DESIGN AND IMPLEMENTATION OF USER LEVEL SOCKET APPLICATION PROGRAMMING INTERFACE WITH SOCKET SPLITTING AND MEDIATION

A Thesis
presented to
the Faculty of California Polytechnic State University,
San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

by
Scott Walter Holzer
November 2010
# COMMITTEE MEMBERSHIP

**TITLE:** DESIGN AND IMPLEMENTATION OF USER LEVEL SOCKET APPLICATION PROGRAMMING INTERFACE WITH SOCKET SPLITTING AND MEDIATION  

**AUTHOR:** Scott Walter Holzer  

**DATE SUBMITTED:** November 2010  

**COMMITTEE CHAIR:** Michael Haungs, Dr.  

**COMMITTEE MEMBER:** Aaron Keen, Dr.  

**COMMITTEE MEMBER:** Franz Kurfess, Dr.
ABSTRACT

DESIGN AND IMPLEMENTATION OF USER LEVEL SOCKET APPLICATION PROGRAMMING INTERFACE WITH SOCKET SPLITTING AND MEDIATION

Scott Walter Holzer

Over the past few decades, the size and scope of the Internet has grown exponentially. In order to maintain support for legacy clients, new applications and services have been limited by dependence on traditional sockets and TCP, which provide no support for modifying endpoints after connection setup. This forces applications to implement their own logic to reroute communications to take advantage of composable services or handle failover. Some solutions have added socket operations that allow for endpoints to be redirected on the fly, but these have been limited in scope to handling failover and load balancing.

We present two new sets of socket operations. The first set allows servers to dynamically insert and remove intermediaries into communication streams. This allows applications to decide in real time whether to use services provided by 3rd parties such as encryption, filtering, and compression. In this way, applications can employ dynamic service composition to customize communication between clients and servers. The second set of operations allows sockets to be split such that all frames written to the socket are sent to multiple recipients. This is useful for implementing fast failover and passive communication monitoring. All of these operations are implemented in user space and gracefully handle legacy TCP clients, making quick deployment of distributed Internet applications a real possibility. Performance tests of the new operations on remote hosts show that the overhead introduced is not prohibitive.
ACKNOWLEDGMENTS

First, I would like to thank my wife Emilie for her support during the long hours preparing this thesis. Also, Dr. Haung’s guidance and advice have proven invaluable. Finally, I’d like to thank Dr. Keen and Dr. Kurfess for their time reading and evaluating this thesis, and participating in my defense.
# TABLE OF CONTENTS

LIST OF TABLES ............................................................................................................. ix

LIST OF FIGURES ............................................................................................................. x

1. Introduction ..................................................................................................................... 1  
   Contributions .................................................................................................................. 3

2. Background ..................................................................................................................... 5  
   2.1 Network Reference Models .................................................................................. 5  
   2.2 Transmission Control Protocol ........................................................................... 7  
   2.3 Sockets .................................................................................................................... 7

3. Using uRedSocks Socket Redirection Operations ......................................................... 9  
   3.1 Server Socket Setup: rs_socket, bind, listen, and rs_accept ................................. 9  
   3.2 Client-Side Socket Setup: rs_socket and rs_connect .......................................... 10  
   3.3 Normal Operation: rs_send and rs_recv ............................................................ 11  
   3.3 Inserting an Intermediary: rs_insert .................................................................... 12  
   3.4 Removing an Intermediary: rs_remove ............................................................... 13  
   3.5 Implementing an Intermediary ............................................................................ 13  
   3.6 Splitting a Socket: rs_split ............................................................................... 14  
   3.7 Merging a Split Socket: rs_merge ..................................................................... 15  
   3.8 Promoting a Secondary Server to Primary: rs_promote ...................................... 15  
   3.9 Implementing a Secondary Server .................................................................... 16  
   3.10 Client View of Insert, Remove, Split, and Merge ............................................. 16  
   3.11 Possible Uses of Insert and Remove ............................................................... 17  
   3.12 Possible Uses of Split and Merge ..................................................................... 18
6.3 Intermediaries .......................................................................................................... 39

7. Conclusion .................................................................................................................... 41

8. Future Work .................................................................................................................. 42

Bibliography ..................................................................................................................... 43
LIST OF TABLES

Table 4.1: uRedSocks Header.......................................................................................................................... 19
Table 4.2: New Op Codes and Statuses ............................................................................................................. 20
Table 5.1: Transfer times using plain sockets.................................................................................................... 32
Table 5.2: Local transfer times with uRedSocks ............................................................................................... 33
Table 5.3: Remote transfer times with uRedSocks ............................................................................................. 33
Table 5.4: Local transfer times with Insert ......................................................................................................... 35
Table 5.5: Remote transfer times with Insert .................................................................................................... 35
Table 5.6: Local transfer times with Split .......................................................................................................... 36
Table 5.7: Remote transfer times with Split ....................................................................................................... 37
LIST OF FIGURES

Figure 2.1: OSI and TCP/IP models with uRedSocks ......................................................... 7
Figure 3.1: Server setup ..................................................................................................... 10
Figure 3.2: Client setup .................................................................................................... 11
Figure 3.3: Pseudo-code for intermediary ................................................................. 14
Figure 3.4: Pseudo-code for secondary server ............................................................... 16
Figure 4.1: Sequence diagram for insert operation ..................................................... 22
Figure 4.2: Sequence diagram for two remove operations ........................................ 23
Figure 4.4: Pseudo-code for file server using insert and remove ................................ 28
Figure 4.4: Pseudo-code for file server using insert and remove ................................ 29
Figure 6.1: Multi-homing in SCTP ................................................................................ 39
1. Introduction

There is no doubt that the growth of the Internet has affected nearly every aspect of modern life. From its beginning forty years ago as ARPANET [14], it has exploded in size to reach every continent with nearly 2 billion users [12]. What once was reserved for academic researchers, government agencies, and large corporations is now available to average computer users.

As the size of the Internet has grown, so has the scope and variety of applications and enterprises relying on it. Social networking and user-generated content have changed the way people view the Internet. And entertainment has come to rely on the Internet as streaming media and large-scale multiplayer games become more popular. Enterprises and governments depend on the Internet for e-commerce, delivering software as a service, advertising, and communication. Academia has expanded its use of the Internet with major distributed computing projects such as Stanford University’s Folding@home [17].

Though the Internet has undergone a dramatic transformation, the underlying technologies it uses have remained somewhat stagnant. As new applications have arisen requiring more stringent security, higher performance, and increased scalability, they have largely been bound to use decades-old protocols such as TCP in conjunction with BSD sockets in order to ensure interoperability with legacy systems.
Haungs et al. addressed some of the limitations to TCP and BSD sockets with their RedSocks protocol [7]. While this protocol is successful in allowing for redirectable sockets with minimal performance degradation, it may be difficult to quickly deploy the protocol into the marketplace since its implementation requires kernel-level modification. Also, while RedSocks provides for fault tolerance and load balancing, there are many complex, commonly implemented services (e.g. encryption, filtering, and monitoring) that would gain from a more robust set of socket redirection operations.

Stream Control Transport Protocol (SCTP) [18] has many of the same benefits and drawbacks as kernel-space RedSocks. Like RedSocks, SCTP is normally implemented in kernel space and provides for failover using multi-homing and heartbeats. One major disadvantage of SCTP is that both the client and server must support SCTP; a client that only supports TCP and UDP cannot connect to an SCTP socket.

Liang has implemented a version of RedSocks in user space, called uRedSocks [11]. uRedSocks supports the same implicit (failover) and explicit redirection operations provided by RedSocks. Additionally, uRedSocks can be used safely with legacy clients: when a non-uRedSocks enabled client connects to a server using uRedSocks, the uRedSocks accept method will automatically treat the connection as a plain TCP socket with no further logic required by the server or client code.

uRedSocks allows for quick adoption due to its implementation in user space and support
for non-uRedSocks-enabled clients, but its added value is limited to fault tolerance and load balancing.

**Contributions**

In this paper, we present the design and implementation of an extension to uRedSocks that adds support for more advanced socket redirection. We introduce two new sets of socket operations: insert/remove, and split/merge/promote.

The insert command modifies a socket to route future messages through an intermediary server. Insert may be invoked multiple times, causing intermediary servers to be daisy-chained together. Intermediary servers may monitor and modify any communication that passes through. The remove command causes the last intermediary server added to be removed from the chain.

The split command splits a socket so that all future messages written to the socket will be sent to both the original server and one or more secondary servers. Each call to split will add another secondary server to the list. The merge command causes one or all secondary server sockets to be shut down and closed. The promote operation can be invoked by the primary server to cause one of the secondary servers to take over and be promoted to primary server status.

The insert and remove commands are useful whenever a server wishes to dynamically decide whether to use the services provided by one or more 3rd parties. For example,
servers may decide on the fly whether to insert intermediaries that encrypt data, compress data, monitor and log data, or all three, depending on network conditions. The split operation is useful primarily to enable fast failover and load balancing. Splitting a socket also provides a mechanism for setting up secondary servers to passively monitor client communications, with very little performance degradation.
2. Background

Before presenting the details of the uRedSocks protocol and implementation, it is useful to discuss the concepts and technologies that we are building on: namely, network reference models, TCP, and sockets.

2.1 Network Reference Models

In 1979, the International Organization for Standardization (ISO) adopted the Open Systems Interconnection (OSI) Reference Model to provide a framework in which heterogeneous networks could connect to one another [20]. The OSI Reference Model segments network implementation into seven distinct layers: application, presentation, session, transport, network, data link, and physical (figure 2.1). The model specifies what kind of services are provided by each layer, but does not specify how those services should be implemented. For example, the model indicates that the physical layer is responsible for encoding bits onto the communication medium, but does not state whether the bits should be transmitted on a copper wire or via satellite.

In the OSI model, when an application wishes to send a message across the network, the data is passed from the application layer down the stack to the layer below (presentation layer). Each layer processes the message received from the layer above, may perform some transformation on the message (e.g. segmentation), and passes the message to the next layer below. On the receiving host, the data is read by the physical layer and passed up the network stack until it finally reaches the application.
One of the advantages of the layered approach to networking is that layer implementations can be swapped out as technologies and usage change. For example, if new discoveries lead to better routing algorithms, network layer protocols can be updated accordingly without affecting the other six layers of the network stack.

In 1974, Cerf and Kahn proposed a Transmission Control Program [6] that evolved into the Transmission Control Protocol [8] and TCP/IP reference model. Like the OSI reference model, the TCP/IP reference model takes a layered approach to networking. Unlike OSI, the TCP/IP model was developed with particular protocols in mind (namely, Transmission Control Protocol and Internet Protocol) [19]. The TCP/IP model removes the presentation and session layer, replaces the network layer with the Internet layer, and combines the data link and physical layers into one host-to-network layer.

Figure 2.1, adapted from Tanenbaum [19], compares the OSI and TCP/IP reference models and shows how uRedSocks fits in. The uRedSocks protocol could best be classified as a session layer protocol. It sits on top of the transport layer, and is used by the application layer to provide socket redirection and socket splitting.
### 2.2 Transmission Control Protocol

The uRedSocks implementation uses the Transmission Control Protocol (TCP) to send frames between clients, servers, and intermediaries. TCP is a widely supported protocol, with implementations even available on mobile phones [15]. The protocol provides a connection-oriented, reliable communication stream that ensures that all frames are received by their intended recipient without errors in the order that they were sent.

### 2.3 Sockets

The IEEE POSIX specification defines a socket as “a file of a particular type that is used as a communications endpoint for process-to-process communication.” [2] Stated another way, sockets provide an abstraction to network communication that allows applications to use normal file I/O functions to send and receive data to and from one another. This abstraction shields applications from many of the details of network communication and fits well with the spirit of layered network models.
uRedSocks extends standard sockets by adding the ability for one socket to point to multiple peers, and by allowing for a socket to be redirected from one peer to another. These extensions are transparent to applications, which may treat uRedSocks sockets as normal sockets by replacing normal socket system calls with their uRedSocks equivalents.
3. Using uRedSocks Socket Redirection Operations

The insert, remove, split, merge, and promote operations are designed to be as similar to existing BSD socket operations as possible. One of the key goals of uRedSocks is to create operations that are easy to add to new and legacy applications alike. uRedSocks is implemented in C and uses standard C libraries. To use the uRedSocks methods in a C application, include the uredsocks.h header file and compile with uredsocks.c on the compiler command line. The following subsections describe how applications should be written to use the new operations we provide.

3.1 Server Socket Setup: rs_socket, bind, listen, and rs_accept

Setting up a uRedSocks server to listen for and accept connections from clients requires four function calls: rs_socket, bind(2), listen(2) and rs_accept; the listen(2) and bind(2) system calls are unmodified. The new rs_socket function has the same signature as the socket(2) system call, but performs some additional setup logic to mark the socket as a uRedSocks socket.

Rs_accept extends the accept(2) function to allow receipt of application data upon connection from the client:

```c
int rs_accept(int socket, struct sockaddr *address, socklen_t *address_len, unsigned char *app_data, int *app_data_len)
```

The argument app_data_len is a pointer to an integer representing the maximum length of application data to accept, and should never be larger than the number of bytes allocated
for the app_data buffer. When rs_accept returns, the integer referenced by app_data_len will be set to the actual length of application data received. If the amount of data received is larger than the app_data buffer, the remainder can be retrieved with subsequent calls to rs_read. Upon successfully accepting a client connection, rs_accept returns a non-negative socket descriptor. If there was an error, -1 is returned and errno is set as appropriate.

Figure 3.1 shows sample code of a simple server that uses rs_socket, bind(2), listen(2) and rs_accept to set up a uRedSocks socket and print out received application data.

```c
sd = rs_socket(AF_INET, SOCK_EOP_STREAM, 0);
bind(sd, (struct sockaddr *) &local, sizeof(local);
listen(sd, backlog);

csd = rs_accept(sd, address, address_len, app_data, &app_data_len);
if(app_len)
    printf("App data from accept: %s ", app_data);
else
    printf("No app data received from accept.");
```

**Figure 3.1: Server setup**

### 3.2 Client-Side Socket Setup: rs_socket and rs_connect

To connect to a uRedSocks server, clients use modified versions of socket(2) and connect(2): rs_socket and rs_connect. As with server setup, clients use rs_socket exactly as they would normally use socket(2). Rs_connect extends connect(2) to allow the client to send application data along with the connection request:

```c
int rs_connect(int socket, struct sockaddr *address,
          socklen_t *address_len, unsigned char *app_data,
          int app_data_len)
```

The app_data argument is a buffer containing app_data_len bytes of application data to
send to the server. Rs_connect returns zero on success; upon failure, -1 is returned and
errno will be set accordingly.

Figure 3.2 shows sample code for a simple client which will connect to the server
described in figure 3.1 and send application data along with the connection request.

\[
\text{sd = rs_socket(AF_INET, SOCK_EOP_STREAM, 0);} \\
\text{rs_connect(sd, remote, sizeof(struct sockaddr_in), "Hello World!", 13);} \\
\] 

**Figure 3.2: Client setup**

### 3.3 Normal Operation: rs_send and rs_recv

During normal operation, clients and servers use rs_send and rs_recv exactly as they
would use send(2) and recv(2):

\[
\begin{align*}
\text{ssize_t rs_send(int socket, const void *buffer, size_t length,} \\
&\quad \text{int flags)} \\
\text{ssize_t rs_recv(int socket, void *buffer, size_t length, int flags)}
\end{align*}
\]

For both functions, the flags are passed to the underlying send(2) and recv(2) system
calls. Rs_send will check to see if the socket is uRedSocks enabled. If it is, it will send
buffer to the client, server, intermediary, and/or secondary servers as appropriate; if the
socket is not uRedSocks enabled, it functions as a plain send(2) call.

Rs_recv also checks to see whether the socket is uRedSocks enabled. If it is, it checks the
contents of the uRedSocks header and handles any special operations. Then, it places the
received payload into buffer and returns the number of bytes received. If the socket is not
uRedSocks enabled, rs_recv functions as a plain recv(2) call.
3.3 Inserting an Intermediary: rs_insert

One of the new socket operations introduced by uRedSocks is the ability to insert one or more intermediaries into the communication stream between the client and the server.

Intermediary insertion is accomplished with a call by the server to rs_insert:

```c
ssize_t rs_insert(int socket, const void *app_data,
                  size_t app_data_len, int flags,
                  const char *interm_host, unsigned short interm_port,
                  int *result)
```

The socket argument is the client socket. App_data optionally contains application data of length app_data_len that will be forwarded to the new intermediary to aid in setup. The flags argument will be passed along to the underlying send(2) system calls. Interm_host should contain the host name of the intermediary to insert, in a format usable by gethostbyname(3). Port specifies the intermediary port to connect to. Result is a pointer to an integer that will be set to INSERT_SUCCESSFUL upon success, INTERM_UNAVAILABLE if the intermediary cannot be reached, NOT_RS_SOCKET if the socket specified is not uRedSocks enabled, or CLIENT_ERROR for any error reported by the client. The function returns the number of bytes of data sent to the intermediary.

One important feature of rs_insert is that a server may safely use the function even when the server does not know whether the client is uRedSocks enabled. If the client is using a plain TCP connection, the rs_insert functions as a no-op.

After calling rs_insert on a socket, the server should continue to use the same socket
descriptor in all future calls to rs_send. In the case of a successful insert, logic inside of rs_send will map the original socket to the new intermediary socket without any further work by the calling server code.

3.4 Removing an Intermediary: rs_remove

When a server wishes to remove the most recently inserted intermediary from the communication stream, it makes use of the rs_remove function:

```c
int rs_remove(int socket)
```

Upon success, this operation returns zero; if the client is not uRedSocks enabled, or there is no intermediary present, NOT_RS_SOCKET is returned and rs_remove functions as a no-op. If the server cannot re-establish a connection with the client (or second-most-recently inserted intermediary) after removal, RECONNECT_ERROR is returned. Any future attempts to send or receive on the socket will fail.

3.5 Implementing an Intermediary

Unlike clients, intermediary servers are expected to know that they are being used with the uRedSocks protocol. The most basic intermediary, which simply passes along all messages received from the client to the server and vice versa, would follow the pseudo-code in Figure 3.3.
LISTEN for and ACCEPT connection
SPAWN a new thread

THREAD:
READ from socket
IF op code is INS_OP_MEDIATE
   STORE token and socket in map
   SEND acknowledgment
   WHILE reading from server socket is successful
      SEND frame from server to client
ELSE IF op code is INS_OP_C2I_START
   FIND server socket in map based on token
   SEND acknowledgement
   WHILE reading from client socket is successful
      IF op code is INS_OP_CLIENT_INSERT
         # Handle daisy-chain insert request
         CONNECT to new intermediary (daisy-chain)
         UPDATE socket in map to point to new intermediary
      ELSE IF op code is INS_OP_S2I_REMOVE
         # Handle removal
         SEND INS_OP_I2C_REMOVE to client
         DISCONNECT from client and server
      ELSE
         # No special operation
         SEND frame to server client
   Figure 3.3: Pseudo-code for intermediary

3.6 Splitting a Socket: rs_split

A server may use rs_split to instruct the client to split its socket, such that all future messages from the client are sent to both the primary server and one or more secondary servers:

int rs_split(int socket,
            const char *secondary_host, unsigned short secondary_port)

Socket is the client socket. Secondary_host specifies the host name of the secondary server to split the socket to, in a format usable by gethostbyname(3); port specifies the port to connect to on the secondary server.

A socket may be split an arbitrary number of times, limited only by system resources.
Upon success, zero is returned. If the secondary server is unavailable, SECONDARY_UNAVAILABLE is returned; if the socket is not uRedSocks enabled, NOT_RS_SOCKET is returned. As with rs_insert, it is safe to call rs_split even when the server doesn’t know whether the client is supports uRedSocks. When rs_split is called on a plain socket, it functions as a no-op.

3.7 Merging a Split Socket: rs_merge

uRedSocks provides two operations for merging a split socket. Rs_merge removes all secondary servers, and rs_merge_one removes only the most recently added secondary server:

```c
int rs_merge(int socket)
int rs_merge_one(int socket)
```

Both functions return zero on success, or NOT_RS_SOCKET if the socket is not uRedSocks enabled. If the client cannot be reached during the merge call, the socket will be shut down and closed, and SOCKET_DISCONNECT will be returned. If either of the merge calls are called on a socket that is currently not split, the call is considered successful and functions as a no-op.

3.8 Promoting a Secondary Server to Primary: rs_promote

The server can cause the most recently added secondary server to be promoted to primary status with a call to rs_promote:

```c
ssize_t rs_promote(int socket, unsigned char *app_data,
                    size_t app_data_len, int *result)
```

Socket is the client socket. App_data optionally contains application data, of length
app_data_len, that will be passed along to the promoted secondary server to aid in setup. Result is a pointer to an integer which will be set to PROMOTE_SUCCESSFUL on success, PROMOTE_ERROR for any errors connecting to the secondary server, or NOT_RS_SOCKET if the socket specified is not uRedSocks enabled. The actual amount of application data sent is returned; -1 is returned if there was an error. If promotion was successful, the client socket will be shut down and closed, and any future attempts to send to this socket will result in an error.

3.9 Implementing a Secondary Server

As with intermediaries, secondary servers must be uRedSocks-aware. A secondary file transfer server could follow the pseudo-code in figure 3.4.

```
LISTEN for and ACCEPT connection
WHILE reading from client socket is successful
  IF op code is SPL_OP_PROMOTE
      READ application data to set up resumption of file transfer
      SEND acknowledgement to client
  # Resume file transfer where primary server left off
  WHILE reading from file is successful
      SEND frame with file data to client
  BREAK
```

**Figure 3.4: Pseudo-code for secondary server**

3.10 Client View of Insert, Remove, Split, and Merge

One advantage of uRedSocks is that client implementations need only replace system calls with their uRedSocks equivalents – no extra logic is required. The logic for handling all insert, remove, split, and merge operations is implemented within the rs_recv function itself, and is not visible to the client; likewise, the logic for determining whether frames should be sent to intermediaries and secondary servers is handled by rs_send.
One important note about rs_recv is that it will block until data that is intended for the client is received. For example, if rs_recv receives a promote command from the server that includes application data, the data will be forwarded along to the new primary server silently and rs_recv will call itself recursively. The function will only return when it receives a message from the new server that is intended for the client.

### 3.11 Possible Uses of Insert and Remove

Servers may wish to send traffic through intermediaries for a variety of reasons. These are a few different intermediary types that may prove useful:

1. **Encryption.** If the communication path from the client to the server includes untrusted links, intermediaries may be inserted on either end of those links to protect the data passed through with encryption. This is one type of intermediary we implemented for our experiments and analysis.

2. **Compression.** If a server detects that its connection to the client is poor, it may be useful to insert intermediaries on either end of the poor connection to compress data that is passed through.

3. **Monitoring and Filtering.** When a client accesses the services provided by a server, the server may wish to protect itself by inserting a trusted intermediary that provides monitoring and filtering of client requests. Or, if a client is being billed for accessing a server, a 3rd party intermediary trusted by both the client and server could be inserted to monitor the traffic and report on usage.

4. **Composable services.** All of the aforementioned intermediary types, and
certainly some others, could be combined as composable services. uRedSocks enabled servers could insert multiple intermediaries, with each intermediary optimized to provide a particular service.

3.12 Possible Uses of Split and Merge

There are at least a few situations where splitting a connection may be advantageous:

1. **Fast Failover.** After a split, there are multiple servers and one client. Since the secondary servers already have a connection set up to the client and are monitoring communications, failover can be very fast. This is the type of secondary server we implemented for experiments and analysis.

2. **Passive Monitoring.** As with insert, a secondary server may be added to perform passive monitoring. By using a split instead of an insert, communications between the client and server can continue even if the monitoring server fails; also, performance would not degrade if a secondary server’s system or network performance degrades. This can be an advantage if monitoring is intended to be optional (e.g., for general business intelligence), but is a disadvantage if monitoring is intended to be required (e.g., for billing).
4. Design and Implementation

The uRedSocks protocol and implementation enhancements presented here build on the original design by Liang [11]. As such, it retains key functionality such as explicit redirection, implicit redirection with failure detection using heartbeats, and support for legacy clients, while adding support for new operations. The following sections describe how the uRedSocks protocol has been modified, and how the new operations function.

4.1 General Protocol Information

uRedSocks uses TCP sockets for normal communications between the client and server. Each uRedSocks frame includes a header with the fields shown in table 4.1, followed by a payload of zero or more bytes. We have extended the uRedSocks header by adding a token field.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op Code or Status</td>
<td>Unsigned int</td>
<td>Specifies uRedSocks operation, or the status of an operation</td>
</tr>
<tr>
<td>IP Address</td>
<td>Unsigned int</td>
<td>IP address of intermediary or secondary server, if appropriate</td>
</tr>
<tr>
<td>Port</td>
<td>Unsigned short</td>
<td>Port of intermediary or secondary server, if appropriate</td>
</tr>
<tr>
<td>Length</td>
<td>Unsigned int</td>
<td>Length of payload</td>
</tr>
<tr>
<td>Token</td>
<td>Unsigned int</td>
<td>Token to send to intermediary during insert</td>
</tr>
</tbody>
</table>

The IP address and port fields are only used when inserting an intermediary or splitting to a new secondary server. Length is always equal to the length of the data in the frame’s payload. Depending on the op code, the payload can either be data intended for the client or server (as in the case of a normal rs_send), or can be application data destined for a
new intermediary server or promoted secondary server. The token field is used during the
insert sequence, as described in section 4.3.1. In addition to the op codes specified by
Liang, we add the op codes and statuses listed in table 4.2. The meaning of each is
explained in greater detail in future sections.

<table>
<thead>
<tr>
<th>Op Code or Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INS_OP_CLIENT_INSERT</td>
<td>Server instructing client to insert intermediary</td>
</tr>
<tr>
<td>INS_OP_MEDIATE</td>
<td>Server instructing intermediary to prepare for mediation</td>
</tr>
<tr>
<td>INS_OP_S2I_REMOVE</td>
<td>Server instructing intermediary to remove itself</td>
</tr>
<tr>
<td>INS_OP_I2C_REMOVE</td>
<td>Intermediary notifying client of its removal</td>
</tr>
<tr>
<td>SPL_OP_SPLIT</td>
<td>Server instructing client to split its socket</td>
</tr>
<tr>
<td>SPL_OP_MERGE</td>
<td>Server instructing client to drop all secondary servers</td>
</tr>
<tr>
<td>SPL_OP_MERGE_ONE</td>
<td>Server instructing client to drop the most recently added secondary server</td>
</tr>
<tr>
<td>SPL_OP_PROMOTE</td>
<td>Server instructing client to promote the most recently added secondary server to primary status</td>
</tr>
<tr>
<td>ACK</td>
<td>Operation was successful</td>
</tr>
<tr>
<td>NAK</td>
<td>Operation was unsuccessful</td>
</tr>
<tr>
<td>INTERM_UNAVAILABLE</td>
<td>Client could not connect to intermediary</td>
</tr>
<tr>
<td>SECONDARY_UNAVAILABLE</td>
<td>Client could not connect to secondary server</td>
</tr>
</tbody>
</table>

In addition to the TCP sockets used for normal communication, uRedSocks retains the
heartbeat functionality (for implicit redirection) implemented by Liang. Heartbeats are
sent from the client to the server using UDP; this functionality has not been altered.

### 4.3 Inserting and Removing Intermediaries

The server, not the client, initiates insertion and removal of intermediaries into a
communication stream. The following sections detail how the server coordinates with the
client and intermediary server to accomplish insertion and removal of intermediaries.
4.3.1 Inserting an Intermediary

As the sequence diagram in figure 4.1 shows, the server initiates insertion by connecting to the intermediary and sending an INS_OP_MEDIATE command with a randomly generated token. If the intermediary agrees to be inserted, the server then sends an INS_OP_CLIENT_INSERT command to the client, with the same token that was sent to the intermediary. Upon receiving this command, the client forwards the token to the intermediary with an INS_OP_C2I_START command. (C2I is short for “Client to Intermediary”). If the intermediary responds favorably, the client notifies the server of the successful insertion, and the original socket between the client and server is closed. In both the client and the server, the original socket is mapped to the new socket that was opened to the newly inserted intermediary.

On the server side, the logic for inserting the intermediary is implemented in the rs_insert function; on the client side, it is included in the rs_recv function. Since intermediaries are always custom-written to work with uRedSocks, they are expected to implement their own logic for insertion and removal. The token is provided as a mechanism for intermediaries to match up clients and servers.

If a server calls rs_insert on a socket that already has an intermediary, the insertion flow proceeds as if the existing intermediary were the client. In other words, the server first contacts the new intermediary, then sends the INS_OPCLIENT_INSERT command to the existing intermediary, which in turn connects to the new intermediary and remaps its original server socket to the new intermediary socket. In this case, the client is completely
unaffected by the insert since all of the coordination occurs between the server, the existing intermediary, and the new intermediary.

![Sequence Diagram](image)

**Figure 4.1: Sequence diagram for insert operation**

### 4.3.2 Removing an Intermediary

When a server calls `rs_remove`, a new socket will be created for the client to connect to after the intermediary has been removed. Then, the `INS_OP_S2I_REMOVE` command will be sent to the intermediary, along with the IP address and port that the new socket is bound to. The intermediary will then forward the IP address and port along with an `INS_OP_I2C_REMOVE` command to the client, which will connect to the new socket on
the server. After this connection occurs, the intermediary sockets are closed by both the
server and the client.

If there is more than one intermediary in play when remove is called, the second-most-
recently added intermediary acts as the client in the removal procedure, while the most
recently added intermediary is removed. Figure 4.2 shows how two intermediaries would
be removed by two calls to rs_remove.

Figure 4.2: Sequence diagram for two remove operations

4.4 Split, Merge, and Promote

As with insert and remove, the split, merge, and promote commands are initiated by the
server and require the cooperation of the client. These sections describe how sockets are
split and merged, and how secondary servers are promoted to primary status
4.4.1 Splitting a Socket

The rs_split function initiates a split by looking up the secondary server host name provided using gethostbyname(3). If this is successful, the IP address and port are sent to the client along with a SPL_OP_SPLIT command. The client then attempts to connect to the secondary server and returns a positive acknowledgement upon success.

It is assumed that secondary servers will be uRedSocks aware, and will be expecting connections from clients responding to a split command. Therefore, there is no need for the client to send any special commands to the secondary server or expect an acknowledgement. If the connection is established successfully, then a successful split is assumed. Figure 4.3 in section 4.4.3 shows a sequence diagram of the split operation.

4.4.2 Merging a Socket

Merging a socket is by far the simplest of all uRedSocks operations. When a client receives the SPL_OP_MERGE command, it iterates through its list of secondary servers, closes each socket, and removes the server from the list. If a SPL_OP_MERGE_ONE command is received, only the most recently added secondary server is removed.

4.4.3 Promoting a Secondary Server

To promote a secondary server to primary status, the rs_promote function sends the SPL_OP_PROMOTE command, along with application data, to the client. The client then forwards the command and data to the most recently added secondary server. If the secondary server replies with a positive acknowledgement, the original server is notified.
of the success and the server socket is closed. All future calls by the client to rs_send or rs_recv on the socket will be mapped to the newly promoted secondary server’s socket. Figure 4.3 is a sequence diagram showing a split operation followed by a promote operation.

![Sequence diagram for split and promote operation](image)

**Figure 4.3: Sequence diagram for split and promote operation**

### 4.5 Send and Recv

From the caller’s perspective, the uRedSocks send and receive functions (rs_send and rs_recv) work exactly like the send(2) and recv(2) system calls they replace. Under the
hood, however, they implement extra logic to work with the insert, remove, split, merge, and promote operations.

When rs_send is invoked, it first checks to see whether the socket supplied is uRedSocks enabled. If it is not, the arguments are passed along to a normal send(2) call and the results are returned. If the socket is uRedSocks enabled, it initializes a uRedSocks header with the NORMAL op code and iterates through the list of secondary servers, sending the header and payload to each. Finally, rs_send checks to see whether an intermediary has been inserted. If so, the header and payload are sent to the intermediary; otherwise, they are sent directly on the specified socket.

Like rs_send, rs_recv also checks to see whether the socket is uRedSocks enabled. If it is not, the arguments are passed along to a normal recv(2) call and the results are returned. Otherwise, rs_recv checks to see whether there is an intermediary and calls recv(2) on either the normal or intermediary socket accordingly. Next, the function checks the op code and payload length in the uRedSocks header received. If a NORMAL op code is received, the payload is returned to the caller; if a split, insert, remove, merge, or promote op code is received, the command is handled and rs_recv calls itself recursively. This continues until a NORMAL op code is received.

One of the challenges of implementing rs_recv is that TCP does not preserve message
boundaries. This means that a call to recv is not guaranteed to consume all of the data that has been sent on the socket, even if the buffer size is sufficient. Therefore, rs_recv cannot simply assume that it will be immediately receiving a uRedSocks header each time it is invoked. To account for this, each time rs_recv receives a uRedSocks header, it compares the payload length to the length of data actually received. If the total frame length (header size + payload size) exceeds the length of data received, the difference is recorded in a “bytes unread” field associated with the socket. The next time rs_recv is called, it will attempt to read the unread bytes from the socket before looking for another header.

4.6 Example Server using Insert and Remove

In figure 4.4 we show pseudo-code for a file server that uses the insert and remove operations to handle billing and proactively enable compression when latency is unacceptably high. In this example, the server accepts a connection from a client and reads the file name supplied. If the file exists on the server, then an intermediary is inserted to monitor the communication so the client can be billed for the transaction. Then, while the file is being transferred, the server checks to see how long it takes to receive an acknowledgement after every frame is sent. If the time exceeds a threshold, the server inserts two intermediaries for compression and decompression. Finally, the server cleans up by removing all intermediaries.
LISTEN for and RS_ACCEPT connection
SPAWN a new thread

THREAD:
RS_RECV from socket
IF file name supplied is valid
  compressing := false
  RS_INSERT 3rd party monitoring intermediary
OPEN file
WHILE READ from file is successful
  startTime := now
  RS_SEND file chunk
  RS_RECV acknowledgement
  endTime := now
  IF endTime - startTime > threshold AND NOT compressing
    RS_INSERT compression intermediary
    compressing := true
  END IF
LOOP
IF compressing
  # Remove compression intermediaries
  RS_REMOVE
  RS_REMOVE
END IF
  # Remove billing intermediary
  RS_REMOVE
ELSE
  RS_SEND negative acknowledgement
END IF
RS_CLOSE socket

Figure 4.4: Pseudo-code for file server using insert and remove

4.7 Example Server using Split and Promote

In figure 4.5 we show pseudo-code for a file server that uses the split, promote, and merge operations to proactively change servers if network performance deteriorates. In this example, the server accepts a connection from a client and reads the file name supplied. If the file exists on the server, the server splits the socket to a secondary server. Then, while the file is being transferred, the server checks to see how long it takes to receive an acknowledgement after every frame is sent. If the time exceeds a threshold, the
server promotes the secondary server to primary status in an attempt to avoid network congestion or failure.

LISTEN for and RS_ACCEPT connection
SPAWN a new thread

THREAD:
RS_RECV from socket
IF file name supplied is valid
    promoted := false
    RS_SPLIT secondary server
    OPEN file
    WHILE READ from file is successful AND NOT promoted
        startTime := now
        RS_SEND file chunk
        RS_RECV acknowledgement
        endTime := now
        IF endTime - startTime > threshold
            RS_PROMOTE
            promoted := true
        END IF
    END LOOP

# Detach secondary server
IF NOT promoted
    RS_MERGE
END IF
ELSE
    RS_SEND negative acknowledgement
END IF

RS_CLOSE socket

Figure 4.4: Pseudo-code for file server using insert and remove
5. Experiments and Analysis

To test the performance of uRedSocks vs. plain BSD sockets, we created a simple file transfer client and server, “rcopy” and “rserver,” respectively. Rserver is a multi-threaded server that listens for clients, accepts connections, receives a file name from the client, and sends the file in 1023 byte chunks to the client along with a 1-bit flag indicating whether the server encountered the end of the file. Rcopy writes the file to disk and records the start and end time of the file transfer.

For control testing, we used normal networking system calls in both rcopy and rserver (e.g. connect(2), accept(2), send(2), recv(2), etc.) For uRedSocks testing, we replaced all applicable system calls with their uRedSocks equivalents.

5.1 Test Environments

Each test was run in two different test environments. First, the tests were run with all clients, servers, and intermediaries (when applicable) running on the same computer. The local test computer is an Apple MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of RAM running Mac OS X 10.6.4 (Snow Leopard). The applications were compiled using GCC version 4.2.1 with the O3 flag set to enable all performance optimizations. To ensure minimal interference with the tests, all other applications on the system were closed except the applications under test (running in Terminal), and Microsoft Excel (to record test results).

Next, in order to get a better view of real-world performance, the same tests were run on
Amazon EC2 instances running Amazon Linux. All instances were “small” instances (1.7 GB memory, one virtual core with one EC2 compute unit, 160 GB instance storage, 32-bit platform) running in Amazon’s East region. (One EC2 compute unit is equivalent to 1.0-1.2 GHz 2007 Opteron processor [1]). The applications were compiled using GCC version 4.1.2 with the O3 flag set to enable all performance optimizations. For every test, each client, server, and intermediary ran on a separate instance.

In all cases, run times were calculated by comparing the results of two gettimeofday(2) system calls in rcopy: immediately before the first call to socket(2) or rs_socket, and immediately after the call to close the socket when the transfer was complete.

5.2 Test Files

In order to gauge how the performance of uRedSocks varied based on file size, we timed the transfer of three different files:

1. A ten kilobyte text file (hereafter referred to as the small file);
2. A ten megabyte binary file (hereafter referred to as the medium file); and
3. A one gigabyte binary file (hereafter referred to as the large file)

For each of the test cases, we recorded the transfer time of 100 small file transfers, 100 medium file transfers, and six large file transfers. (Large file transfers were limited to 6 runs each because Amazon charges for data stored and transferred between instances, and for the time that the instances are running). In order to account for cache warm-up and intermittent, externally-caused spikes in processor and network usage, we:

1. Discarded the first five test runs for the small and medium file transfers, and
discarded the first test run for the large file transfers.

2. Used median transfer time instead of mean transfer time in order to discard outliers.

5.3 Control Testing – Standard Sockets

The control version of rcopy and rserver use plain BSD system calls to transfer files using TCP. The results in table 5.1 show the file transfer times, and will be used as the basis for performance analysis.

<table>
<thead>
<tr>
<th>Table 5.1: Transfer times using plain sockets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control (Plain Sockets) Transfer Times</td>
</tr>
<tr>
<td>10 KB (sec)</td>
</tr>
<tr>
<td>Local</td>
</tr>
<tr>
<td>Remote</td>
</tr>
</tbody>
</table>

5.4 No Operation Testing

In order to obtain a baseline performance difference between uRedSocks and plain sockets, rcopy and rserver were simply modified to replace all networking system calls with their uRedSocks equivalent. For these tests, no special uRedSocks operations were used. Each test run was executed with the heartbeat processing disabled, and again with heartbeat processing enabled.

As tables 5.2 and 5.3 show, the overhead introduced by uRedSocks is severe for small file transfers. This is because the time taken by the uRedSocks protocol negotiation procedure, which requires the client to connect to the server twice, cannot be amortized
It is worth noting that local testing represents the worst case for uRedSocks. Since there is no network latency, any performance overhead can be attributable directly to uRedSocks processing.

### Table 5.2: Local transfer times with uRedSocks

<table>
<thead>
<tr>
<th></th>
<th>10 KB</th>
<th>10 MB</th>
<th>1 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Overhead Multiplier</td>
<td>Time (sec)</td>
</tr>
<tr>
<td>HB</td>
<td>Disabled</td>
<td>0.88650</td>
<td>448</td>
</tr>
<tr>
<td>HB</td>
<td>Enabled</td>
<td>0.99909</td>
<td>505</td>
</tr>
</tbody>
</table>

### Table 5.3: Remote transfer times with uRedSocks

<table>
<thead>
<tr>
<th></th>
<th>10 KB</th>
<th>10 MB</th>
<th>1 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Overhead Multiplier</td>
<td>Time (sec)</td>
</tr>
<tr>
<td>HB</td>
<td>Disabled</td>
<td>0.98887</td>
<td>305</td>
</tr>
<tr>
<td>HB</td>
<td>Enabled</td>
<td>0.88754</td>
<td>274</td>
</tr>
</tbody>
</table>

### 5.5 Insert Operation Testing

To test the performance of the insert operation, we executed three scenarios:

1. Inserting one simple pass-through intermediary after the first frame with file data is sent: This intermediary forwards all messages from the client to the server, and vice versa. This test case is designed to simulate the scenario where an intermediary is used for logging or monitoring communications between two
parties.

2. Inserting two simple pass-through intermediaries after the first frame with file data is sent: This test case is designed to test the performance hit realized when additional intermediaries are inserted into the communication path.

3. Inserting two encrypting/decrypting intermediaries before the first frame with file data is sent: In this case, frames from the server are sent unencrypted to the first intermediary. The first intermediary uses 256-bit AES encryption to encrypt the frame’s payload before sending it along to the second intermediary. The second intermediary decrypts the payload and forwards it to the client. (The intermediaries also work in reverse, so that messages from the client to the server are also encrypted).

Note that in all cases, heartbeat processing was enabled.

The results of the insert operation testing reveal that inserting simple intermediaries has little effect on performance. Especially during remote testing, the overhead of using intermediaries was nearly identical to the overhead of using no intermediaries with heartbeat processing enabled. Naturally, if one of the intermediaries began experiencing degradation in network or system performance, we could expect the overhead to increase as it becomes a bottleneck for all traffic flowing through.

One interesting result, as seen in tables 5.4 and 5.5, is the performance of the encrypting intermediaries in local testing compared to their performance in remote testing. In local testing, transfer times suffer when encrypting intermediaries are used,
while in remote testing the transfer times are flat. This highlights the fact that local testing shows the worst case scenario by exposing all additional processing overhead, while remote testing tends to hide it due to network latency.

<table>
<thead>
<tr>
<th>Table 5.4: Local transfer times with Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 KB</td>
</tr>
<tr>
<td>Time (sec)</td>
</tr>
<tr>
<td>One Intermediary</td>
</tr>
<tr>
<td>Two Intermediaries</td>
</tr>
<tr>
<td>Encrypting Intermediaries</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5.5: Remote transfer times with Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 KB</td>
</tr>
<tr>
<td>Time (sec)</td>
</tr>
<tr>
<td>One Intermediary</td>
</tr>
<tr>
<td>Two Intermediaries</td>
</tr>
<tr>
<td>Encrypting Intermediaries</td>
</tr>
</tbody>
</table>

5.6 Split and Promote Operation Testing

To test the performance of the split and promote operations, we executed three scenarios:

1. Splitting the socket to one secondary server: This scenario simulates the case where a secondary server is brought in to monitor communications from the client and be ready to take over as the primary server at the request of the original server.
2. Splitting the socket to two secondary servers: This case is intended to measure the performance hit realized by splitting the socket to more than once.

3. Splitting the socket to two secondary servers, then promoting one of them to be the new primary server: This case simulates the scenario where a fast failover, at the request of the original server, occurs shortly after the file transfer starts.

As tables 5.6 and 5.7 show, there is very little, if any, overhead introduced by splitting the socket multiple times. One peculiarity, which looks like a test data error at first glance, is the sharp overhead drop-off in most cases when the promote command executes. This reveals a weakness in the current promote implementation: after promotion occurs, the heartbeat processing is suspended and implicit failover is no longer supported. The overhead in the promotion test cases is quite similar to the overhead of no-op testing with heartbeat processing disabled. Implementing heartbeat processing after promotion is left as future work.

<table>
<thead>
<tr>
<th></th>
<th>10 KB</th>
<th>10 MB</th>
<th>1 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Overhead</td>
<td>Time (sec)</td>
</tr>
<tr>
<td>One Secondary</td>
<td>1.1223</td>
<td>567</td>
<td>1.9876</td>
</tr>
<tr>
<td>Two Secondaries</td>
<td>1.0696</td>
<td>540</td>
<td>1.3769</td>
</tr>
<tr>
<td>Promote</td>
<td>1.0374</td>
<td>524</td>
<td>0.9862</td>
</tr>
</tbody>
</table>

Table 5.6: Local transfer times with Split
<table>
<thead>
<tr>
<th>10 KB</th>
<th>10 MB</th>
<th>1 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Overhead</td>
</tr>
<tr>
<td></td>
<td>(sec)</td>
<td>Multiplier</td>
</tr>
<tr>
<td>One Secondary</td>
<td>0.93087</td>
<td>287</td>
</tr>
<tr>
<td>Two Secondaries</td>
<td>0.98364</td>
<td>304</td>
</tr>
<tr>
<td>Promote</td>
<td>1.0060</td>
<td>310</td>
</tr>
</tbody>
</table>

5.6 Performance Testing Conclusions

Much of the performance overhead of using uRedSocks can be attributed to heartbeat processing and protocol negotiation – the insert, split, and promote commands have little performance impact. Heartbeat processing is required for implicit failover support, and protocol negotiation is required in order for uRedSocks servers to be able to function with legacy clients using plain sockets. uRedSocks could be extended in the future to support switching these features off in order to boost performance when they are not required.

During performance testing, the results from Amazon EC2 turned out to be more predictable and consistent than testing on a local computer. Since EC2 instances are virtualized and managed for reliability, they are less susceptible to unexpected degradation in system or network performance. Thus, the remote testing environment is more similar to a real-world business environment, but may not model all possible usages, such as peer to peer applications.
6. Related Work

6.1 RedSocks and uRedSocks

Our work draws heavily on the contributions of Haungs [7] and Liang [11]. Haungs defined and implemented the RedSocks protocol in the Linux kernel to provide new socket redirection primitives, and Liang extended that to create a user space version (uRedSocks) while adding a novel protocol negotiation algorithm that allowed for legacy client support. Our contributions to uRedSocks allow for usage beyond failover and load balancing, to include support for composable services, passive monitoring, and fast failover.

6.2 Multi-homing in SCTP

In 2000, RFC 2960 [18] proposed a new transport protocol called Stream Control Transmission Protocol. SCTP offers a number of advantages to TCP, including built-in failover support using multi-homing and heartbeats.

In order to use SCTP’s multi-homing functionality, each endpoint may bind to multiple transport addresses. The number of paths available between the two endpoints is equal to the product of the number of transport addresses used by each endpoint. For example, there are four possible paths in the configuration show in figure 6.1 (adapted from Ravier et al [16]).
Depending on network topology, multi-homing provides fault tolerance as long as at least one path remains unbroken.

While multi-homing helps protect against network faults, it doesn’t provide a mechanism for redirection if one the hosts itself becomes stressed or fails. With uRedSocks, however, there are at least three ways that failover can occur even when one of the hosts fails:

1. Servers can use the promote operation if they have split the socket and detect deteriorating conditions;
2. Servers can use explicit redirection if they detect deteriorating conditions; and
3. Connections can be redirected automatically using implicit redirection when heartbeats fail.

### 6.3 Intermediaries

Informational RFC 3135 [5] provides a very useful overview of the types of performance enhancing proxies (intermediaries) currently in use, some of which could use uRedSocks for insertion and removal. Informational RFC 3752 [4] provides use cases for more intermediary types, and RFC 3835 [3] describes an architecture that could be used when implementing intermediaries. Mizikovsky et al describe an approach for adding intermediaries to be used by wireless mobile devices, but the protocol changes would not
be backwards compatible with legacy systems, and they have not yet published an attempt to implement their approach [9].
7. Conclusion

The functionality provided by uRedSocks enables applications to enjoy improved durability and flexibility without sacrificing support for legacy clients. Retrofitting existing systems to support uRedSocks is trivial, and the new socket operations presented here function similarly to existing system calls.

Clearly, using uRedSocks incurs a performance penalty, due in large part to heartbeat processing and protocol negotiation. In its current form, the protocol is probably not a good choice for time sensitive short-lived connections transferring small amounts of data. However, uRedSocks is a practical choice for applications with longer-lived connections, which stand to gain from the failover and mediation options it provides.
8. Future Work

One shortcoming in the current implementation of remove, merge, and promote is that there is no way for the server to specify which intermediary or secondary server is to be affected. In order to allow for maximum flexibility, the protocol and implementation should be extended to allow for specification of which intermediary should be removed, which secondary server should be disconnected, and which secondary server should be promoted.

Heartbeat functionality can also be extended in at least two ways. First, uRedSocks should be extended such that heartbeat processing continues after a split operation, allowing for implicit failover after a split. Second, heartbeat processing should be made optional. Not all applications require implicit failover, so they should not be required to pay the associated performance penalty.

Finally, the new operations introduced in uRedSocks should be folded in to the kernel RedSocks implementation. This would allow for reuse of existing system calls and could open the door for further performance tuning.
Last visited November 4, 2010.


