Using Simple Remote Evaluation to Enable Efficient Application Protocols in Mobile Environments

Steven E. Czerwinski and Anthony D. Joseph
UC Berkeley, Computer Science Division
Soda Hall
Berkeley, CA 94720-1776
{czerwin,adj}@EECS.Berkeley.EDU

Abstract

This paper describes the REAP toolkit, a reusable solution for enabling Remote Evaluation in Application-level Protocols. By using this toolkit, server developers can reduce the number of bytes sent/received and round-trip-times used by their protocols to fulfill a user task, making them more efficient in high latency, low bandwidth networks. This is accomplished by allowing the clients to upload simple mobile procedures to the server to be executed locally. This paper provides an overview of the REAP toolkit design and architecture, and gives experimental results showing significant reduction in the latency of two popularly-used application-level protocols.

1 Introduction

As new network access methods become available, developers are faced with the difficult task of creating distributed systems that behave well under diverse latency and bandwidth conditions. Users expect reasonable performance from these systems, whether their device is connected via Gigabit Ethernet or a packet radio modem. If the developer has complete control over the application-level protocol used to communicate information between the client device and the distributed components, then the developer can use several optimization tricks to hide the latency of the network and reduce bandwidth consumption. However, if the application-level protocol is a predefined standard, such as SMTP [17], IMAP [4], HTTP, or LDAP [26], then the optimizations that can be performed are greatly constrained because both the protocol commands and wire format are fixed. If fact, we often find that clients must issue multiple protocol commands and use more bandwidth than necessary in order to fulfill simple user requests. In low-latency, high-bandwidth networks, this has little affect on client performance, but as the latency increases and the bandwidth drops, performance suffers greatly.

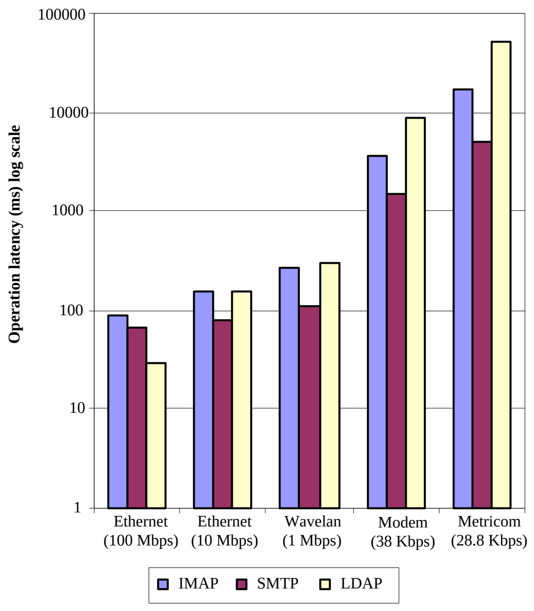

Figure 1. Client sessions over various network technologies (log scale)

Figure 1 illustrates this performance drop by showing the latencies of typical client sessions for three popular application-level protocols: IMAP (mail access), SMTP (email transmission), and LDAP (database information access). Note, the y-axis is in log scale. For IMAP, we traced the login sequence of a user with 20 folders and 100 INBOX messages using the Netscape Communicator e-mail client, which performs what we refer to as *client synchronization* consisting of authentication, message synchronization, and downloading of new mail. For SMTP, we traced the SMTP

	Bytes Sent	Bytes Received	Blocking Calls
IMAP	541	6694	18
SMTP	1058	498	7
LDAP	2769	107777	6

Table 1. Session characteristics

	RTT (ms)	BW (Kbps)
100 Mbps Ethernet	.21	66560.2
10 Mbps Ethernet	1.23	7335.7
Wavelan	3.8	1120.7
Modem	154.5	33.8
Metricom	672	20.4

Table 2. Network characteristics

commands Communicator issues to send a 2 KB message to a single recipient. For LDAP, we traced how Communicator uses LDAP to save a user's 100 KB bookmark file in a central LDAP server to implement its Roaming feature. Table 1 lists the amount of bytes the client sends/receives and how many times it blocks for an intermediate response in each experiment. Table 2 lists the round-trip-time (RTT) estimate and measured throughput for each network. The RTT was computed using 20 trials, while the throughput was computed by the time it took the client to send a 200 KB file.

As Figure 1 shows, the latencies for all three protocols increase by an order of magnitude in high-latency, low-bandwidth network settings, as represented by the modem and the Metricom Ricochet packet network [14] measurements. Looking at individual times, we see that the IMAP client synchronization takes 0.12 seconds when using 10 Mbps Ethernet, 3.5 seconds with the modem, and 16.0 seconds when using Metricom. Similarly, the LDAP trace only takes 0.16 seconds with the 10 Mbps Ethernet, and 50.6 seconds with Metricom. Obviously, these application-level protocols need to use these different networks much more efficiently.

After investigating these protocols and their performance problems, we find that a major contributing factor to the increased latencies is extraneous RTTs incurred because clients must issue multiple blocking protocol commands to fulfill one client task. For example, the IMAP client synchronization consists of 18 blocking commands. In this paper, we present a solution to this problem in the form of the REAP toolkit. The toolkit reduces client latencies by allowing clients to upload mobile procedures to the server for remote execution [20]. By making the servers programmable, we essentially introduce a "slop" factor into the application-level protocol, giving the clients some room to optimize their performance. Of course, the extraneous RTTs are not the only performance problem, but it is the one we are focusing on in this paper.

We argue that it is inherently difficult to build application-level protocols that issue only one blocking command per user action, so a "slop" factor is needed in general, and not just for the IMAP protocol. This is because application-level protocols, such as IMAP, are standardized which essentially means their protocol command set is a compromise between many parties. No one client developer will get the exact commands they desire. Furthermore, standardization favors creating protocols with small, simple commands because the commands are easier to discuss and agree upon. For example, it is understandable that IMAP has separate commands to select a mailbox and to list its contents, even though they are almost always used together. This leads to clients issuing multiple commands to perform complex actions such as cache synchronization. Finally, protocol designers cannot anticipate all future needs, so some flexibility must be included to evolve the protocol over time.

We should stress that our solution is meant to be fairly light-weight and reusable. We could have implemented a mobile code system in Java, but this would have added significant complications such as supporting objects, object creation, methods, and possibly unterminated control structures. Rather, we take a minimalist approach, creating a simple scripting language for protocols.

Additionally, our solution and experience is meant to be instructive for future application-level protocol development, rather than an absolute way to fix existing protocols. Our approach requires changes to both the client and server, which would be difficult for existing systems. We do use existing protocols to motivate our solution and to demonstrate what the performance savings would be if they used our techniques. Of course, by using proxies on both the client and server sides, we could transparently handle existing protocols, as described in 3.1.

Based on the concepts mentioned above, we have created the REAP toolkit, a reusable solution which can be used to enable remote evaluation in servers. In the next several sections, we will describe the design concepts and architecture of the REAP toolkit and its mobile procedure language. We will also describe how we apply the toolkit to existing protocols, focusing on results we have obtained from the modified IMAP server we have created, along with results from a SMTP proxy.

2 Design Principles

In this section, we describe some of the design principles of the REAP toolkit and explain how they will help typical server development.

```
A1 capability        ;(supported server options)
A2 authenticate login
+aW1hcHVzZXItMQ==    ;(username)
+VGVtcDEyMw==        ;(password)
A3 namespace         ;(get folder locations)
A4 lsub "" "#mh/*"   ;(1 LSUB per namesapce)
A5 lsub "" "#mhinbox*"
A6 lsub "" "*"
A7 lsub "" "#public/*"
A8 lsub "" "#news.*"
A9 lsub "" "#ftp/*"
A10 lsub "" "#shared/*"
A11 lsub "" "~*"
A12 list "" "INBOX"
A13 select "INBOX"
A14 UID fetch 1:* (FLAGS) ; verify cached flags
A15 UID fetch 4486:* (ENVELOPE) ; download new
A16 logout
```

Figure 2. Netscape IMAP login sequence

2.1 Remote evaluation to hide latency

The REAP toolkit's goal is to enable application-level protocols to behave efficiently across different network settings. As Figure 1 shows, this is not the case for existing protocols – increasing a link's RTT from 1.23 ms to 672 ms results in over a 15 second increase in IMAP client synchronization latency. Ideally, the latency should only increase by 2.7 seconds, which is the sum of the RTT difference and the information transmission time using the lower bandwidth. The reason for a 15 second increase is that Netscape's IMAP client synchronization method with a UW IMAP server consists of 18 separate protocol commands, which are listed in Figure 2. Netscape treats each of these commands as blocking, so each costs at least a RTT to perform. This means that for every increase in the link's RTT, there is roughly a 18 fold increase in IMAP client synchronization latency.

By using the REAP toolkit to embed remote evaluation at the server-side, we can eliminate most of these extraneous RTTs. If we examine the command sequence in Figure 2, we find that most commands block just to verify an "OK" is returned, to ensure correct execution before issuing the next command. Instead of checking this return value at the client, we can create a mobile procedure that batches all these commands and performs the status check at the server, halting execution upon failure. This is a very simple approach, but eliminates 16 of the 18 RTTs.

The two remaining blocking commands can also be eliminated by using the REAP's remote evaluation features. The first, the `CAPABILITY` request used to determine the available optional commands, is eliminated by using an if-then statement in the mobile procedure to have the server issue different requests based on the result. The second command, the `NAMESPACE` request used to determine where a user's folders are stored, can be eliminated by iterating over the `NAMESPACE` response at the server, issuing a `LSUB` for each namespace returned. Iterators will be described in Section 3.5.

Thus by employing a very simple remote evaluation model, we are able to eliminate all of the extraneous RTTs needed by the Netscape client. As Section 4 will show, these results are equally applicable to other IMAP clients. We should point out that the IMAP protocol does allow for asynchronous operations, but none of the clients we examined used this feature in order to reduce client complexity. This provides further motivation for a toolkit solution, since it hides complexity from the clients and servers.

2.2 Simplified language helps security

As in any system that allows remote execute of code, the REAP toolkit must ensure the server cannot be disrupted by the mobile procedure sent by the clients [7, 9, 16, 22]. This problem is somewhat simplified in REAP, since the REAP toolkit language only allows the clients to perform a very small subset of actions, reducing the client's ability to abuse the server. Specifically, the language does not allow clients to define internal procedures, allocate memory for variables other than integers and booleans, or perform arbitrary loops. These restrictions eliminate the possibility that a mobile procedure might infinitely loop, perform recursive procedures, or use excessive amounts of memory. Moreover, the client mobile procedure will always execute in $O(c+r+a)$ time and use $O(r+a)$ space, where c is the length of the mobile procedure, r is the length of the largest response returned by the server to the client code, and a is the length of the largest user supplied argument. Additionally, the REAP toolkit uses an interpreted language, so the toolkit can enforce various security policies at runtime, such as limiting execution time and the number of requests that can be issued by a mobile procedure.

2.3 Built-in request/response support

Since the REAP toolkit is specialized for application-level protocols, we were able to build request/response support directly into the toolkit's language which allows for many high-level optimizations. For example, the REAP toolkit allows for request reordering to improve overall efficiency, and has been designed to allow for incremental operation on partial responses. These optimizations would not be possible if we used a lower level language such as C or Java because much of the semantic notions of what responses and requests are would be lost in the implementation details.

2.4 Reusable solution

We have designed the REAP toolkit to be a reusable solution for various protocols. This approach follows the current trend in distributed systems of creating middleware tools [1, 10] to ease the development of higher-level applications. Since it can be used as a black box by many protocols, the REAP toolkit is an ideal place to implement advanced protocol optimizations. This choice both hides the complexity of the optimizations from protocol developers and reduces the chances they will be implemented incorrectly.

3 Toolkit Architecture

3.1 Overview

The REAP toolkit is currently implemented as a C library that exports a LISP-based [27] runtime environment. We have used the open-source librep project [11] as the basis of the LISP runtime environment. We chose to use C and LISP as the implementation and mobile languages merely for convenience – there is nothing inherent about the REAP design that ties it to either. In fact, we explicitly designed the REAP toolkit to not use object-oriented programming to make it more portable between different languages.

Figure 3 illustrates how the REAP toolkit ties into the server application. As the figure shows, the server application initializes the REAP library by registering the requests it wishes to handle and information about the responses it will produce (see Section 3.2 for more detail). The REAP toolkit will then accept and manage incoming TCP/IP connections for the server application. After REAP receives a complete mobile procedure from a client, it will execute it using the librep environment, upcalling to the server application whenever a server-specific request function is invoked. The REAP toolkit also invokes the server's handlers when sending response to clients, allowing the server application to specify the wire format. When the mobile procedure is finished, REAP sends a completion signal and closes the connection if instructed.

In addition to the architecture described above, we are building proxy support so that we can use unmodified clients and servers. In this scheme, two proxies are used, one next to the client and another next to the server. The unmodified client and server use the original application-level protocol when communicating with their proxy, but the proxies use REAP and the modified protocol between themselves. If the bottleneck link is between the proxies, then we should see performance improvements. However, this design presents difficulties because it requires a lot of careful design to hide the modifications from the client and server.

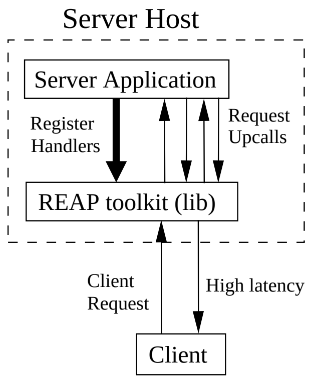

Figure 3. Server architecture

In the current implementation, the REAP toolkit specifies incoming wire formats for requests, while the server application specifies the outgoing wire formats for responses. We have attempted to make the request format universal enough that it can handle most protocols, but it is not an elegant solution. We would rather adopt a universal format, such as SOAP [25], as we will discuss in the future work section.

3.2 Request and response handlers

The request handlers are responsible for performing the majority of the work in the REAP toolkit. These server application hooks are invoked whenever REAP encounters a LISP procedure with a name registered to the server application while executing the client's mobile procedure. When invoking the handler, REAP passes the argument list and a pointer to the current LISP environment to the server application. The server application performs the request, either returning results in a LISP data structure or throwing an exception.

In the current implementation, the request handler must process the entire request before it can return, which is inefficient because the handler must hold the entire response in memory and the handler must block until all requests actions are complete. We are currently re-implementing the handlers to return partial response, as mentioned in Section 2.3. These partial responses may be incrementally propagated to operators, which will lead to better efficiency because asynchronous I/O can be used within request handlers. This approach does constrain the type of operators that can be used, but we have found it is sufficient for many tasks.

The server application also specifies response handlers for each response type it supports. When a response is sent to a client, REAP invokes these handlers to convert the response's LISP representation to the appropriate wire format.

3.3 Command syntax

All commands sent to the REAP toolkit are composed of three parts, a major tag, an argument list for the mobile procedure, and the mobile procedure definition. The major tag is a client-chosen string that is used to identify this particular client/server interaction, which the toolkit attaches to each response it generates during the procedure's execution. The second two components are fairly straightforward – they give the definition for the mobile procedure and its arguments.

As an optimization, we allow the mobile client to associate a name with the mobile procedure it defines so that subsequent calls may reuse the cached procedure object, even across different connections, as long as the name and definition match. Clients using popular procedures, such as Netscape's IMAP client synchronization procedure, would benefit because the cache-hit would avoid the costs of parsing and interpreting the mobile procedure. Additionally, we may extend this to include signatures on the procedures, in order to convey special privileges and relaxed security requirements to trusted parties.

The syntax of the mobile procedures follows standard LISP very closely, with the caveat that many language features have been removed. As discussed in Section 2.2, the mobile procedures cannot define large-sized variables (only integers and booleans are allowed), create other procedures, or loop. Responses are not allowed to be stored in variables, but are instead referred to by *labels*, which act as pseudo-variables and have scope within the entire procedure. We also extend the syntax to include minor tags, which are used to identify individual calls in exceptions for debugging.

We provide some examples of mobile procedures in Section 4.1.1 and Appendix A.

3.4 Operators

Table 3 lists and briefly describes the operators supported by the REAP toolkit. These operators act as the glue with which the server-provided request/response procedures are bound together to create more complex functionality. These operators may be overloaded by server applications, in case the default definitions are not appropriate for their response types. As the table shows, these operators are fairly simple, but they are sufficient to create fairly powerful mobile procedures.

3.5 Response and variable iterators

As mentioned earlier, the mobile procedures are not allowed to perform arbitrary loops, but they are allowed to loop over the response elements in a given response using the `iter-response` operator, and the elements in a procedure argument using the `iter-argument` operator. The response iterator takes in a response, a substitution variable, and a command sequence which is executed for each element in the response. The argument iterator behaves similarly, just on the elements of a client supplied argument. Without iterators or loops, mobile procedures would not be able to perform such tasks as counting response items, or invoking additional requests based on a previous response.

3.6 Response digests

Digest filtering is a pseudo-compression tool that prevents servers from sending information to the client it has already seen. In many applications, the data returned by a request does not change significantly over time. For example, each time a client requests a list of users from an LDAP server, the response received is not significantly different from the previous; it would be more efficient just to transmit the minor changes. Response digests approximate this functionality.

Describing our scheme at a high level, clients create a digest based on the most recent response they have cached, and send this digest as a parameter to the mobile procedure. The server uses this digest to determine what has changed, only returning the differences.

In our design, each response is composed of a set of attribute/value pairs describing the overall response, and a list of response elements referred to as the result listing. The semantic content of the result list is dependent on the response. For example, in a folder list request, each response element in the list would represent one folder. We have implemented our response digests to operate only on the results listing present in a response, which is the bulk of the data, and require that the listing is sorted. To construct the digest, a client segments its sorted results listing, creating a list of each segment's endpoint and the hash value of its data. To determine what has changed, the server segments its own result listing using the endpoints in the supplied digest, and compares its segment's hash value against the client's – if they differ, the entire segment is marked as changed and sent to the client.

This design has the nice property that the client chooses the resolution and weighting of the response digest. There is a trade-off between the size of the digest and its granularity. If the granularity is too coarse then a single change causes a large segment to be sent, but if its too fine, the digest will be large and consume more bandwidth. Similarly, the client can also perform trade-offs in weighting portions of the result listing differently. For example, the client can decide that the first half of the result listing rarely changes, so portion can have coarse resolution, while the later portion has finer.

Name	Description
`catch-exception`	Prevent a particular exception from halting execution
`create-response`	Create a generic response
`digest-filter`	Filter the given response based on a client digest
`iter-response`	Perform the specified commands for each element
`ok-response`	Test the response status
`response-attribute`	Return an attribute value for the response
`response-contains`	Test if the response contains the specified string
`send-response`	Send the response to client
`throw-exception`	Create and throw a generic exception
`union-responses`	Union the responses together to form a single response

Table 3. REAP toolkit operators

3.7 Exceptions

Exceptions are supported by the REAP toolkit for better error handling and cleaner syntax. In the IMAP example discussed in Section 2.1, we pointed out that 16 of the 18 checks performed are just to ensure the last command succeeded. If the mobile procedure had to explicitly check each of these results, it would quickly become unreadable. Instead requests and operators throw exceptions which halt execution, and by default, send an error message to the client with the offending major and minor tags. Mobile procedures may modify this behavior by catching the exceptions.

3.8 Security policies

The REAP toolkit allows the server application to set security policies for the mobile procedures uploaded by the clients. Currently, policies they can specify are still primitive, such as setting limits on overall execution time, memory usage, or the number of requests performed. If a mobile procedure violates any of these limits, it is halted and an uncatchable security exception is thrown.

4 Application of Toolkit

In this section, we report measurements comparing a modified IMAP server we have built using the REAP toolkit versus the standard University of Washington IMAP server [3] implementation. We will also report some measurements gathered from a REAP-enabled SMTP proxy that we have created. Finally, we will demonstrate how this toolkit can be applied to the LDAP protocol.

4.1 IMAP client synchronization

Each time an IMAP user logins into his/her account, their client verifies and updates its cached version of the user's account, a process we are calling client synchronization. Typically, the client checks the user's folder listing, the flags of previously seen messages, and downloads the relevant headers of new messages.

To demonstrate the benefits of the REAP toolkit, we have modified the UW IMAP server to use REAP and allow clients to upload mobile procedures to perform client synchronization. We then compared the latency of using REAP versus the latencies of both Netscape 4.74 and Outlook 2000's cache synchronization method when logging into the traditional UW IMAP server. We should point out that we modified the traditional UW IMAP server to use process pools to improve performance, instead of having `inetd` fork a new process for each incoming connection

4.1.1 Application to IMAP

Integrating the REAP toolkit into the UW IMAP server was fairly straight forward. The code used to implement the necessary IMAP requests and responses only amounted to 519 lines of C code. However, we only implemented the functionality needed to perform the synchronization. Additionally, the UW IMAP server was written in a very modular fashion, which made the integration much easier.

Figure 4 lists the mobile procedure used to perform the IMAP client synchronization, which performs all the same work as Netscape's method, as listed in Figure 2. The mobile procedure employs most of the advanced features in the REAP toolkit, including response iterators and response digests. A response iterator is used to invoke a `LSUB` request for each namespace return. In fact, the namespace response is not really needed by the client, so it is never returned which reduces the client response size.

Response digests are used to reduce the size of the folder and flag list returned by the server. We programmed the client to create folder list digests by segmenting the list evenly into 10 parts. For the flag list, the client created the digest such that finer granularity was given to newer messages. This approach reflects the idea that the flags of older messages are less likely to change, thus reducing the expected number of bytes sent in the response.

```
(defun sync-client (username password folder-digest inbox-digest last-msg)
  (label capabilities (capability-request ’a1))
  (send (login-request ’a2 username password))
  (label folder-listing nil)
  (if (request-contains "namespace" capabilities)
      (iter-response x (namespace-request ’a3)
         (union-responses folder-listing
                          (list-request ’a4 (car x) "*")))
      (label folder-listing (list-request ’a5 "" "*")))
  (label inbox-listing (progn (select-request ’a6 "INBOX")
                              (fetch-request ’a7 "1:*" ’(uid flags))))
  (send (digest-filter folder-listing folder-digest))
  (send (digest-filter inbox-listing inbox-digest))
  (send (fetch-request ’a8 (concat (+ last-msg 1) ":*") ’(uid envelope))))
```

Figure 4. IMAP synchronization mobile procedure

4.1.2 IMAP results

Figures 5 and 6 show the latencies of each of the three IMAP client synchronization methods (Netscape, Outlook, REAP mobile procedure) over various network links. As the figures indicate, we measured the latency when the client was connected to the server over 100 Mbps Ethernet, 10 Mbps Ethernet, WaveLAN (802.11 wireless Ethernet with 1 Mbps maximum bandwidth), a modem (38 Kbps), and Metricom’s Ricochet service (a radio packet network with 28.8 Kbps maximum bandwidth). Statistics for these networks are given in Table 2.

For these experiments, the IMAP server was run on a dual 500 MHz Pentium III machine with 256 MB physical memory running Linux 2.2.16. The client software was run on a 600 MHz Pentium III machine, with 192 MB physical memory running Linux 2.4.4. Each latency was computed by averaging ten runs of the synchronization method. The IMAP account contained 30 folders and 100 messages in its INBOX. Message arrival and folder/flag changes were simulated.

Figure 5 clearly illustrates that the REAP synchronization method takes significantly less time than Netscape or Outlook’s in high latency and low bandwidth network settings, which typically characterize mobile settings. In this figure, each group is normalized by the maximum latency for that network setting. As we can see, the REAP synchronization method’s latency is only 20% of Netscape’s when the client is using Metricom, and 13% when using a modem.

Figure 6 shows the actual magnitudes for the various latencies, plotted on a log scale. Examining the figure, we find that Netscape’s latency over Metricom is 16.0 seconds, while the REAP mobile procedure only takes 3.2. Similarly, over the modem connection, Netscape takes 3.5 seconds, compared to the mobile procedure’s .4 seconds. Clearly, the latency savings are very significant, especially in high latency settings.

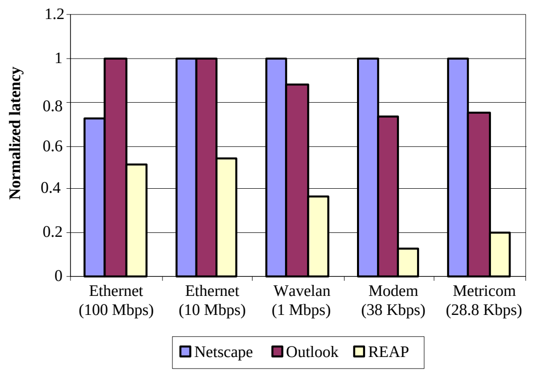

Figure 5. Normalized IMAP synchronization times

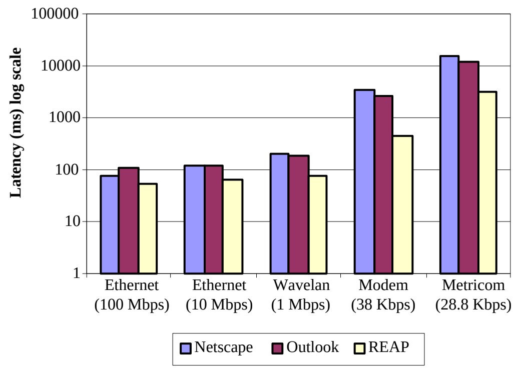

Figure 6. IMAP synchronization times (log scale)

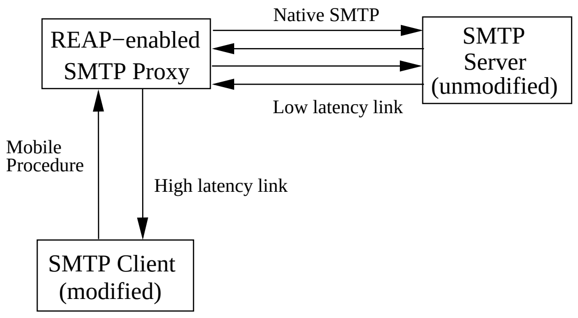

Figure 7. SMTP proxy architecture

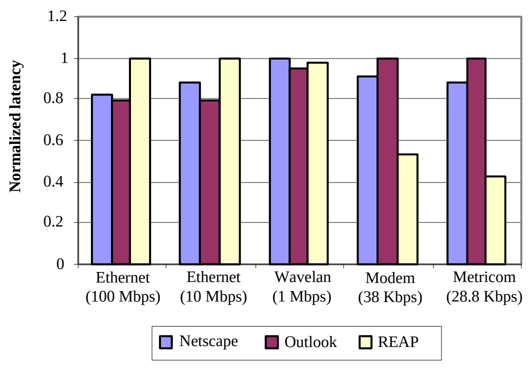

Figure 8. Normalized SMTP send times

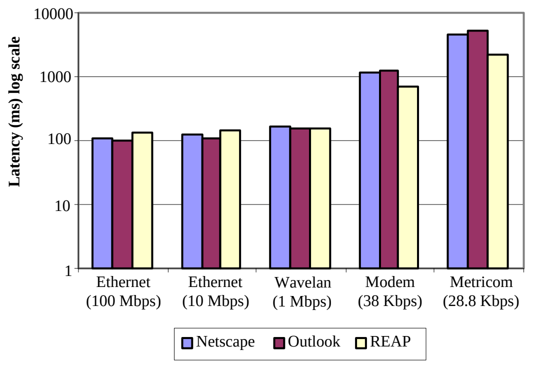

Figure 9. SMTP send times (log scale)

4.2 SMTP

The next protocol we investigated was the Simple Mail Transfer Protocol (SMTP) [17]. This is a fairly simple protocol, but we do see some performance improvements as we will show. Typical of other clients, Netscape 4.74's email client uses at least 6 blocking commands to send a single email, with an additional one for each recipient. These commands are issued to handshake with the SMTP server, specify the from address, recipient addresses, the data content, and then log off.

In order to test the lower bounds of performance improvements, we decided to create a REAP-enabled SMTP proxy instead of creating a REAP-enabled SMTP server. Figure 7 illustrates how this proxy allows modified clients to use the REAP protocol to connect to the proxy over high latency links, while the proxy uses native SMTP commands to connect to the target SMTP server over low latency links. This is a good test of the toolkit's ability to improve a protocol's performance because SMTP is simple enough that there is not as much room for improvement as in IMAP and because the proxy will introduce more overheads than a REAP modified SMTP server would.

Section A.2 gives the SMTP mobile procedure we used. We have decided to split the operation into two stages in order to avoid transmitting the email content unless it will be accepted for delivery. The first stage specifies the sender and recipient, and verifies the server will accept the e-mail. After the client receives confirmation, the second stage is used to send the data content. By splitting the operation into two stages, we are again hindering REAP's ability to improve performance so that we can gain an honest evaluation of the toolkit.

For our experiments, we measured the latency associated with sending a 1 KB email with one receipt via the SMTP proxy, while varying the client's network access method. We then compared this against the latency associated with sending the same email by connecting directly to the SMTP server using both Netscape 4.74 and Outlook 2000's methods for sending email. Netscape and Outlook's methods differ slightly in terms of when they perform the server reset command. Each given latency is the average of 10 trials. The proxy and SMTP servers ran on separate hosts, each being a dual 500 MHz Pentium III with 256 MB physical memory running Linux 2.2.16 connected via 100 Mbps Ethernet. The client software was run on the same machine as the IMAP tests.

Figures 8 and 9 show the results of the experiments. As Figure 8 illustrates, the SMTP proxy actually has higher latency when the client is connected via a fast network access method, such as 100 Mbps or 10 Mbps Ethernet. However, we see as the network latencies increase, the latency to send an email via the SMTP proxy decreases relative to the traditional SMTP methods, finally resulting in a 50% decrease when using Metricom. Specifically, when using a modem, the Outlook method takes 1.28 seconds compared to REAP's .68, and when using Metricom, Outlook takes 5.09 seconds compared to REAP's 2.19.

These results are interesting because they point out that in some circumstances, the client should not use REAP since the overheads associated with going through the proxy and running the REAP evaluation environment outweigh the possible savings. However, even with these overheads, the REAP toolkit still improves performance for a simple protocol such as SMTP in mobile environments. Furthermore, these savings increase for all network access methods as we add more recipients since both Outlook and Netscape add another command for each recipient. When using four recipients with a 1 KB email, Outlook takes 7.64 seconds, Netscape takes 6.21 seconds, while REAP takes only 2.07 seconds over Metricom, which is a 70% reduction in latency.

4.3 LDAP

To demonstrate its applicability beyond the IMAP and SMTP protocols, we are developing additional REAP modified serves for other protocols. Currently, we have not finished their development, but we wanted to discuss how we are applying REAP to at least one more protocol. The protocol we will discuss is the Light-weight Directory Access Protocol (LDAP) [26] which exports a simple a hierarchical database-like interface so that client applications may store complex attribute/value pairs. We targeted Netscape Communicator's use of LDAP to implement their Roaming Access feature, which allows users to save bookmarks and preferences on a central computer.

Netscape's Roaming Access stores each data item (the bookmark or preference file) as a single object on the LDAP server. Upon startup, Communicator binds to the appropriate LDAP database, compares the modification time of the bookmark and preference objects against the local client's modification times, and if they differ, downloads them. On exit, the client uploads modified files to the LDAP server. Excluding the uploading requests, we found that the Netscape 4.74 client issued 8 blocking requests to download the user's profile.

Section A.1 gives the mobile procedure we will use to implement Netscape's Roaming Access. There are only two LDAP requests needed: `bind-request` which performs authentication to a particular database, and `search-request` which performs directory searches. The arguments to the `search-request` are not relevant for our discussion, but specify such things as limits on search time, data elements, and filters on what is returned.

5 Future Work

Chief among the areas that still need to be explored, is to apply the toolkit to more application-level protocols. We are currently developing modified LDAP and SMTP servers, and will also investigate other protocols such as Palm Pilot synchronization, remote CVS access, and HTTP. By applying the REAP toolkit to these protocols, we hope to gain experience in its general usefulness and verify its feature set is useful.

To improve the REAP toolkit, we should also investigate better ways to support a protocol's wire format. In the current version, the REAP toolkit takes an ad-hoc approach to parsing incoming requests and writing the responses to the client. Each protocol has a different standard, and this needs to be reflected through the use of pluggable modules or some other solution.

The Simple Object Access Protocol (SOAP) [25] may represent one very good way to solve this problem. SOAP is an emerging XML standard that creates a common way to describe/format data and commands for exchange between loosely connected distributed components. Essentially, SOAP could become the base language for all new application-level protocols. This would be an ideal fit with the REAP toolkit – REAP would understand how to parse and manipulate base SOAP requests and responses, and then be able to forward them cleanly to the server application. By having this common language base for all protocols, middleware tools will be able to easily manipulate many diverse application-level protocols.

Dynamic adaptation and introspection by measurement is another area of potential research. Because of the simple remote evaluation environment, the REAP toolkit can easily measure a mobile procedure's bandwidth, CPU, and memory usage. Based on these statistics and client information, the REAP toolkit may be able to automatically decide where the mobile procedure should be executed and the amount of resources to dedicate to it. For example, if the server is overloaded, the toolkit may decide that all clients connecting via Ethernet should not be able to upload mobile procedures, while allowing mobile clients to do so. Similarly, this can be used to decide when to employ a local access proxies.

Finally, the lessons we have gather from the REAP toolkit may also be used to build tools for automatically measuring application-level protocol performance. We could use metrics such as the number of blocking requests and amount of redundant data issued by a server to analyze its performance. We can then use this information to identify troubled protocols and determine ways to improve them.

6 Related Work

Remote evaluation and code migration has been employed in other systems to reduce communication between hosts for efficiency. The REV project [20, 19] was an early effort to explore this area. This was a generic remote evaluation tool used to build distributed systems. The Java programming language [8] is another popular system that uses mobile code, though its mainly used to augment client functionality. SQL [5] can be viewed as a remote evaluation system, since it allows clients to skip intermediate results by submitting complex, nested queries to the database server.

Active networks [23, 28] uses mobile code to not only modify server functionality, but also the functionality of the entire network to aid the client application.

Collocation of computation with data has also been explored in many projects. Emerald [13] was a distributed object-based language that allowed for data objects to migrate across nodes to reduce communication. The Rover toolkit [12] enabled mobile applications to hide network latency and disconnectivity through relocatable dynamic objects and its queued remote procedure call mechanism.

The REAP toolkit is also a communication abstraction, hiding network programming complexities from the server. This is similar in concept to Remote Procedure Calls (RPC) [2] and Java Remote Method Invocation (RMI) [21]. There are several commercial packages that do this along with object management, such as CORBA [24] and DCOM [6].

Finally, there have been other projects that improve an application-level protocol's performance by compressing and removing redundant information, much like REAP's response digest mechanism. These tricks have been applied to HTTP traffic in [15], and to generic TCP/IP protocols in [18].

7 Conclusion

This paper has described the REAP toolkit, which allows server developers to integrate remote evaluation into their applications. Integrating remote evaluation into the application-level protocol used between the server and clients builds a "slop" factor into the protocol design. This allows clients to make more efficient use of the protocol, as we have shown through examples of commercially used protocols. The performance gains in terms of reduced latencies and bandwidth can be very dramatic for mobile clients, such as the 80% reduction we showed for IMAP client synchronization over Metricom, dropping the latency from 16.0 seconds to 3.2.

One of the main features of the REAP toolkit's design is that it employs simplicity. We have only included the most essential operators into our language, eliminating many potential areas of abuse. Even with our small operator set, we can still create complex mobile procedures that optimize existing protocols. Additionally, because of this simplicity, development with the toolkit is easy. The modified IMAP server we created only used 519 lines of C code to create the bridge between the existing IMAP server code base the and REAP toolkit.

The high-level request-response design used by the REAP toolkit also opens opportunities for more advanced optimizations. Some of these features have already been built into the toolkit, such as the response digests which performs response compression based on previously seen responses. Additionally, we are developing the toolkit to support dataflow computation models, which will allow for intermediate results to be processed incrementally, reducing the amount of memory needed to hold responses.

Overall, the REAP toolkit has already demonstrated it can significantly improve existing application-level protocols. However, the research is still just beginning, and we anticipate future versions will employ even more advanced tricks to further hide latency and low bandwidth conditions.

8 Acknowledgments

We would like to thank Albert Kwok for his work in creating the SMTP proxy and measuring its performance. We would also like to thank John Harper for creating and maintaining the open-source `librep` project. Finally, we thank the anonymous reviewers for their helpful feedback.

References

[1] David Bindel, Yan Chen, Patrick Eaton, Dennis Gees, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao, and John Kubiatowicz. OceanStore: An Extremely Wide-Area Storage System. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, November 2000.

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computing Systems*, 2(1):39–59, February 1984.

[3] M. Crispin. The University of Washington IMAP server. `http://www.washington.edu/imap/`.

[4] M. Crispin. Internet Message Access Protocol - Version 4rev1. RFC 2060, December 1996.

[5] C.J. Date. *A Guide to the SQL Standard*. Addison-Wesley, Reading, Massachusetts, 1987.

[6] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, Redmond, WA, 1998.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *1996 USENIX Security Symposium*, 1996.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[9] Robert S. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In *Proceedings of the Fourth Annual Usenix Tcl/Tk Workshop*, 1996.

[10] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, , and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Computer Networks*, 2000. Special Issue on Pervasive Computing.

[11] John Harper. The librep Project. `http://librep.sourceforge.net`.

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

[13] Eric Jul, Henry M. Levy, Norman C. Hutchinson, and Andrew P. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computing Systems*, 6(1):109–133, 1988.

[14] Metricom Corp. Ricochet Wireless Modem, 1998. `http://www.ricochet.net`.

[15] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential Benefits of Delta-encoding and Data Compression for HTTP. In *Proceedings of SIGCOMM 97*, pages 181–194, September 1997.

[16] George C. Necula. Proof-Carrying Code. In *The 24th Annual Symposium on Principles of Programming Languages*, Paris, France, January 1997. ACM.

[17] Jonathan B. Postel. Simple Mail Transfer Protocol. RFC 821, August 1982.

[18] Neil T. Spring and David Wetherall. A Protocol-Independent Technique for Eliminating Redundant Network Traffic. In *Proceedings of the conference on SIGCOMM 2000*, pages 87–95, 2000.

[19] James W. Stamos. *Remote Evaluation*. PhD thesis, MIT, Cambridge, Massachusetts, January 1986.

[20] James W. Stamos and David K. Gifford. Remote Evaluation. *ACM Transactions on Programming Language Systems*, 12(4):537–564, October 1990.

[21] Sun Microsystems. Java Remote Method Invocation—Distributed Computing for Java. `http://java.sun.com/`.

[22] Joseph Tardo and Luis Valente. Mobile Agent Security and Telescript. In *Proceedings of the 41st International Conference of the IEEE Computer Society (CompCon '96)*, February 1996.

[23] David L. Tennenhouse and David J. Wetherall. Towards an Active Network Architecture. In *ACM SIGCOMM '96 (Computer Communications Review)*. ACM, 1996.

[24] The Object Management Group (OMG). The Common Object Request Broker Architecture. `http://www.corba.org`.

[25] W3C. Simple Object Access Protocol (SOAP) 1.1 . `http://www.w3.org/TR/SOAP/`.

[26] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251, December 1997.

[27] D. Weinreb and D. Moon. *Lisp Machine Manual*. MIT Artifical Intelligence Laboratory, Cambridge, Massachusetts, 1981.

[28] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of IEEE OPENARCH'98*, San Francisco, CA, April 1998.

A Example Code

A.1 LDAP Roaming mobile procedure

```
(defun roaming-user (profile-name owner password roaming-base
                     bookmarks-timestamp prefs-timestamp)
   (send (bind-request ’1 owner password))
   (if (not (response-contains ’2 owner
                                (search-request ’3 roaming-base
                                                 0 0 0 0 true "" ’all)))
       (throw-exception ’3 "roaming profile not present"))
   (label check-prefs
          (search-request ’4 (concat "nsLIElementtype=liprefs, "
                                     romaining-base)
                           0 0 0 0 true "" ’(modifyTimeStamp)))
   (if (ldap-time-greater prefs-timestamp
                          (attribute-value ’modifyTimeStamp check-prefs))
       (send (search-request ’4 (concat "nsLIElementtype=liprefs, "
                                     romaining-base)
                           0 0 0 0 true "" ’(modifyTimeStamp))))
   (label check-bookmarks
          (search-request ’5 (concat "nsLIElementtype=bookmarks, "
                                     romaining-base)
                           0 0 0 0 true "" ’(modifyTimeStamp)))
   (if (ldap-time-greater bookmarks-timestamp
                          (attribute-value ’modifyTimeStamp check-bookmarks))
       (send (search-request ’6 (concat "nsLIElementtype=bookmarks, "
                                     romaining-base)
                           0 0 0 0 true "" ’(modifyTimeStamp)))))
```

A.2 SMTP mobile procedure

```
(defun setup-send-mail (username domain to-list)
   (ehlo-request ’1 domain)
   (from-request ’2 username)
   (iter-argument x to-list (rcpt-request ’3 x))


(defun send-mail (data)
   (data-request ’1 data)
   (send (create-response ’2 ’ok "message queued")))
   (quit-request ’3))
```