Introduction

The Rover Toolkit was developed by Anthony D. Joseph, et. al. to isolate mobile applications from the limitations of mobile communication systems [1]. The Rover model is a check-in/check-out system for manipulating Relocatable Dynamic Objects (RDOs). A Rover client makes a Queued Remote Procedure Call (QRPC) to import an object, manipulates the object, and then exports the object back to the Rover server. QRPC permits applications to make non-blocking remote procedure callls when the client is disconnected from a Rover server. These requests are fulfilled when the client is reconnected and has access to a Rover server.

As can be seen from Figure 1, what allows the user to perform a significant amount of work while off-line is that the Rover Toolkit provides an Access Manager that caches imported objects and allows import requests for these objects to be satisfied directly from the cache. Therefore, the issue becomes how one can package the objects such that as few imports as possible need to be made. Once the objects are in the cache, the application itself can manipulate and/or extract various parts of the data, and then export any modifications of the object, if necessary. If objects are not in the cache, then they are requested from the Rover server. If the server cannot be accessed, then the request is queued until it can be fulfilled. Meanwhile, the application keeps on running. Thus, the user doesn't see the latencies of the network (or lack thereof). He is just able to keep on working while the QRPC attempts to fulfill its request. The Access Manager is transparent to the Rover application developer, who simply issues commands to import and export objects, although the developer may specify whether an object can be cached or not.

The treatment that follows gives an example of how Rover is applied to allow the use of Hypertext Transfer Protocol (HTTP) over an intermittently connected network. Since Rover itself uses HTTP to communicate between the client and the server, this is a rather natural extension, and treating the Universal Resource Locators (URLs) accessed by the World Wide Web (WWW) browser as Rover objects cached on the client is rather straightforward. However, there are differences between the HTTP and Network News Transfer Protocol (NNTP) that do not allow NNTP to be supported as easily using the Rover toolkit as HTTP. To get a better idea of how NNTP will be supported by Rover, a typical session between an NNTP client and NNTP server is examined. Based on this, one can get a better idea about how the naming scheme in Rover must be handled and what general rules for simplicity and coherence will be adhered to in the implementation. The goal in the implementation is to allow NNTP browsing with disconnected/intermittent connectivity. Rover NNTP itself contains several different components, each of which are examined in detail. The current implementation only supports reading of news and not posting. However, possibilities for future extensions to Rover NNTP, including adding posting capabilities, are considered later. Rover NNTP provides a consistent scheme for manipulating Rover objects to produce NNTP responses based on unmodified NNTP commands issued from a client.

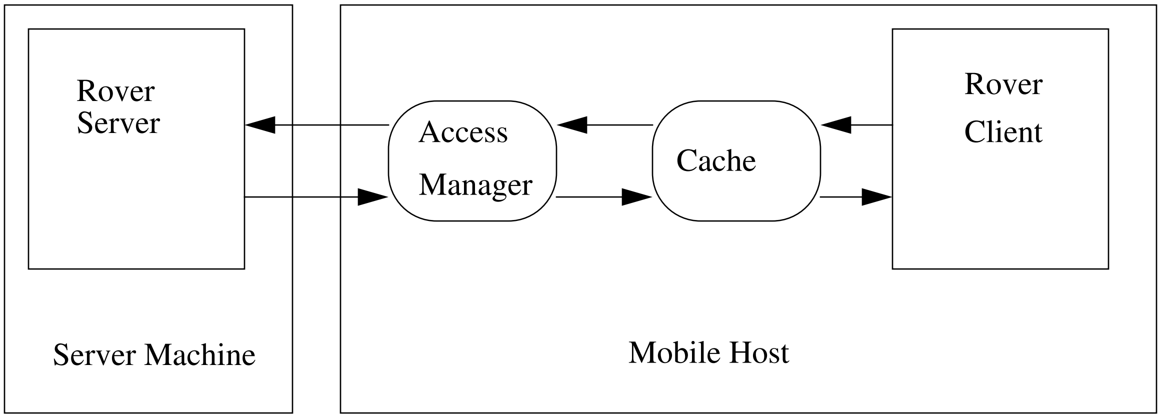

Figure 1. The Rover Model

Example Implementation of a Network Protocol in Rover: HTTP

The Rover Toolkit has been used, for example, to allow web browsers to treat the Rover client as an HTTP proxy for browsing the WWW. The Rover HTTP proxy treats each URL as an object identifier. After a request for an object identified by its URL is made and sent by the Access Manager to the Rover server, the Rover server connects to the HTTP server specified by the URL and retrieves the document. The document is then saved by the Access Manager and loaded onto the client. It can then be retrieved from the cache by the access manager when future repeat requests for this URL are made. The user may keep on working while the object is being retrieved -- note that in some cases, the QRPC may stay unfulfilled for a while (in the case that the Mobile Host is disconnected from the network). Rover HTTP was the first click-ahead web browser proxy that was implemented without any client changes.

The HTTP protocol is easily implemented using Rover for several reasons. First, all URLs on the web refer to a unique document. Therefore, since objects are named in accordance with their URL, a request never retrieves an incorrect document, whether it is in the cache or the server, unless the maintainer of the document has changed it. Second, the HTTP model consists of opening a network connection, retrieving the needed information, and then dropping the connection, a "connectionless, stateless protocol." All information needed by the HTTP server is contained within the URL itself, and thus the object name. This corresponds to Rover requests for objects from the Rover server (which is why it uses the HTTP protocol itself).

How NNTP Differs

While Rover lends itself to implementing basic network protocols to act as a proxy for various network applications, NNTP has inherent differences from HTTP, raising problems when attempting to "Roverize" the communication between the NNTP client and NNTP server. Commands and responses from each end are not modified, but the way the requests are satisfied is tied in with interactions between the Rover client and Rover server. The differences between the protocols complicates these interactions in the case of NNTP. While the HTTP model is "connect-retrieve-disconnect", NNTP depends on opening a connection to an NNTP server and initiating an NNTP "session." During this session, the NNTP client issues commands that place the session in a certain state. Therefore, to ensure that the NNTP proxy issues the correct response for a given command, there must be a way for the Rover client to keep track of the current "state" so the Rover server can initialize itself into this state when it makes a new connection to the NNTP server and retrieves the needed data.

It should be noted that most web browsers support some form of news reading capability by making NNTP requests using connections that open, receive data, and drop. Especially since the Rover toolkit's communication between the client and server is implemented using HTTP, writing a news reader that acts in the same way as a web-browser based newsreader may be more appropriate for implementing NNTP with Rover. Though that is beyond the scope of this project, it certainly deserves further attention in the future.

A Typical NNTP Session

A useful guide when deciding on the general design of Rover NNTP was an examination of a news reading session using a typical newsreader, in this case xrn 8.00, written by Ellen M. Sentovich and Rick L. Spickelmier. This allowed me to better decide what form the objects should take to ensure acceptable performance. Objects must be general enough that several commands will be able to satisfied with the import of a single object. At the same time, they should be small enough that the time it takes to do an import over a slow dialup line will not be too great.

```
MODE READER
LIST
GROUP ne.forsale
XHDR subject 57605-57605
XHDR subject 57607-57611
XHDR subject 57612-57613
XHDR subject 57615-57622
XHDR subject 57623-57623
XHDR subject 57625-57627
XHDR subject 57629-57639
XHDR subject 57641-57645
XHDR subject 57646-57665
XHDR subject 57666-57667
XHDR subject 57669-57688
XHDR subject 57690-57707
XHDR subject 57708-57709
XHDR subject 57711-57711
XHDR subject 57713-57713
XHDR subject 57715-57725
XHDR subject 57727-57727
XHDR from 57605-57605
XHDR from 57607-57613
XHDR from 57615-57623
XHDR from 57625-57627
XHDR from 57629-57639
XHDR from 57641-57667
XHDR from 57669-57688
XHDR from 57690-57709
XHDR from 57711-57711
XHDR from 57713-57713
XHDR from 57715-57725
XHDR from 57727-57727

. . .

GROUP ne.forsale
ARTICLE 57899
XHDR xref 57899
ARTICLE 57897
XHDR xref 57897
QUIT
```

Figure 2. A sample of an NNTP Session

To get an idea of what kind of objects would need to be created, a simple proxy for reading articles was created that read in commands from xrn, saved the commands in a file, passed the command onto the NNTP server, and relayed the server's response back to the newsreader. After an NNTP session was over, the file was closed and could be inspected to see all the commands that were issued. As can be seen from Figure 2, a typical session from xrn makes several separate requests from the NNTP server for "XHDR" fields, which are the separate fields in the opening lines of a news article (the article subject, the author, etc.). Since these requests come in sequence at the rate of several in a row, seem to request most of the X-Headers from the newsgroup, and are relatively small in size for each request (each single header is no more than one line), it would probably be advantageous to simply make objects involving X-Headers possess a rather large granularity -- in this case, a Rover NNTP object for an XHDR field of a single group contains all the header fields for that group (or, at least, a significant fraction of them).

On the other hand, articles are requested one at a time and are comparatively large-- each article probably has at least 20 lines or so. Thus, the granularity of a Rover NNTP object dealing with articles should be much finer grain and will certainly not contain all of the articles for the group.

Thus, a simple perusal of a typical NNTP session gives us some clues as to how to efficiently encapsulate information for the Rover NNTP proxy.

Rover and NNTP

A basic implementation that allows one to read netnews using the Rover toolkit involves careful consideration as to how to name the Rover objects. The choice of the naming scheme for these objects was driven by two important guidelines: (1) the name of the object had to indicate the state that the NNTP session had to be in before data could be retrieved, and (2) the name had to contain a representation of the actual NNTP command issued so as to simplify the implementation of NNTP commands in the Rover server. These choices allow the Rover server to parse names to quickly determine how to set up the appropriate state and ask for the requested information. This process was meant to return data unprocessed, leaving the Rover client to place the data in the most efficient possible form, depending on the application needs.

Implementation

Overview

The Rover NNTP implementation contains several layers of communication in each component. At each end are the unmodified newsreading components -- the newsreader (in this case, xrn version 8.0) and the news server (senator-bedfellow.mit.edu). In between are various go-betweens that allow the exchange of netnews to be supported through the Rover toolkit.

The Rover NNTP client receives commands from the newsreader, but since the NNTP port is designated as port 119, which can only be used by a user running as “root,” there is a simple setuid root proxy that relays newsreader requests directed to port 119 of the local host to port 7777. Port 7777 is the port on which the Rover NNTP client listens for requests from the newsreader. The Rover NNTP client, upon receiving these commands, decides which object should be imported from the access manager. The access manager then retrieves the object either from the cache or the Rover server.

The Rover NNTP server handles NNTP requests for objects by connecting to the NNTP server, using the name of the object to initialize the connection state (as though a connection-based session had been in effect), and issing the desired command to retrieve the requested data. This data is then sent back to the Rover NNTP client which extracts the needed response from the object. The response is then sent through the proxy to the newreader. The proxy then waits for the next command from the newsreader.

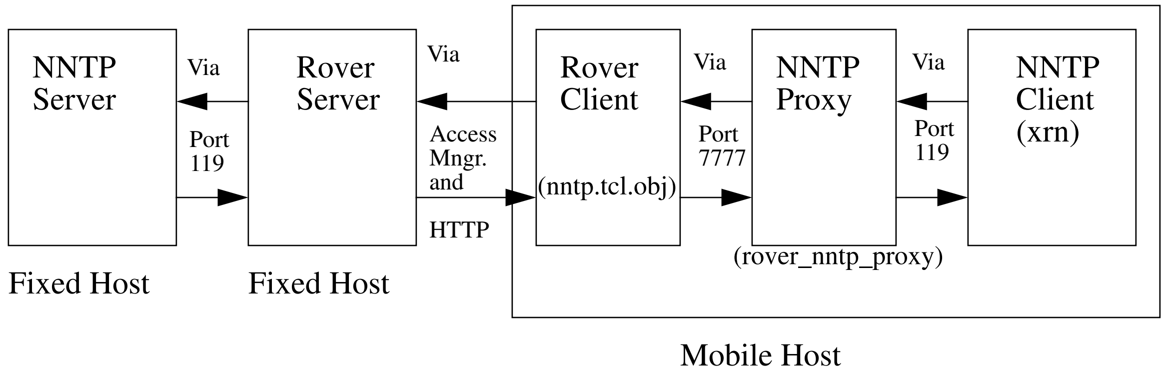

Figure 3. The communication paths of Rover NNTP

The Simple Port-to-Port Proxy

Some newsreaders, specifically xrn, only issue network commands to port 119 of the NNTP Server, without giving the user the option of changing this value. At the same time, any application listening on port 119 must be running as root to bind to that port. This presents a problem because the Rover toolkit implicitly assumes a model in which a user starts an application that runs under his or her username on the client machine. Therefore, it became necessary to have a simple proxy daemon residing on the local machine that would listen on port 119 and transmit its data to the port on which the Rover NNTP client program was listening, port 7777. The use of the proxy thus allows us to use the newsreader without making any changes.

The actual implementation of the proxy is simple, although this has performance costs. The implementation was one that I felt was the easiest to visualize-- the proxy would receive a command, send that command to port 7777 of the local host, and wait for a response from port 7777, which it would then relay back to port 119. Since newsreaders occasionally issue many commands at a time and wait to receive them in order, it was necessary for the proxy to receive one line at a time from the newsreader, issue it as the command to the Rover client, and not get the next line-- i.e., the next command-- from the newsreader until after the newsreader had received all the response data from the Rover client. To receive one line from the newsreader, a read() command must be called for each character until it encounters a newline. An alternative, more efficient (albeit more complicated) approach would be to tag requests and match responses.

The proxy contains two main sections: the main procedure that waits for connections to port 119 and a procedure that reads commands from the NNTP client and executes them until receiving a "QUIT" command from the client.

The proxy procedure, called `start_proxy` takes a socket stream as an argument and connects to the NNTP server (or, in this case, the Rover client, which acts as an NNTP server) before it runs an infinite for loop that reads lines off the input, ignoring blank lines. Each non-blank line is then checked to see if it is a "QUIT" command, in which case it closes the connection with the NNTP client and server, returning control to the main procedure. Lines are read from the NNTP client by way of a procedure `read_line` which reads one character at a time from port 119 until encountering a newline. Unfortunately, this is quite inefficient and results in a performance delay. However, it is sufficient for the current model of allowing a user to read news transferred via NNTP. Once the command from the NNTP client has been sent through the proxy to the Rover client, the proxy waits for a response. The proxy enters a loop in which it receives data in lengths of 256 bytes at a time until it encounters a "." character on a single line. At each stage in the loop, the data read from the Rover client is sent to the NNTP client. Ideally, it is possible to have just one 256-byte buffer for both sending and receiving. However, for simplicity's sake, there are separate buffers each for sending and receiving for both the NNTP client and the proxy's server (which is, as noted above, the Rover client).

The NNTP proxy is a simple solution that allows the newreader to continue to communicate on the NNTP port 119 without resorting to either modifying the newsreader or running the Rover application as root. Despite its performance penalties, it provides a simple solution to the problem.

The Tcl Socket Libraries

The Rover NNTP client application was written entirely in Tcl. However, since it was necessary for the application to receive commands from the newsreader (or, in this case, the newsreader proxy), Tcl commands, backed by C code, needed to be created to allow this communication. Each procedure used the standard Berkeley sockets in C and supported the following operations:

`Cmd_open_server_connection` - Argument: Port Number

Returns: Integer (TCL_OK or TCL_ERROR)

Tcl Result: Socket Stream ID

The invoking Tcl application binds to the port passed to it and listens for connections. The Tcl application returns the socket stream id resulting from the application's binding to the port.

`Cmd_close_socket` - Argument: Socket Stream ID

Returns: Integer (TCL_OK or TCL_ERROR)

Tcl Result: If error, then a string indicating the cause. Otherwise, none.

This procedure closes the connection to the stream via C's close(sockfd) command.

`Cmd_connect_to_client` - Arguments: Socket Stream ID and an initial connect message

Returns: Integer (TCL_OK or TCL_ERROR)

Tcl Result: Socket Stream Id for the session if successful.

This procedure waits for a connection from a client and returns a socket stream id when it receives a connection. When a client successfully connects, it receives the initial connect message specified by the second argument as a status line indicating a successful connection.

`Cmd_accept_data` - Argument: Socket Stream ID of the current session

Returns: Integer (TCL_OK or TCL_ERROR)

Tcl Result: A string of up to 256 characters in length.

After a connection has been successfully established, this procedure is invoked to listen to the socket identifier until it receives data. Since the expected model in Rover NNTP is that the Rover client will be receiving line-by-line NNTP commands from a newsreader, this procedure reads in 256 bytes and returns the data to the caller. Note that in the future, if Rover NNTP was required to receive multi-line information from a newsreader (such as posting an article), then this procedure could still be used, with the requirement that the Tcl code for Rover NNTP would have to keep reading in data until encountering a "." on a line by itself. Blank lines are handled as strings of length 0 passed back to the caller.

`Cmd_send_data`: Arguments: Socket stream ID and a string of data to be sent

Returns: Integer (TCL_OK or TCL_ERROR)

Tcl Result: The number of bytes sent if successful, or an error message if the execution fails.

The `Cmd_send_data` command takes a string of arbitrary length, places it in a send buffer and sends the data to the given socket stream via the “write” command.

These C procedures are initialized into Tcl commands when Rover NNTP is started. The Rover NNTP application itself is downloaded and executed by the Access manager, and the application creates a global Tcl interpreter when it starts executing. After it creates the necessary C functions for use in Tcl, it imports the Rover client program, nntp.tcl.obj.

The Rover NNTP Client

The code for the Rover NNTP client defines all the data classes used to generate the NNTP responses to the newsreader, reads in NNTP commands, generates objects and extracts the necessary data from them, and sends the responses back to the newsreader. The main part of the Rover NNTP client code contains a loop which listens on port 7777 for NNTP commands then imports objects and sends out a response before waiting for the next command. The Rover NNTP client uses the generated Tcl commands mentioned above to communicate with the newsreader.

When the Rover NNTP client starts up, it initializes the environment, binds to port 7777 using the C procedure Cmd_open_server_connection, and enters an infinite loop in which it listens for connects from a client program. Once a connection is received, a mock NNTP session begins and the Rover NNTP client enters another loop which reads one-line commands from port 7777 and continues until receiving a “quit” command. After receiving each command from the proxy, it is compared against a set of reading-related NNTP commands. Once a match is found, an object name is created based on the command and current state and an import request is made for the object. After the object is imported, the NNTP command issued to the Rover NNTP client invokes a method on the object (usually “unmarshall”) to yield the desired output, and that output is sent through port 7777 to the listening client. If no match is found, then an error is sent back as output.

Objects are named based on the command. All Rover NNTP object names begin with "Rover__Nntp__". What follows is a class name (such as "Article" or "AList" or "HeaderLines"), then a command (such as "NEXT" or "ARTICLE@1523"), and finally the current state, which is given by the current group and article number (e.g. "athena.forsale@1523"). The "@" symbols are used because whitespace characters are not allowed in object names and the "@" is not valid in NNTP news group names. The "@" characters are replaced by whitespace when the various fields are parsed out by the Rover NNTP server.

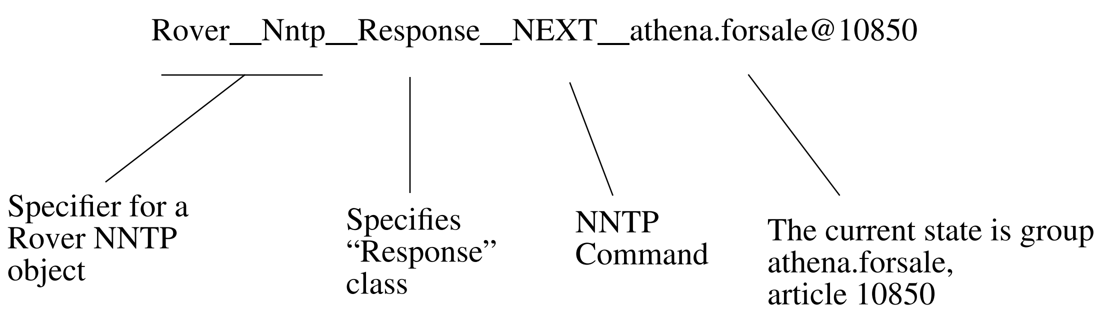

Figure 4. The fields of a Rover NNTP Object name

As an example, assume that a user begins an NNTP session. The first command issued places the system in "Reader Mode." Rover NNTP does not support any other modes, as normal newsreaders do not make use any other modes. Therefore, the command "MODE READER" instantly returns a response indicating "success" to the newsreader. Then assume the user requests the group athena.forsale. The Rover NNTP client matches the "GROUP" command and constructs the object name "Rover__Nntp__Response__GROUP@athena.forsale__none". The beginning name indicates that it is a Rover NNTP object, the "Response" indicates that the Rover

NNTP client is asking for a “Response” object from the server, “GROUP@athena.forsale” indicates the command to be executed (so the Rover NNTP server can just grab this field, parse out the “@” and execute it directly), and the trailing “none” indicates that the Rover server does not have to initialize itself to a given state before executing the command. When “group” is executed, the Rover NNTP client makes sure to remember this group for subsequent commands. The response from this command is sent from the Rover NNTP server to the Rover NNTP client after the import is complete, and the Rover NNTP client gets the relevant data from the object and sends it to the newsreader before waiting for the next command. Now assume that the user requests article 1000 from the newsgroup athena.forsale (which he or she has already selected). The Rover NNTP client then constructs the object Rover__Nntp__Article___ARTICLE@1000__athena.forsale@1000. The Rover NNTP server opens an NNTP connection, knows to go to the athena.forsale group, and executes the “ARTICLE 1000” command, whose output is sent to the Rover client. This process continues until a “QUIT” command is received.

A clear demonstration of the use of an NNTP command to manipulate a Rover NNTP object is the “xhdr” command. In a typical newsreading session, “xhdr” commands are issued several times in a row, referring to the same group and the same field (“subject”, “from”, “lines”, etc.). The object imported by “xhdr” contains all the relevant data for the given newsgroup’s field (i.e., Rover__Nntp__xhdr@subject__athena.forsale@0000 contains all subject lines for the group athena.forsale). If the last object to be imported was the object specified by this command, then the object itself is still in memory, and the Rover client does not make an import. Rather, it just extracts the requested data from the existing object. For example, if the user is in the group athena.forsale and requests “xhdr from 1000”, then the client will import an object which contains all "from" headers from the group athena.forsale and just return the header for article 1000 to the receiver connected on port 7777. If the user requests the next subject header-- "xhdr subject 1001" -- then the Rover NNTP client will remember that the last xhdr command was issued from the same group and same field as the current one, and thus will get the "from" header for article 1001 from the object in memory and send it back to the user/newsreader without making an import. This optimization was necessary because the performance overhead for performing several of these imports was far too severe, even when the objects were already present in the access manager's cache.

Rover NNTP objects have a specific class associated with them. These classes are usually relatively simple. However, where possible, their structure was used to improve the efficiency of the Rover NNTP session and to allow room for extensibility. Each class along with its purpose and implementation is described below.

The classes are "Headerlines", which refers to a list of headers in a news article (produced by the "xhdr" NNTP command), "Groups", which is a collection of newsgroups, "Article", which is a new article, "Response", which refers to a one-line NNTP response (such as that produced by a "GROUP" NNTP command), and "AList", which handles the output produced by the "LIST" command.

In the current implementation, all of these classes simply hold the response information from the execution of a single command. However, possible future extensions to the classes can be based on the implementation of the "HeaderLines" class, which is meant to pre-fetch a large number of article headers once and just extract the relevant data. At the same time, the entire object could be cached for future requests.

When a HeaderLines class is created, it takes as an argument the raw data from a (presumably long) “xhdr” command. This request and reponse will be similar in format to Figure 5.:

```
xhdr subject 10822-10830
221 subject fields follow
10822 book sale
10823 WTB: Graduation tickets!
10824 COMPLETE MULTIMEDIA PC FOR SALE $490
10825 chevy spectrum 89 FOR SALE
10826 Kenwood Stereo
10827 DataProducts Laser Printer FOR SALE CHEAP
10828 WTB: Graduation Tickets
10829 Mac for sale
10830 <<< Moving Sale >>>
.
```

Figure 5. A sample issue of the “xhdr” command and response

The HeaderLines object expects to receive everything from “221 subject fields follow” onwards. When it receives this data, the creation procedure first turns all the data into ASCII to ensure that only regular text is accepted, and then strips off the first response line “221 subject fields follow” and places it in a slot reserved for itself within the HeaderLines class. The rest of the header lines are then parsed into a large list containing a set of sublists, each of which has two elements, the article number and the actual header line, as depicted in figure 6.

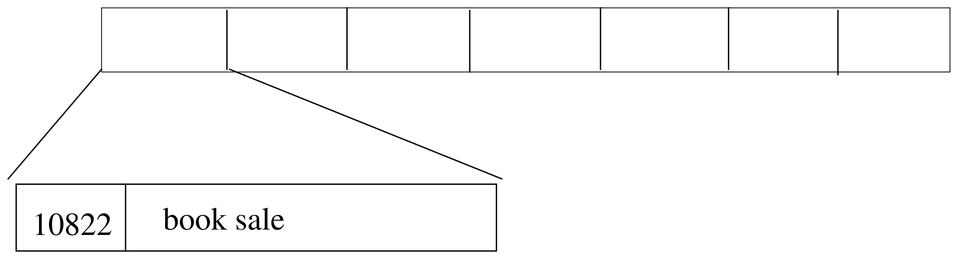

Figure 6. A graphical representation of the linelist slot of the HeaderList object

After the object is imported, an NNTP response is generated by the Rover NNTP client. This is done by executing the “unmarshall” method for HeaderLines, which takes as an argument the “start” and “finish” of the list of headers that are required. The unmarshall procedure then finds the start and finish indices of the articles in the list and returns a proper NNTP response, beginning with the response line and ending with a “.” on a line by itself. Thus, several different NNTP responses can be generated from a single object, depending on what command is issued (in this case, the range of the articles whose subject headers are requested).

For future work, this model should be the general one to be followed -- constructing an object containing different data where each piece can be recalled via a single NNTP command that invokes a method on the object. Since the newsreader typically requests all the relevant header lines at once, it was necessary to create an object containing all the header lines and their article numbers for each field. On the other hand, compared to the number of “xhdr” commands issued, the number of commands for “article” or “group” are comparatively few, and a straight-forward implementation of their objects by storing them as raw data for each unique command has been made without sacrificing performance. Of course, a more reasonable implementation of the “Rover model” where the user keeps several frequently-used objects on a mobile computer will include support for these “multi-part” objects that will be cached and not require an additional import from the Rover server for every new command issued by the newsreader.

The Rover NNTP Server

The Rover NNTP server accepts import requests for objects and sends NNTP commands to the NNTP server based on the object name. To accomplish this, the Rover NNTP server parses the object name to both the command or set of commands to execute and to determine the “state” that the session should be in with the NNTP server before executing the command. Once the Rover NNTP server places the NNTP server in the appropriate state, it executes the command specified by the object name, and sends code to the Rover client to create this object along with the raw data (i.e., the response from the NNTP server) associated with the request.

Since the NNTP command and state that the NNTP server should be in are encapsulated in the object name, the Rover NNTP Server has a set of procedures to parse out the necessary information from the object name, based on the format described in Figure 4. When the Rover NNTP Server receives an import request for this object, the nntp_import procedure receives the object name. From this, the Rover NNTP Server then parses the necessary data, storing the class type and the NNTP command as strings named “class” and “cmd”, respectively, and the state in a string named “misc.” Since the “cmd” and “misc” strings can contain “@” characters, which are used to replace spaces, both these strings are scanned to replace these characters with whitespace. The name thus provides all the information necessary for the server to return a uniquely specified set of data back to the client in the object.

Once the Rover NNTP server has all the necessary information, it sets up the session state. It opens up a socket connection to the NNTP server and first sends the server a "MODE READER" command, which specifies that the NNTP client is a newsreader (the default for most standard NNTP clients). The Rover NNTP server then takes the newsgroup name from the "misc" string and sends the NNTP server a "GROUP" command specifying the newsgroup. If an article was specified in the state (other than "0000"), the server sends a "STAT" command to have the current state point to that article before executing any more commands. Otherwise, the Rover NNTP server defaults to the first article in the newsgroup, which is given by the response to the "GROUP" command. After all of these commands are complete, the NNTP server is in an acceptable state, and the Rover NNTP server can begin executing its command or group of commands to obtain the data for the object.

In most cases, the command field of the Rover NNTP object needs no additional parsing before it is simply sent to the NNTP server. However, in the case of an "XHDR" command, the command in the object name is simply given as "XHDR subject" with no additional article numbers specified. In the current implmementation, the Rover NNTP server takes the first and last articles returned by the "GROUP" command and uses these values to execute an "XHDR" command over all the articles in the current group. For example, when selecting the group "athena.forsale", the NNTP Server will return this response:

```
GROUP athena.forsale
211 128 10830 10957 athena.forsale
```

The last two numbers refer to the first and last articles in the group. Thus, the rover server takes a an import request for an object named

"Rover__Nntp__HeaderLines__XHDR@subject__athena.forsale@0000" and, after setting up

```
if (!(strcmp(class, "Groups")))
   outstrC("\nreturn \"Groups-with-name mytest_groups\"");
 else if (!(strcmp(class, "Article")))
   outstrC("\nreturn \"Article-with-name mytest_article\"");
 else if (!(strcmp(class, "Response")))
   outstrC("\nreturn \"Response-with-name mytest_response\"");
 else if (!(strcmp(class, "HeaderLines")))
   outstrC("\nreturn \"HeaderLines-with-name mytest_headerlines\"");
 else if (!(strcmp(class, "AList")))
   outstrC("\nreturn \"AList-with-name mytest_alist\"");
 else {
   fprintf(stderr, "Incorrect class field.\n");
   exit(1);
 }
```

Figure 7. The Rover NNTP Server sends code to create a new Tcl object to the Rover Client

the state, executes the command “XHDR subject 10830-10957”.

In any case, once the Rover NNTP server has sent the necessary command for the object import request, it sends Tcl code to the Rover client telling it to create a new object of the appropriate class. It then waits to receive the data from the NNTP server. In many cases, the response from the NNTP server can be of arbitrarily long length (the typical response of a “LIST” command can be about 400K in length). Therefore, it was most efficient in terms of memory and speed to read in a block of data and then send it back to the Rover client, block by block, until all the data is read and sent, as shown in Figure 8. The Rover NNTP client will interpret these blocks as one continuous stream of data. NNTP responses take two forms -- one form is a multi-line response whose end is recognized by a “.” on a line by itself. These responses are typical for ARTICLE, LIST, and XHDR commands. On the other hand, the other form is simply a one-line acknowledgement. These types of responses are typical for GROUP, STAT, DATE, LAST, MODE, and NEXT commands. In the receive-send loop between the NNTP server and Rover Client, a 256-byte block of data is read in from the NNTP server and then the data is sent to the Rover server as the data argument to the object creator. The loop stops when the last five characters indicate a "." on a line by itself or if the command that produced the reponse is one which only gives one-line resposnes.

```
do {

  fprintf(stderr, "about to get bytes\n");
  n = read(sockfd, recvbuff, 256);
  p = p + n;
  fprintf(stderr, "got bytes\n");
  recvbuff[n] = '\0'; /* set the last byte to null */
  fprintf(stderr, "about to emit data\n");
  outstrD(recvbuff);
  if ((p <=256) && (recvbuff[0] == 4) || recvbuff[0] ==5)) break;
   /* return an error response */
  fprintf(stderr, "n is %d\n", n);
  fprintf(stderr, "last 10 bytes: |%s|\n", recvbuff+(n-10));
} while ((strcmp(recvbuff+(n-5), "\r\n.\r\n")) &&
         (
          (strcmp(cmd_name,"DATE")) && strcmp(cmd_name, "LAST") &&
          strcmp(cmd_name, "NEXT") && strcmp(cmd_name, "MODE") &&
          strcmp(cmd_name, "STAT") && strcmp(cmd_name, "GROUP")
          ));
```

Figure 8. The receive-send loop of the Rover server

The Rover NNTP server is thus able to create the appropriate object with the needed data for the Rover client based on the name of the Rover object requested for import. Each field of the Rover object name contains the relevant information needed to set up the NNTP server in the appropriate state, issue an NNTP command, and create the data for an object needed by the Rover NNTP Client. Finally, the Rover NNTP server reads in the NNTP response and returns the data to the Rover client to create the requested object.

Further Work

This project leaves many avenues for future extensions. At this point, only requests for X-Headers (via the “XHDR” command) can be fulfilled from a single object with a wide range of different articles within that group. This model should be extended to articles, where requests for different articles could be fulfilled using a single object. Requests that could not be fulfilled, could be by making a request from the server to add additional infomation to the specified object.

Furthermore, some method of checking the current status of objects needs to be considered. A standard needs to be developed to indicate to the Rover server when the client’s copy of the object is out-of-date. The object associated with the “LIST” command (which contains all the newsgroups on the NNTP server) changes slowly and does not need to be re-created every time (since it has a size of more than 400k). The “NEWGROUPS” command could be used to detect changes in the current list of newsgroups, based upon the date when the object was imported or when the .newsrc file was last modified. Full “LIST” commands could be used sparingly, perhaps only when the mobile client is in close proximity to the Rover server.

Finally, this implementation of Rover NNTP only supports the reading of news and the importing of Rover NNTP objects. In the future, full support of the NNTP protocol should be implemented and would include support for news posts and mail replies.

Another possibility that needs to be explored is the creation or modification of a newsreader to support Rover, specifically, instead of a transparent NNTP proxy meant to interact with Rover. Several WWW browsers support a connectionless interface to news servers that more closely models the way that Rover is used. It could be that this may be a better model for supporting NNTP with Rover because the newsreader would be able to specify (or at least be aware of) the underlying structure of the objects constructed by the Rover client. The newsreader could then tailor its requests for objects to be ones which take the most advantage of Rover's caching of objects. The functionality for constructing "usable" objects cached on the local machine would thus be moved to the client newsreader.

Conclusion

The Rover toolkit can be used to read news by representing NNTP commands as objects. The data needed for a single command can be extracted and returned to a newsreader. Performance declines significantly because of the many layers of communication that have to be traversed between the newsreader and the NNTP server. However, once a significant number of frequently-used objects are cached on the local Rover Client, this penalty is reduced. The Rover Client's features, including a set of socket libraries for use in Tcl provide a simple network interface, and the port-to-port proxy running on the client allow an unmodified newsreader to run on the client machine. Finally, the Rover server is implemented such that it is simple enough so that it can issue NNTP commands based only on the name of the requested object, yet intelligent enough to read in the right amount of data and return an object which contains general data that could be used by a number of different commands on the client side.