines of client code. The modifications to support reliable operation only affected the server portion of the application.

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

The proxy was modified to make the hash table it uses stable. This was a very simple modification consisting of changes to five lines of code.

The proxy experiments measured the time to fetch the Rover project home page and its inlined images, http://www.rover.lcs.mit.edu/ . To help reduce the variability of the results, the experiments used a local httpd server running on the same machine as the Rover server. The page consists of three elements: two pictures and the HTML text. The size of all the objects with compression is 56477 bytes. Only disks were used for stable storage.

The first proxy experiment measured the performance of the server portion of the proxy (the time from when the Rover server dispatched the proxy module to when the module returned). The times include the transport and processing overheads of the httpd server (running locally) and some of the transport overhead of sending the data back to the client.

The results presented in Table 5 show the execution times when the proxy was implemented without hash tables, with volatile hash tables, and with logged stable hash tables with asynchronously and synchronously flushed buffers. As can be seen from the table, the other costs (httpd server overhead and transport cost) and their variability dominate the results. Overall, the results show that the cost of stable hash tables is an acceptably small component of the execution time of the proxy.

The second proxy experiment measured the end-to-end performance of the proxy. The results presented in Figure 6 show that server logging of QRPCs and stable hash tables impose very little overhead on normal operation. Note that the numbers for cellular CSLIP reflect the high degree of variability of the wireless link.

What both Table 5 and Figure 6 do not show is the primary benefit to the proxy of using a stable hash table — after a server failure, it allows the server to avoid sending duplicate objects to clients; this conserves potentially expensive and/or limited bandwidth.

5.5.3 Text file search

We constructed a simple text file search application that uses stable variables to store information about the state of the search (*i.e.*, the remaining files and directories to be searched and the matches found so far). An excerpt from the server code is provided in Figure 7. The difference between the stable and volatile versions is only a few lines of code. By using stable variables, after a failure, the search in-progress can be resumed with only a small amount of lost work.

<table>
<tr><th rowspan="2">Failures</th><th rowspan="2">Volatile</th><th rowspan="2">Tracing only</th><th colspan="2">Stable logging</th></tr>
<tr><th>Async.</th><th>Sync.</th></tr>
<tr><td>None</td><td>11.2</td><td>11.2</td><td>11.3</td><td>18.6</td></tr>
<tr><td>Single (75%)</td><td>20.4</td><td>—</td><td>12.1</td><td>19.6</td></tr>
</table>

Table 6: Time in seconds to execute the file search application (using Flash RAM for stable logging). For the single failure case, the fault was injected after 75% of the files were searched.

We used the file search application to scan 1.9 Mbytes in 328 files in 3 directories both without faults and with a single fault injected after 75% of the files were searched. The results are presented in Table 6.

The overhead associated with tracing writes to the stable variables was negligible. When there are no faults, the performance using logged stable variables with asynchronously flushed buffers is nearly identical to the performance using volatile variables. However, when synchronous logging is used, there is a significant performance slowdown.

When a single fault is introduced, the execution time of the volatile variable version of the application is nearly doubled. The stable versions of the applications show only a small degradation in performance compared to the failure-free case.

As mentioned earlier, logging with asynchronous flushing trades a small reduction in reliability for a significant performance gain.

6 Related Work

Some of the ideas behind the extensions to the Rover toolkit were drawn from existing work in distributed computing. In applying them to the Rover toolkit we are moving them to a domain where failures are frequent and network transports offer intermittent connectivity, high latency, and low bandwidth.

There is a large body of research on logging and distributed fault-tolerant transactions; for an excellent discussions of some of the issues, see [10] and [11].

Other systems have addressed some of the problems relating to reliable communications. The Tacoma project explored the use of rear guard agents to guarantee agent delivery and execution [15]. More recent work on Tacoma relies upon Horus for fault-tolerant communication and execution [23]. ISIS defined an environment for fault-tolerant, group communication-based computing [4]. The failure model used by ISIS is a fail-stop model, which requires recovering processes to recover their state from other active processes instead of a log.

The state of guardians in the the transaction-based Argus system is split into stable and volatile variables [17]. Recovery relies upon replay of a local stable log. Likewise, the state of objects in the Clouds distributed operating system project [1] is split into permanent and volatile data. Clouds also provided computation fault-tolerance support for mobile objects by using primary and backup schedulers.

The Bayou project [8, 22] uses a peer-to-peer database model for sharing data among mobile users, where mobile hosts store updates locally in a stable log and communicate with other mobile hosts to propagate the changes. Conflicts are resolved using log replay.

Several commercial systems for mobile environments, including Telescript [25] and Oracle Mobile Agents [7], offer reliable message delivery. Other systems providing mobile code that most closely resembles the mobile code aspect of RDOs are Ousterhout's Tcl agents [19] and Java [2, 20]. However, none of these systems support reliable mobile code.

7 Conclusions

Long-running applications are the most likely to be affected by transient software and hardware faults. We have found that adding the fault-tolerant features described in this paper to the Rover Toolkit has yielded a powerful tool for building mobile-aware applications that are reliable in the presence of such faults.

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

Stable logging at the server provides two key advantages in the case of a server failure. First, retransmission costs are avoided. In addition, once the request is logged by the server, it will be processed without further client intervention (in particular, the client need not even remain connected).

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

Acknowledgments

We thank the anonymous reviewers, Greg Ganger, Robert Gruber, Eddie Kohler, Massimiliano Poletto, Joshua Tauber, and Deborah Wallach for their careful readings of earlier versions of this paper and insightful discussions.

References

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

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

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

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

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

[6] R. Card, T. Ts'o, and S. Tweedie. Second extended filesystem. In *Proc. of the First Dutch International Symposium on Linux*, Amsterdam, Holland, 1994.

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

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

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

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

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

[12] Intel Corporation. *Series 2+ Flash Memory Cards: 4-, 8-, 20- and 40-Megabyte.* Order Number: 290491-006, March 1996.

[13] International Business Machine Corporation. *2.5-Inch Travelstar LP 360, 540 and 720 MB Low Profile Disk Drives.* http://www.ibmlink.ibm.com/HTML/SPEC/goem7054.html, April 1996.

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

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

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

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

[18] National Center for Supercomputing Applications. *Common Gateway Interface.* http://-hoohoo.ncsa.uiuc.edu/cgi. University of Illinois in Urbana-Champaign, 1995.

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

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

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

[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 ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, 1995.

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

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

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

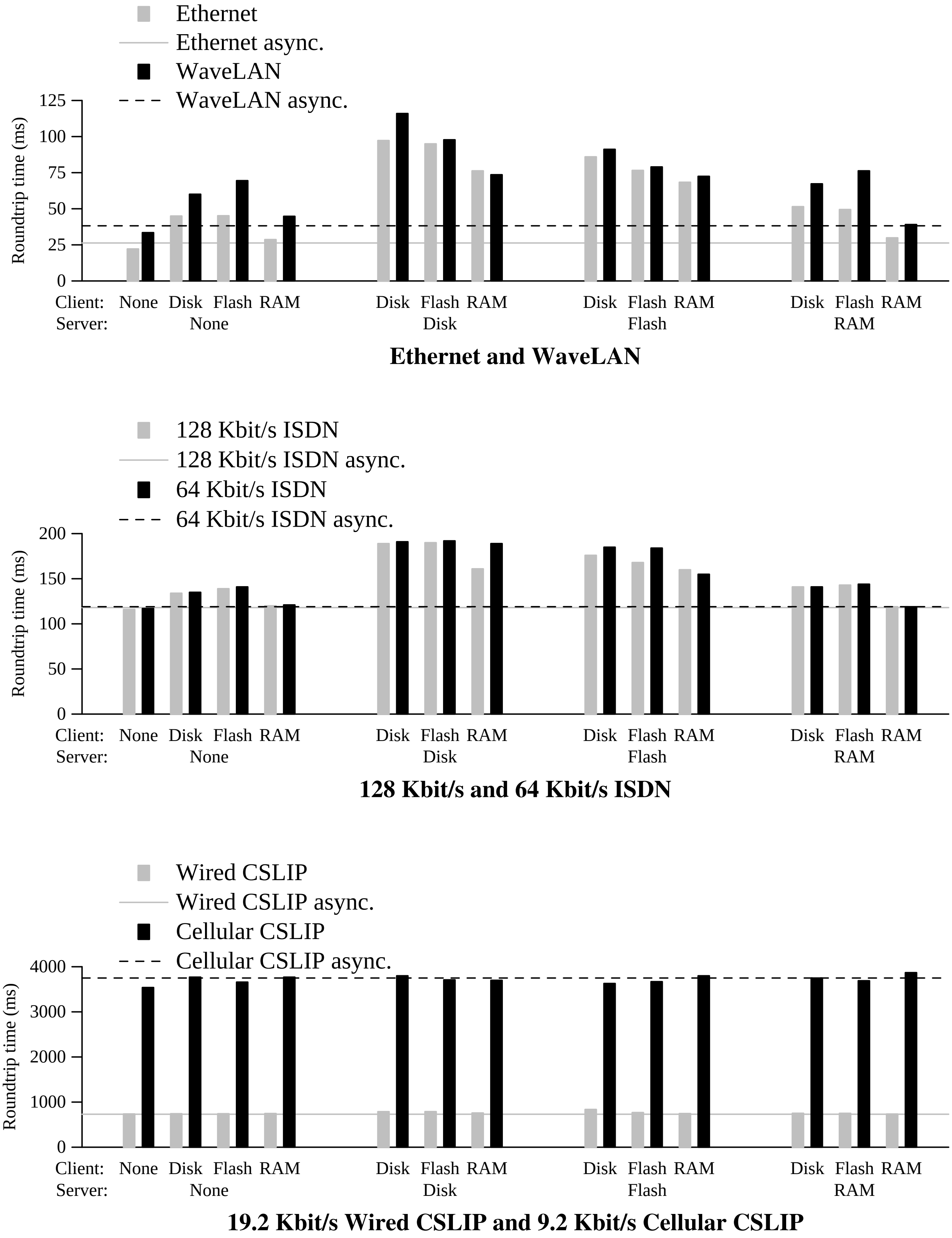

Figure 5: Roundtrip time in milliseconds to perform a 200 byte QRPC using Ethernet and WaveLAN networks. The horizontal line for each network is the time for asynchronous flushing of log buffers to disk.

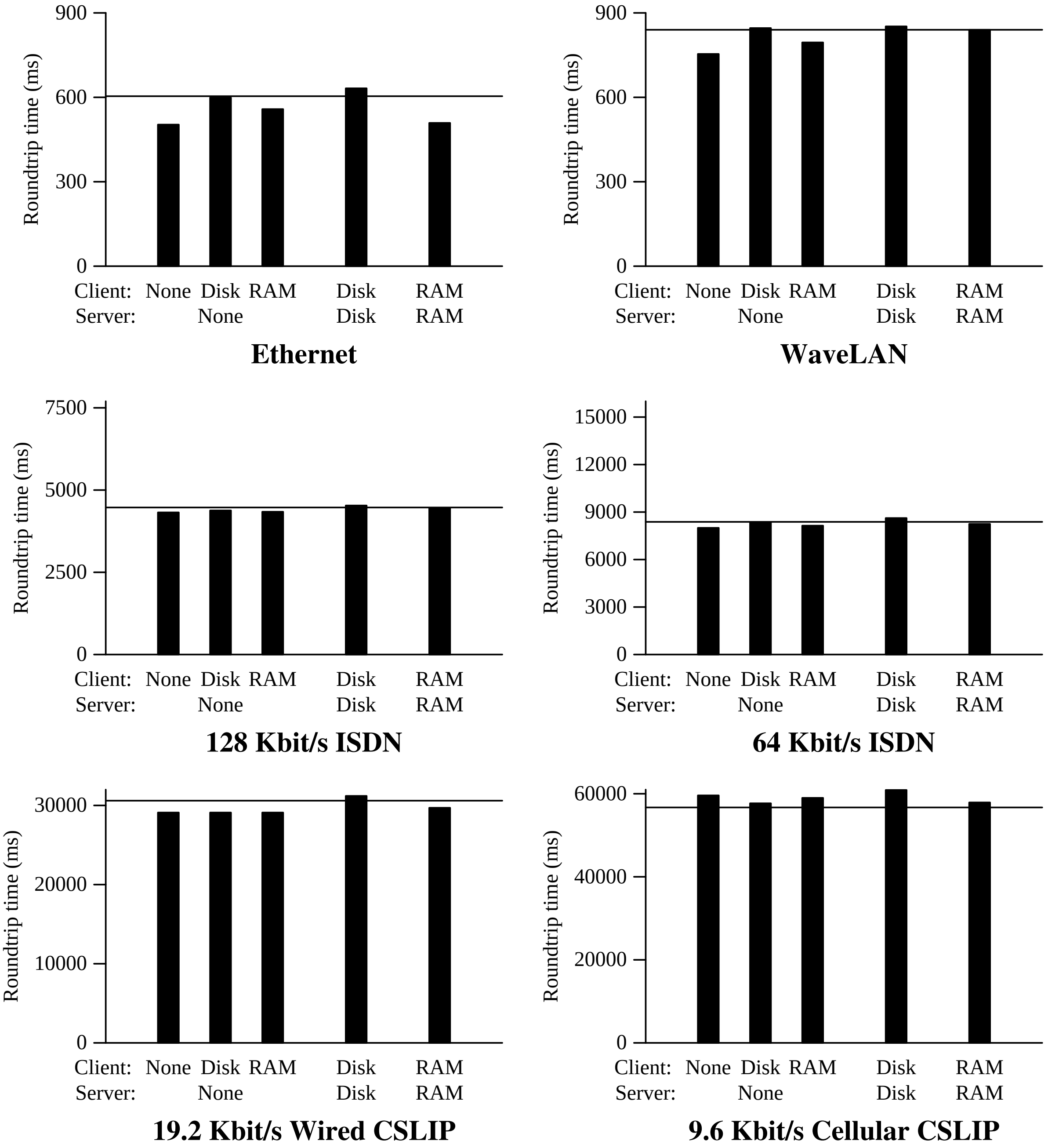

Figure 6: Round trip time in milliseconds for the Rover Web Browser proxy to fetch a copy of the Rover project home page and its inlined images, *http://www.rover.lcs.mit.edu/* . The round trip time is the total time to fetch the documents. The horizontal line on each bar graph is the time for asynchronous flushing of log buffers to disk.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 7: Server-side code for a reliable file search application.