CFHT Socket I/O Library

Sidik Isani

2002 March 18th
Last revised: 2004 May 18th (v1.6)

This document is available on the Web at: http://software.cfht.hawaii.edu/sockio/

Version 1.6 2004-5-18 - Made it safe to call sockserv_del_client() inside sockserv_run().
Version 1.5 2003-6-2 - Added client disconnect_hook and finalize sockclnt half of libsockio.
Version 1.4 2003-5-27 - Documented the new send-binary hook and client-delete features. Added full prototypes to the API documentation.
Version 1.3 2002-4-16 - Documented the receive hook-disconnect feature (2.2.2).
Version 1.2 2002-3-30 - New sections 1.2, 1.3, and 3.x added.
Version 1.1 2002-3-20 - Corrected sbuf_t state diagram to show SBUF_RECEIVED going directly to write() instead of SBUF_EMPTY. Revised last server flow chart and text above it to show which steps happen inside client_recv_hook() versus client_send_hook().
Version 1.0 2002-3-18 - First version. Lowest layers of server-side done.

Abstract

`libsockio.a` is a library for clients and servers that need to pass messages to each other over a socket. It is designed specifically for the needs of the NEO status server, and other servers which use "line-based" communication protocols. If used as intended, it is possible to create very fast and responsive single-threaded multi-client socket servers with only a few lines of code needed to manage the socket communication. A second part of the library contains functions to help implement clients, or client protocol libraries to communicate with the server.

Contents

1 Overview **3**

1.1 Messages 3

1.2 Library Components 3

1.3 Example: Environment Server and Client 4

2 Server-Side 5
2.1 sockserv_t* sockserv_create(const char* servname); 6
2.2 void sockserv_destroy(sockserv_t* sockserv) 6
2.3 int sockserv_run(sockserv_t* sockserv, int timeout_hundredths) 6
2.4 void sockserv_del_client(sockserv_t* sockserv, void* cinfo) 6
2.5 void client_recv_hook(void* cinfo, char* message_in_out) 7
2.5.1 client_recv_hook() and disconnect commands 8
2.6 void client_send_hook(void* cinfo, char* message_out) 9
2.7 void client_send_binary_hook(void* cinfo, char* data, int* bytes) 10
2.8 void* client_add_hook(unsigned char ip_addr[4]) 11
2.9 void client_del_hook(void* cinfo, char* buffer) 13

3 Client-Side 14
3.1 sockclnt_t* sockclnt_create(const char* hostspec, int io_timeout_seconds) 14
3.2 disconnect_hook(void* userdata) 14
3.3 reconnect_hook(void* userdata) 14
3.4 void sockclnt_destroy(sockclnt_t* sc) 14
3.5 void sockclnt_send(sockclnt_t* sc, const char* message) 14
3.6 const char* sockclnt_recv(sockclnt_t* sc) 15
3.7 const char* sockclnt_check(sockclnt_t* sc) 15

A Internal Server-Side Half-Duplex Buffer Mechanism - sbuf_t 16

List of Figures

1 libsockio components 3
2 libsockio and example client and server 4
3 Internal Flow Diagram for Server 5
4 echoserv source code 5
5 envserv source code 7
6 disconnect command 8
7 Flow Diagram for Server with Mailbox System 9
8 Flow Diagram for Server with Mailbox System and Long Replies 10
9 Using client_add_hook 11
10 Using client_add_hook for security 12
11 Using client_del_hook 13
12 Possible States for an sbuf_t 17

1 Overview

1.1 Messages

The primary purpose of libsockio is to pass messages between programs. The content of the messages depends on the specifics of the protocol implemented on top of libsockio, but libsockio does impose some restrictions on this protocol.

Messages passed back-and-forth across the socket are octet (byte) data with a pre-defined maximum size (defined in the header file as SBUF_SIZE, currently 32768 bytes) and are line-based (terminated with '\n' or '\0'). The terminator is added by the library on outgoing messages, and removed by the library from incoming messages. Neither '\n' nor '\0' may appear inside the message itself. Trailing '\r' characters are removed from the end of a message, if present. This is to allow the alternate "\r\n" newlines sent by some "telnet" clients. All other characters are legal (as far as libsockio is concerned.) The safest approach is to encode all bytes less than ASCII 32 (space). Actual encoding of messages is outside the scope of this library.

Another restriction placed on the protocol is that the each message sent by a client to the server *must* have a response. The client can never send more than one message at a time, without an intervening message back from the server. The messages from the server to the client can be asynchronous, or multi-line (if your protocol supports it), but communication in the other direction is more limited. These restrictions are a result of the design of libsockio.

As of version 1.4, a new facility has been added to allow a server to send binary data in response to a client query. Client and server must negotiate the size of this transfer as part of their protocol, and the client API provided by sockio does not handle reading this binary response itself. Messages from the client-to-server direction are always limited to single, line-based messages terminated with '\n' or '\0'.

1.2 Library Components

Figure 1 shows each of the C files in the library and the main functions each one exports. A server would typically use the library directly (i.e., it would link with `-lsockio`) while a client might use a protocol library to facilitate communication with a specific type of server. If this protocol library is built on top of the client-side of libsockio, a client would need to link with both (i.e., `-lprotocol -lsockio`). See section 1.3 for and example of this.

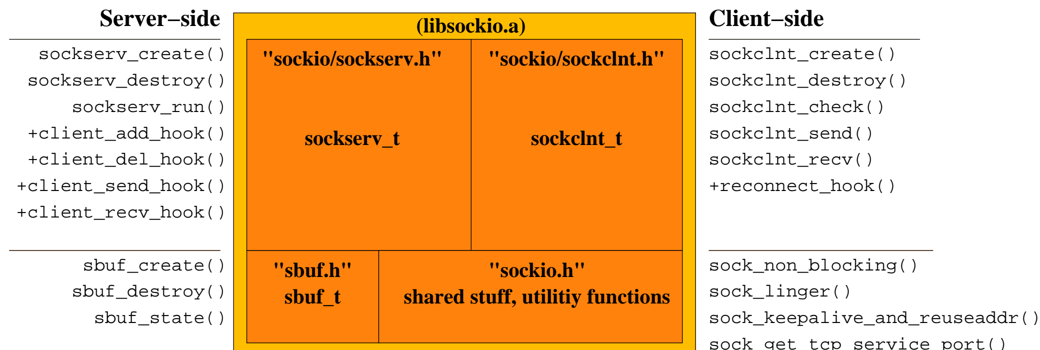

Figure 1: libsockio components

1.3 Example: Environment Server and Client

Figure 2 shows a server (the `envserv` example in the project directory) and two clients based on a an envserv-specific protocol library called `libssenv.a`. The clients simply set and get values in the server, by making calls to libssenv. More clients could be created to interact with this server just by using the three calls (ss_initenv, ss_getenv, ss_setenv) in `libssenv.a`.

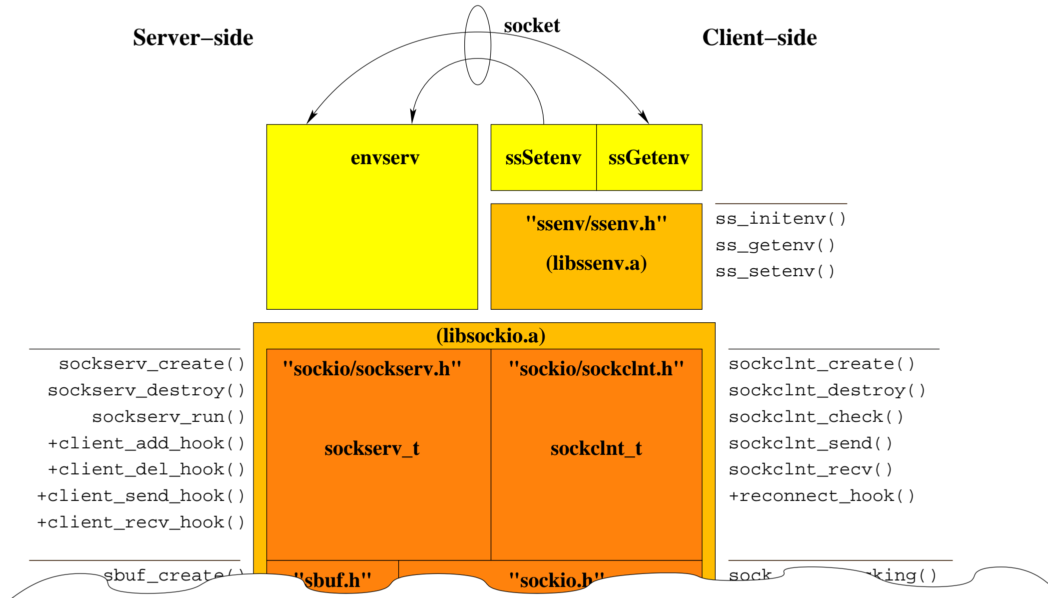

Figure 2: libsockio and example client and server

2 Server-Side

The calls in libsockio which are used to build a server are built on an internal buffer and state machine called sbuf_t (for details on how it works, see the appendix). The library provides a routine which creates a `sockserv_t` and a function which runs the state machine for you. The minimum logic required to process i/o is shown in figure 3.

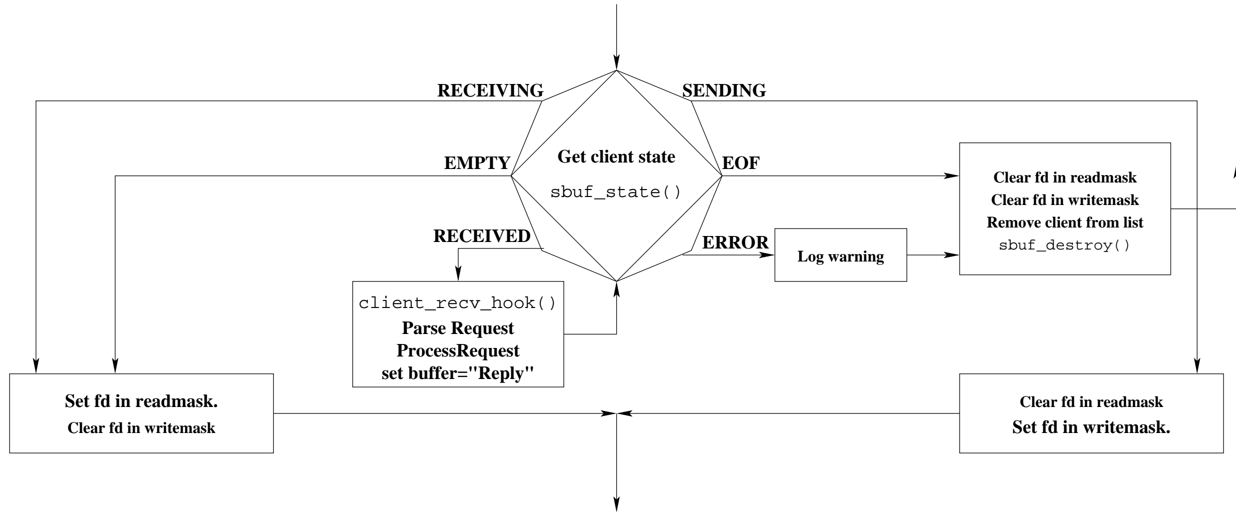

Figure 3: Internal Flow Diagram for Server

There are only three functions to create, run, and destroy the sockserv_t. A minimal socket server can be implemented just by calling sockserv_create() followed by sockserv_run() in an endless loop. The following example is in the sockio project directory, and is called `echoserv`:

```
#include "sockio/sockserv.h"

int
main(int argc, const char* argv[])
{
   sockserv_t* echoserv = sockserv_create("5252");

   if (!echoserv) exit(1);
   for (;;) sockserv_run(echoserv, SOCKSERV_WAIT);
   exit(0);
}
```

Figure 4: echoserv source code

2.1 sockserv_t* sockserv_create(const char* servname);

The sockserv_create() function takes a single string argument that is either the name or number of a TCP port on which to listen. The owner of the program must have permission to use this port. Ports less than 1024 can only be used by the superuser.

2.2 void sockserv_destroy(sockserv_t* sockserv)

The pointer returned by sockserv_create() should be passed back to sockserv_destroy() before your program exits, especially if you use the client_del_hook() feature. Both of these are shown in greater detail in examples below.

2.3 int sockserv_run(sockserv_t* sockserv, int timeout_hundredths)

The pointer returned by sockserv_create() is passed back to sockserv_run(), along with a timeout value (in hundredths of seconds). The previous example uses the special value SOCKSERV_WAIT, which will wait indefinitely for a new client to connect, or for an existing client to send a new request. To add a sockserv back-door to an existing program that has some kind of polling loop, sockserv_run() can also be called with SOCKSERV_POLL which returns immediately after one round of servicing clients.

Normally, all actions are triggered from within sockserv_run() using the "client_*_hook()" functions described below. Each time a client has sent a complete line, your receive hook gets called. Any time it is possible to send a line to a client (which is almost every time sockserv_run() happens usually) your send hook gets called. This means the client must always be the one to initiate a disconnect.

2.4 void sockserv_del_client(sockserv_t* sockserv, void* cinfo)

The sockserv_del_client() function is not needed by most typical socket servers, but in some cases, the server may wish to force a client to be disconnected (after some timeout period, for example). If such behavior is desired, the server *must* assign unique client info pointers to each client, using the client_add_hook() described below. Then the server must pass that same key (cinfo) to sockserv_del_client() at the time it wishes to force that client to be disconnected the next time your program calls sockserv_run().

Prior to version 1.6 of the sockio library, any existing client_del_hook() that might have been set up would get called during sockserv_del_client() itself. Now, this routine only marks the client for deleting during the next call to sockserv_run(). Any client_del_hook() will be called at that time (and no other hooks will get called in the intervening time.)

As a special case, if cinfo is NULL, this function causes all of the clients to be disconnected on the next call to sockserv_run(), without destroying the server itself.

2.5 void client_recv_hook(void* cinfo, char* message_in_out)

In order to make your server do something useful, a call-back function must be provided in client_recv_hook(). This function returns void, and has an arbitrary pointer as the first argument (described later) and a pointer to a buffer. Whenever a client sends in a request, this function will be called, and the client's message will be in the buffer. The call-back function should process the request, write a (null terminated) response back into the same message_in_out buffer, and return. The response will go out to the client immediately, if possible. (But if downstream FIFOs are full, it will go out on a subsequent call to sockserv_run(), but this is completely hidden from the caller.)

Using only a client_recv_hook(), it is possible to change the echoserv example into a server which shares its own environment (**envp) with any number of clients. These clients can all read and write the same environment variables by writing messages to the socket. This example is also in the project directory, and is called `envserv`:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "sockio/sockserv.h"

void
client_recv(void* cinfo, char* buffer)
{
   char* p;
   if (strchr(buffer, '='))
   {
      putenv(strdup(buffer));
      sprintf(buffer, ". OK");
   }
   else
   {
      p = getenv(buffer);
      if (p) sprintf(buffer, ". %.999s", p);
      else   sprintf(buffer, "! no such variable");
   }
}

static sockserv_t* envserv = NULL;

static void cleanup(void) { sockserv_destroy(envserv); }

int
main(int argc, const char* argv[])
{
   sockserv_t* envserv = sockserv_create("5253");

   if (!envserv) exit(EXIT_FAILURE);
   atexit(cleanup);
   envserv->client_recv_hook = client_recv;
   for (;;) sockserv_run(envserv, SOCKSERV_WAIT);
   exit(EXIT_SUCCESS);
}
```

Figure 5: envserv source code

The example also shows the proper way to call the counterpart to sockserv_create(), sockserv_destroy(). Without this cleanup function, some operating systems may not allow you to start a new server on the same port number for some period of time. To test your server, connect to the port with a telnet client:

```
> telnet localhost 5253
foo
! no such variable
foo=hello 123
. OK
foo
. hello 123
```

2.5.1 client_recv_hook() and disconnect commands

The server may also wish to implement a close-connection command. The actual command could be something like "exit", "quit", "logout" but depends on the protocol you decide to implement in your server. When such a command is received, write the empty string (set buf[0] to the NUL character, `buf[0]='\0'`).

Normally, the server must write a (non-empty) response if it provides a receive hook. If the response is empty, this condition is treated the same as if an EOF condition had occurred (for example if the client closed the connection.) A common reason for implementing a disconnect command is for convenience during manual connections made with a telnet client.

The program segment in figure 6 would add a "quit" command to a server's receive hook:

```
static void
client_recv(void* cinfo, char* buffer)
{
   /*
    * Allow clients to disconnect themselves.
    */
   if (!strcmp(buffer, ``quit'') ||
       !strcmp(buffer, ``bye'') ||
       !strcmp(buffer, ``exit'') ||
       !strcmp(buffer, ``logout''))
   {
      buffer[0] = '\0'; /* disconnect the client */
      return;
   }
   . . .
```

Figure 6: disconnect command

2.6 void client_send_hook(void* cinfo, char* message_out)

If the server wishes to send asynchronous messages, it may do so whenever the socket is ready for writing, and there is no partially received message in the buffer. This function will be called almost every time that sockserv_run() happens, so if there is nothing to send, this function should calculate that quickly and efficiently, and just return without writing anything in message_out. If there is a message to go out, it should simply be copied into the buffer. The following figure shows how a client could be notified that something which the server is monitoring for it has changed. This uses a mailbox system, forcing the client to come back with a request to actually read the change. (The actual message could be sent instead, if the client can handle it and keep up.) Figure 7 shows the additional logic needed to add asynchronous notification.

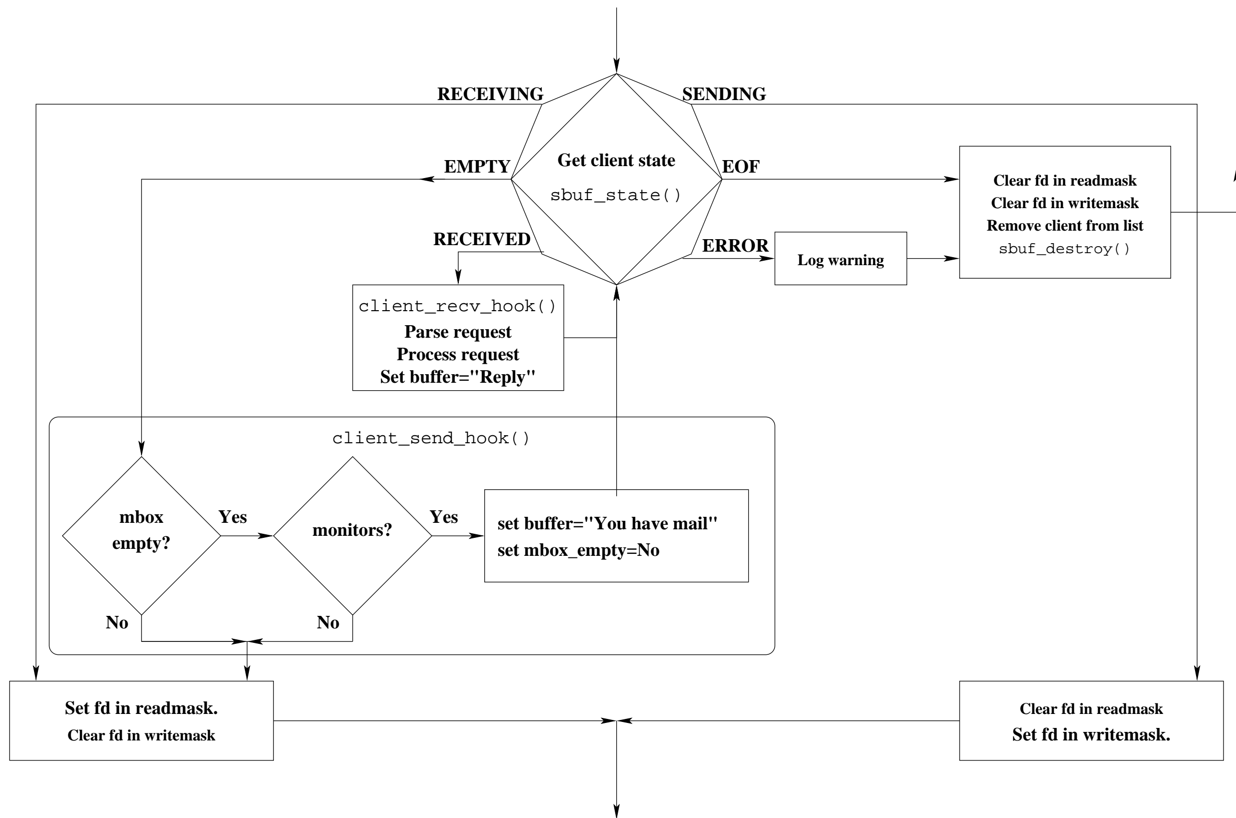

Figure 7: Flow Diagram for Server with Mailbox System

For better throughput, multi-line transfers in the direction from the server to the client may be useful (multi-line transfers in the other direction are not supported.) As the client is ready to accept each line of a multi-line response, the client_send_hook() will be called. For the first line to be processed, the client_recv_hook() can usually set things up and then pass control to the client_send_hook() to get things started.

Typically, multi-line responses will be sent in one “turn” of the client, because the underlying network FIFOs are generally large enough to hold several messages. The buffer in libsockio can only hold one message at a time, however, so it is rare but possible that an entire multi-line response will not be sent at once. In this case, there is no difference in the way client_send_hook() gets called, but it should not make any assumptions about the state of anything which might have been changed by intervening transactions with other clients.

Figure 8 shows the most complete usage of sockserv_t, including support for both a mailbox and multi-line reply messages.

Both the mailbox feature and multi-line responses can be implemented by creating a second hook, client_send_hook(),

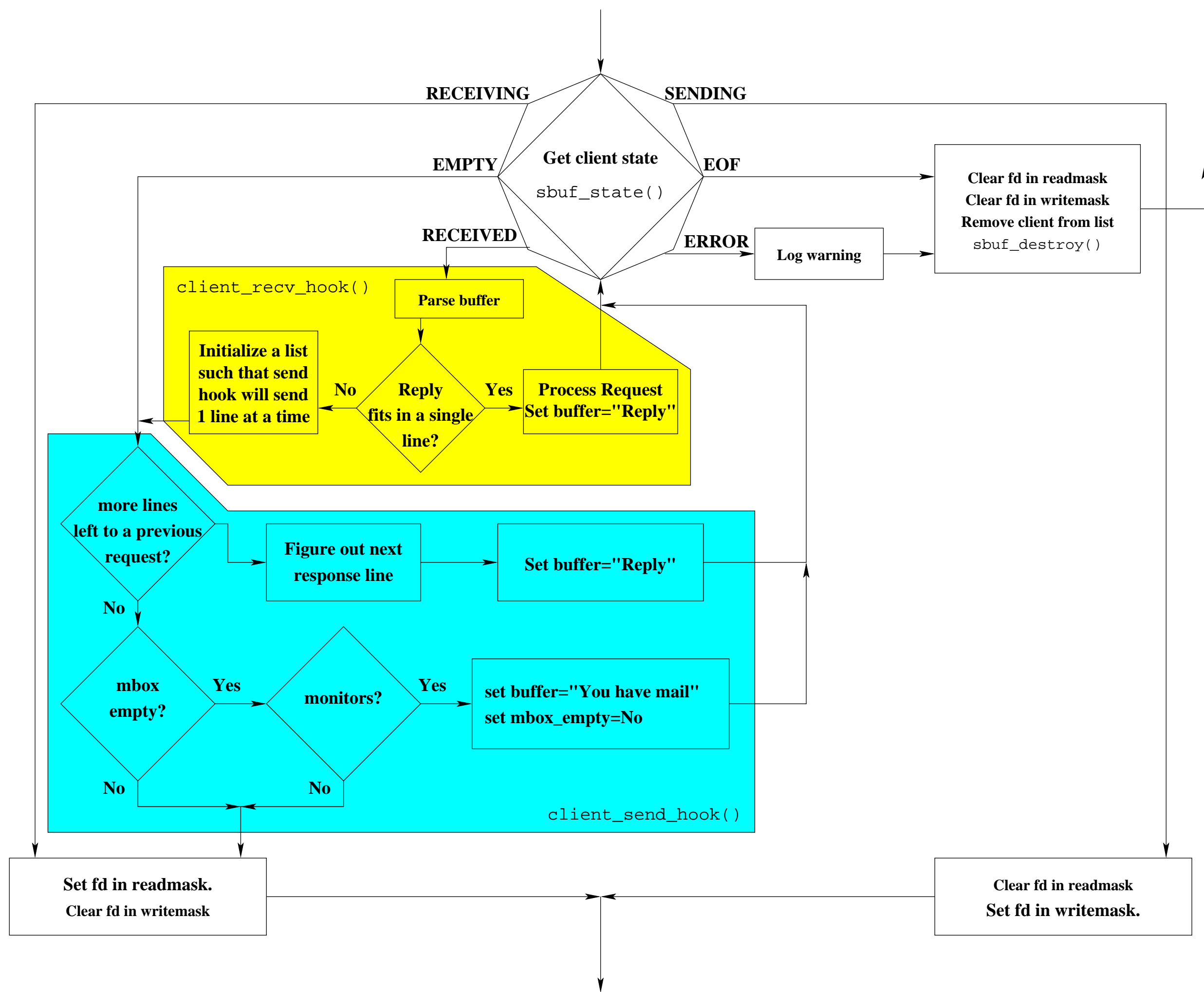

Figure 8: Flow Diagram for Server with Mailbox System and Long Replies

which will be called whenever the server has the opportunity to send something out to the client. This call-back will typically check for previous responses that are still in progress, mailbox flags, time stamps, etc., and if it determines it wants to send anything else to a client, it does so by writing the message to the buffer (exactly the same way as the client_recv_hook() call-back would do).

2.7 void client_send_binary_hook(void* cinfo, char* data, int* bytes)

This is a special version of the send hook for use in implementing protocols that require a server to send binary data responses to a client (binary communications in the other direction, from the client to the server, are not possible.) This hook is used in exactly the same way as the normal line-based send_hook, except that a maximum of SBUF_SIZE (32KB) of binary data can be copied into the data buffer, and a byte count must be specified in bytes. As with the send_hook(), if send_binary_hook() does nothing but return, nothing is sent.

If your server defines both a send_binary_hook() and a send_hook(), the binary hook is always tried first.

An example of a server that uses binary transfers can be found in `/cfht/src/medusa/espadons/fli/fliserv.c`.

2.8 void* client_add_hook(unsigned char ip_addr[4])

The send_hook is usually not very useful unless you know *which* client is calling back. So you probably also want make use of the client_add_hook() in this case. Inside this call-back, you may allocate a per-client structure of information. The first piece of information you have about the client is its IP address, passed as four bytes in the first argument of client_add_hook(). Other information about the client can be filled in later, as transactions proceed with this client, since each of those will cause the pointer to your structure to be passed back as the first argument to client_recv_hook() and client_send_hook(). All client_add_hook() must do is allocate the structure and return the pointer. Here is an example of a simple client_add_hook call-back.

```
typedef struct
{
   BOOLEAN mbox_empty;
   BOOLEAN long_reply;
} client_info_t;

/*
 * Returns per-client data to store with the new client, or
 * returns NULL to reject the client.
 */
void*
client_add(unsigned char remote_ip[4])
{
   client_info_t* cinfo;

   cinfo = (client_info_t*)malloc(sizeof(client_info_t));
   memset(cinfo, 0, sizeof(cinfo));
   cinfo->mbox_empty = FALSE;
   cinfo->long_reply = FALSE;
   return cinfo;
}

.
.
.

int main(int argc, const char* argv[])
{
   . . .
   sockserv->client_add_hook = client_add;
   . . .
}
```

Figure 9: Using client_add_hook

The client_add_hook() can also be used to implement some basic security. If you wish to reject all connections except those from specifically allowed IP addresses, the client_add_hook() can look at the IP address and return NULL to let sockserv_run() know that this client should be rejected. Here is how the `client_add()` function from the above example would look with an IP-address check added:

```
/*
 * Returns per-client data to store with the new client, or
 * returns NULL to reject the client.
 */
void*
client_add(unsigned char remote_ip[4])
{
   client_info_t* cinfo;

   /*
    * Hardcoded to only allow connections from:
    * 127.*, 128.171.80.* and 128.171.83.*
    */
   if (!(remote_ip[0]==127 ||
         (remote_ip[0]==128 && remote_ip[1]==171 &&
          (remote_ip[2]==80 || remote_ip[2]==83))))
   {
      fprintf(stderr,
              "warning: rejecting nonlocal client"
              " from %d.%d.%d.%d\n",
              (int)remote_ip[0], (int)remote_ip[1],
              (int)remote_ip[2], (int)remote_ip[3]);
      return NULL; /* Reject client */
   }

   fprintf(stderr, "client accepted from remote_ip %d.%d.%d.%d\n",
          (int)remote_ip[0], (int)remote_ip[1],
           (int)remote_ip[2], (int)remote_ip[3]);

   cinfo = (client_info_t*)malloc(sizeof(client_info_t));
   memset(cinfo, 0, sizeof(cinfo));
   cinfo->mbox_empty = FALSE;
   cinfo->long_reply = FALSE;
   return cinfo;
}
```

Figure 10: Using client_add_hook for security

2.9 void client_del_hook(void* cinfo, char* buffer)

Whenever an end-of-file or error condition occurs with a client, this clean-up callback can be used to free the `cinfo` client info allocated by client_add_hook(). An example:

```
void
client_del(void* cinfo, char* buffer)
{
   fprintf(stderr, "Cleaning up client\n");
   free(cinfo);
}

.
.
.

int main(int argc, const char* argv[])
{
   . . .
   sockserv->client_del_hook = client_del;
   . . .
}
```

Figure 11: Using client_del_hook

The buffer argument in this callback is not used.

3 Client-Side

Just as the server side calls are used to build a server, this part of `libsockio` can be used to build a protocol library for your server. The API for `envserv`, called `libssenv` is an example. That, in turn is used by the `ssGetenv` and `ssSetenv` client programs which talk to `envserv`. (Refer to figure 2 to see how the pieces depend on each other.) The source files `sockclnt.h` and `sockclnt.c` contain calls which can be used to build an API library like `libssenv`.

3.1 sockclnt_t* sockclnt_create(const char* hostspec, int io_timeout_seconds)

This returns a pointer to a `sockclnt_t` that must be passed back to the other functions below. A connection attempt is made immediately. Each host in hostspec (see below) is tried at most one time by this call. Any subsequent calls to other sockclnt routines will (indefinitely) keep trying to re-establish the connection by cycling through the list (unless you define a disconnect_hook() and exit the program.)

The hostspec argument must be a string of the form `"hostname:port,hostname2:port,..."`. The hostname can be either an IP address or resolvable host name, and port can be either a port number or resolvable service name. If the first hostname:port specification fails, sockclnt_create() and any later attempts to reconnect will cycle through all of the choices.

The timeout for a connection, a read (sockclnt_recv), and a write (sockclnt_send) is set in the io_timeout_seconds parameter. 15 seconds is a reasonable value, but the correct setting depends on your protocol and function of the server.

3.2 disconnect_hook(void* userdata)

3.3 reconnect_hook(void* userdata)

If this happens, a `reconnect_hook` function in the user code will be called after the connection is back up. This function may exchange messages with the server to rebuild any state information that was lost when the old connection broke. If `reconnect_hook` is not needed, do not set it.

3.4 void sockclnt_destroy(sockclnt_t* sc)

This closes the underlying file descriptor and frees resources allocated by sockclnt_create().

3.5 void sockclnt_send(sockclnt_t* sc, const char* message)

Reconnects and/or sends a message to the server. If a previous receive operation has failed, `sockclnt_send` will reconnect to the server. If such a reconnection was needed, and if a user `reconnect_hook` has been defined, it will be called before the message is sent so it has the chance to renegotiate the current state with the server. `reconnect_hook` will also be called if “message” can not be sent completely. This function will try to reconnect to the server and send the message indefinitely. The only way to give up is to implement a `reconnect_hook` that raises a signal, exits, or never returns. A warning message will be printed to stderr when a retry is in progress.

By passing NULL for “message”, this function can be used to ensure a connection to the status server. For example, socket clients which use `sockclnt_check` or `sockclnt_recv` will want to use `sockclnt_send(sc, NULL)` any time a NULL is returned from these functions. Instead of passing a NULL, it may be necessary to resend the most recent protocol command.

If “message” is not NULL, the design of the server-side of sockio, *requires* that it be followed by one or more calls to receive a response. This is because the server must generate a response to all messages that it receives (except for disconnect, but even in that case the receive function should be called to get the end-of-file.)

3.6 const char* sockclnt_recv(sockclnt_t* sc)

This waits up to io_timeout_seconds (or indefinitely if timeout is 0) for a response to the previously sent message. If the "response" is end-of-file or a timeout happens, then NULL is returned when the server closes the socket. A NULL is also returned in case of any error. In any of the cases that NULL is returned, a subsequent call to `sockclnt_send` will cause the connection to be re-established. This receive call never handles reconnecting though. (It wouldn't make sense, because if the socket closed unexpectedly while reading a response, the request will have to be re-sent anyway, and the client code must handle that.)

3.7 const char* sockclnt_check(sockclnt_t* sc)

Like sockclnt_recv, but returns NULL immediately if no messages are waiting.

A Internal Server-Side Half-Duplex Buffer Mechanism - sbuf_t

An `sbuf_t` manages per-client buffer space for socket i/o. It is implemented as a small state machine which has six (6) states, not including the hidden states of being in a read() or write() call. Transitions between the states are triggered in sockserv_run() by the library. There are only two operations done on the sbuf_t:

1. Send a message to the client. This is done simply by writing directly into the buffer space associated with the `sbuf_t`, `(char*)(sbuf->buffer)`. This must only be done if the current state is SBUF_EMPTY. In all other states, it is not possible for the server to send a message to the client.

   WARNING: Writing data into the buffer when the state is not SBUF_EMPTY will cause unpredictable results!

2. Wait for a complete message to be received from a client. When the state becomes SBUF_RECEIVED, a message is waiting in the same buffer which is used to send a message. Once in SBUF_RECEIVED, the caller must do whatever is needed with the data (parse an incoming command as the case might be with a typical socket server.) Before calling sbuf_state() again, the caller must write a response into the same buffer. All received messages must have a response. If the call does no processing at all, this model ends up implementing an echo-server since the received message, still in the buffer, is simply sent back to the client (see the echoserv example).

`sockserv_run()` essentially calls `sbuf_state()` in a loop, which makes the state machine run (and makes data go in or out of the socket.) `sbuf_state()` returns only when a stable or blocking state has been reached. The resulting state is always the return of `sbuf_state()`, and sockserv_run() uses this in a switch statement to decide what to do next.

While the socket may be able to handle communication in both directions at once, note that this state machine cannot. If there happens to be a partially received message in the buffer when the server wishes to send something out, the receiving message must first come in completely and then be processed, before anything can go out. In this sense, the communication channel is ”half-duplex”.

This is prototyped in `sbuf.h` and implemented in `sbuf.c`. The complete state model is shown in figure 12.

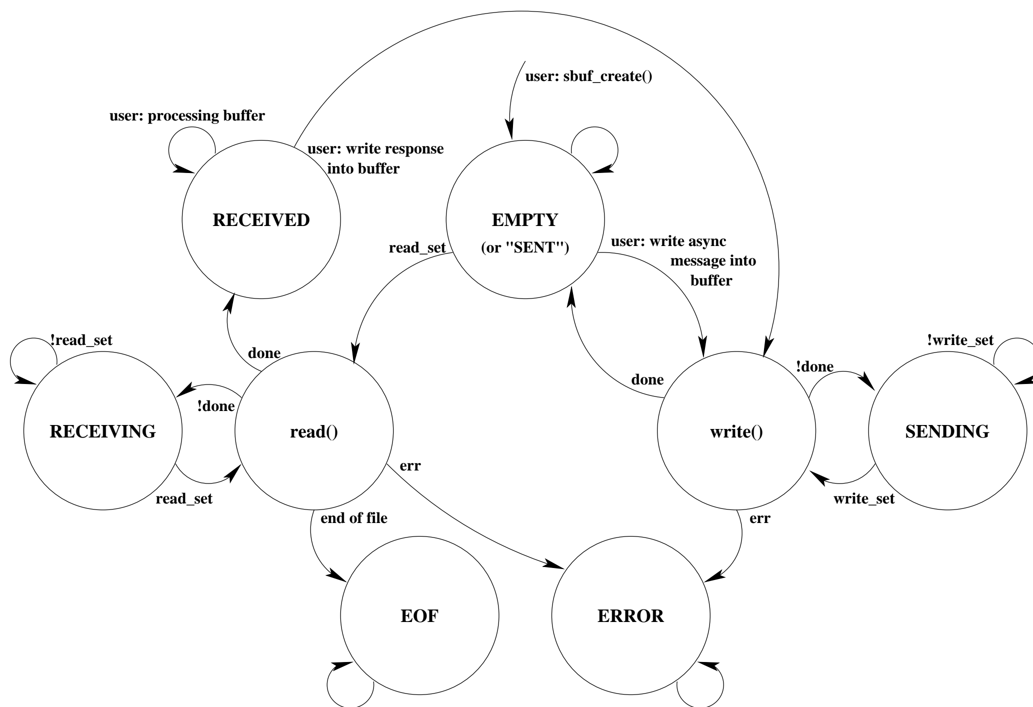

Figure 12: Possible States for an sbuf_t