An Introduction to Client-Server Programming with Sockets

Rob Shillner

March 14, 1997

1 Introduction

1.1 Distributed applications

A distributed application is a collection of running programs which cooperate to accomplish some task. Distributed applications are designed to work over a network, such that the component programs need not run on the same machine.

One possible software architecture for a networked distributed application is the *client-server* model. In this model, a central program, known as the *server*, runs on some machine. The other programs are called the *clients*; each client runs on machine possibly different from the one which houses the server. Typically, the client programs transfer information (such as input from human users) to the server program. The server processes the inputs it receives from the client programs and sends information back to the clients for further processing (and possible display on users' terminals).

This document details the implementation of a minimal distributed application: a "chat" facility. The application is composed of two C programs, `server` and `client`. The server listens at a particular network address, accepting connections from users running the client program. The client program transfers text between the user's terminal and the server. When the server receives a line of text from a client, it sends copies of the text to all other connected clients.

1.2 Sockets

The communication between the server and client programs occurs using *sockets*. Communication follows these steps:

1. The server creates a socket using the system call `socket`. We'll call this socket the *master socket*.
2. The server binds the master socket to a network address, using the system call `bind`.
3. The server starts listening for client connections on the master socket, using the `listen` system call.
4. The client creates a socket of its own, using `socket`. This is the *client socket*.
5. The client attempts to connect its socket to the master socket, using the system call `connect`.
6. The server notices the connection attempt and accepts the connection using the system call `accept`. This returns a new socket to the server, the *server socket* for that client.
7. The server and client may now communicate with each other, using the `read` and `write` system calls on the server and client sockets.
8. When either the server or client decides to terminate the connection, it issues the `close` system call on its socket. The other party notices the terminated connection and closes its socket as well.
9. When the server no longer wishes to accept new connections, it uses `close` to shut down the master socket.

This is not the only way to use sockets, but it is perhaps the easiest way; this method provides reliable, ordered transfer of byte streams between two unrelated processes. Coding with sockets is much easier to understand if an example is handy, so the rest of this document is the full source code for the "anonymous chat" application.

2 The server

The server looks like a normal C program, with included files followed by type declarations, global variables and the `main` function.

"server.c" 2a ≡
```
⟨Server include files 2b⟩
⟨Server type declarations 3c⟩
⟨Server global variables 3a, ... ⟩
⟨Server main routine 2d⟩
◇
```

The server includes the standard I/O header and some socket-related headers.

⟨Server include files 2b⟩ ≡
```
#include <stdio.h>
⟨Socket headers 2c⟩◇
```
Macro referenced in scrap 2a.

These are the headers needed to make Internet socket communication work. See the manual pages of the socket-related system calls for more information about when each header must be included.

⟨Socket headers 2c⟩ ≡
```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <netinet/in.h>◇
```
Macro referenced in scraps 2b, 7a.

After initializing itself, the server's `main` routine loops forever, accepting and dealing with inputs it gets from its sockets.

⟨Server `main` routine 2d⟩ ≡
```
main(int argc, char **argv)
{
  ⟨Initialize the server 2e⟩
  for (;;) {
    ⟨Accept and handle inputs 4c⟩
  }
}◇
```
Macro referenced in scrap 2a.

2.1 Initializing the server

In our simplified server, initialization consists of only two tasks: setting up the master socket and initializing the user list.

⟨Initialize the server 2e⟩ ≡
```
⟨Initialize the master socket 3b⟩
⟨Initialize the user list 4b⟩◇
```
Macro referenced in scrap 2d.

2.1.1 Initializing the master socket

We keep the master socket in a global variable called `masterSocket`. Note that `masterSocket`, like every socket, is of type `int`, just as file descriptors are of type `int`. In fact, many of the file descriptor system calls, such as `read`, `write`, `select` and `close` also work on sockets.

⟨Server global variables 3a⟩ ≡

```
int masterSocket;
◇
```

Macro defined by scraps 3a, 4a.
Macro referenced in scrap 2a.

To initialize the master socket, we create the socket and bind an address to it, then start listening at it. The `socket` call creates and returns `masterSocket`; the arguments to that call indicate a socket for the Internet domain (`AF_INET`), of the reliable stream type (`SOCK_STREAM`) using the default protocol (0). Next we need to specify the address to which the socket should be bound; to do this we fill in some fields of a `sockaddr_in` structure. The `sin_port` field specifies the port to which the socket will be bound; we use port 9001, but almost any positive integer will do, so long as the client program is able to supply the same integer when it connects to the server. The `bind` call actually binds the socket to the port. The `listen` call sets the server listening for attempted connections at the port. The argument of 5 specifies that up to five clients may be waiting for a connection at one time.

⟨Initialize the master socket 3b⟩ ≡

```
struct sockaddr_in serverAddress;

masterSocket = socket(AF_INET, SOCK_STREAM, 0);
if (masterSocket == -1) {
  perror("server");
  exit(-1);
}
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = INADDR_ANY;
serverAddress.sin_port = htons(9001);
if (bind(masterSocket, (struct sockaddr *)&serverAddress,
         sizeof(struct sockaddr_in))) {
  perror("server");
  exit(-1);
}
listen(masterSocket, 5);
◇
```

Macro referenced in scrap 2e.

2.1.2 Initializing the user list

Now that the master socket is set up, we need to create the server's internal data structure for keeping track of what users have connected. We keep a doubly-linked list containing one `User` data structure for each connected user. Each `User` structure keeps track of the server socket for that user. In a larger application, the `User` structures would hold other kinds of information about a user, such as a name, password, etc.

⟨Server type declarations 3c⟩ ≡

```
typedef struct user {
  int socket;
  struct user *next, *prev;
} User;
◇
```

Macro referenced in scrap 2a.

We keep the list of currently-connected users in the global variable `userList`.

⟨Server global variables 4a⟩ ≡
```
User *userList;
◇
```
Macro defined by scraps 3a, 4a.
Macro referenced in scrap 2a.

At startup, of course, there are no users, so `userList` is `NULL`.

⟨Initialize the user list 4b⟩ ≡
```
userList = NULL;◇
```
Macro referenced in scrap 2e.

2.2 Accepting and handling inputs

Once the server has been initialized, it goes into an infinite loop in which it repeatly accepts and handles inputs. There are two types of input to the server: new connections to the master socket, and data arriving from already-connected users on individual server sockets. Since the server must handle inputs from so many different sockets, and since the `read` and `accept` calls *block* when no data is ready to be read, the server must attempt to service only those sockets which need immediate attention.

To multiplex between many sockets, the server creates a variable `ready` of type `fd_set`. An `fd_set` is a bit vector, with each bit representing a socket. To use an `fd_set`, a program sets the bits which correspond to the sockets it is interested in. When the `fd_set` is passed to the `select` system call, all the bits will be cleared except the bits corresponding to sockets which have data waiting to be read. Then the program can read from or accept connections on the sockets whose bits are still set.

Thus, to accept inputs, we use `select` to find which sockets are ready, setting `numReady` to the number of ready sockets. Then, if there is at least one ready socket, we attempt to accept new connections and read input from existing users.

⟨Accept and handle inputs 4c⟩ ≡
```
int numReady;
fd_set ready;
⟨Select sockets which are ready for reading 5a⟩
if (numReady > 0) {
  ⟨Accept a new connection if masterSocket is ready 5b⟩
  ⟨Take input from users who are ready 6a⟩
}◇
```
Macro referenced in scrap 2d.

To select the sockets that are ready, we first use the system-supplied macro[1] `FD_ZERO` to zero out `ready`. Then we use `FD_SET` to set the bit for the master socket. Then, for each user in the user list, we use `FD_SET` to set the bit corresponding to the user's socket. Along the way, we find `maxSocket`, the highest-numbered socket we have.

When `ready` is ready, we call `select`, passing the number of entries in `ready` which `select` should consider (that is, `maxSocket + 1`). The call to `select` returns the number of sockets that are ready, and it clears the bits in `ready` corresponding to "unready" sockets. Also, the `select` call blocks[2] until some input appears; this keeps the server off the run queue and avoids burning excess CPU time.

[1] The `FD_` macros are documented in the `select` manual page.

[2] `select` may also be set up to return immediately or to time out if no input appears on any socket within a given interval; see the manual page for details.

⟨Select sockets which are ready for reading 5a⟩ ≡

```
User *user;
int maxSocket = 0;

FD_ZERO(&ready);
FD_SET(masterSocket, &ready);
maxSocket = masterSocket;
for (user = userList; user; user = user->next) {
  int socket = user->socket;

  FD_SET(socket, &ready);
  if (socket > maxSocket) maxSocket = socket;
}
numReady = select(maxSocket+1, &ready, NULL, NULL, NULL);
if (numReady < 0) {
  perror("server");
  exit(-1);
}◇
```

Macro referenced in scrap 4c.

2.2.1 New connections

Once the `fd_set` called `ready` has been set up properly, we can use it with the `FD_ISSET` macro to check whether the master socket has a connection pending. If so, we accept the new connection using `accept`; the second and third arguments to this call are a pointer to and the size of an Internet addressing structure (which we will ignore). If `accept` is successful, it returns a new socket, the server socket for a new user, which we assign to `userSocket`. We then allocate a new `User` structure, `user`, and we save `userSocket` into it and put the new structure into our list of connected users.

⟨Accept a new connection if `masterSocket` is ready 5b⟩ ≡

```
if (FD_ISSET(masterSocket, &ready)) {
  struct sockaddr_in addr;
  int addrLength = sizeof(addr);
  int userSocket = accept(masterSocket, (struct sockaddr *)&addr, &addrLength);
  User *user = (User *) malloc(sizeof(User));

  user->socket = userSocket;
  user->prev = NULL;
  user->next = userList;
  if (userList) userList->prev = user;
  userList = user;
}◇
```

Macro referenced in scrap 4c.

2.2.2 Input from existing users

Next we attempt to read text from users who are already connected. To do this, we loop through each user on our list. If the user's bit is set in `ready`, we can read from that user without fear of blocking.

⟨Take input from users who are `ready` 6a⟩ ≡

```
user = userList;
while (user) {
  User *nextUser = user->next;
  int socket = user->socket;
  if (FD_ISSET(socket, &ready)) {
    ⟨Take input from user on socket 6b⟩
  }
  user = nextUser;
}◇
```

Macro referenced in scrap 4c.

To take input from an existing user, we use the `read` system call on `user->socket` to transfer data from the socket into `buf`. The `read` call returns the number of bytes transferred; if the number of bytes is negative, then an error occurred. If we read zero bytes, the user closed connection, so we remove the user from our list. Otherwise, all is well, so we broadcast the data to all other users.

⟨Take input from `user` on `socket` 6b⟩ ≡

```
char buf[1025];
int bytes;

bytes = read(socket, buf, 1024);

if (bytes < 0) {
  perror("server");
} else if (bytes == 0) {
  if (user->next) user->next->prev = user->prev;
  if (user->prev) user->prev->next = user->next;
  else userList = user->next;
  close(user->socket);
  free(user);
} else {
  ⟨Broadcast user's input to all other users 6c⟩
}◇
```

Macro referenced in scrap 6a.

To broadcast a string, we first null-terminate the data, then loop through the users, using `write` to send the text to all users other than the current one.

⟨Broadcast `user`'s input to all other users 6c⟩ ≡

```
User *otherUser;

buf[bytes] = '\0';
for (otherUser = userList; otherUser; otherUser = otherUser->next) {
  if (otherUser != user) write(otherUser->socket, buf, bytes);
}◇
```

Macro referenced in scrap 6b.

3 The client

The client also looks like a typical C program.

`"client.c"` 6d ≡

```
⟨Client include files 7a⟩
⟨Client main routine 7b⟩
◇
```

The client needs to include the standard I/O headers, the socket headers, and `netdb.h`, which provides the `gethostbyname` routine.

⟨Client include files 7a⟩ ≡

```
#include <stdio.h>
⟨Socket headers 2c⟩
#include <netdb.h>◇
```

Macro referenced in scrap 6d.

In the client's `main` routine, we declare the socket `sock` and the event-loop terminator `done`. The first task is to initialize `sock`; then we then begin looping until something causes the client to quit. Finally we close the socket and exit.

⟨Client `main` routine 7b⟩ ≡

```
main(int argc, char **argv)
{
  int sock, done = 0;
  ⟨Initialize and connect sock 7c⟩
  while (!done) {
    ⟨Respond to input 8a⟩
  }
  close(sock);
  exit(0);
}◇
```

Macro referenced in scrap 6d.

3.1 Initializing the client

The client will attempt to connect to a server listening at port 9001 of the machine named on the command line. We first create `sock` by calling `socket`. Then we convert the machine name given on the command line into a network address by calling `gethostbyname`. This call returns a pointer to a `hostent` structure, whose `h_addr` field we can copy into the `sin_addr` field of our Internet address structure called `address`. When the `address` structure has been constructed, we call `connect` to connect our socket to the server's master socket.

⟨Initialize and connect `sock` 7c⟩ ≡

```
struct sockaddr_in address;
struct hostent *hp;

if (argc < 2) {
  fprintf(stderr, "Usage: %s <servermachine>\n", argv[0]);
  exit(-1);
}
sock = socket(AF_INET, SOCK_STREAM, 0);
hp = gethostbyname(argv[1]);
address.sin_family = AF_INET;
address.sin_port = htons(9001);
memcpy((char *) &address.sin_addr, hp->h_addr, sizeof(address.sin_addr));
if (connect(sock, (struct sockaddr *)&address, sizeof(struct sockaddr_in))) {
  perror("client");
  exit(-1);
}◇
```

Macro referenced in scrap 7b.

3.2 The client loop

Once the client's socket has been created and connected to the server, we enter the main loop. Inside this loop we use `select` to multiplex between two descriptors: the client socket `sock` and the standard input.

⟨Respond to input 8a⟩ ≡
```
int serverReady = 0, stdinReady = 0;
⟨Select descriptors from which the client may read 8b⟩
if (serverReady) {
  ⟨Read from server 8c⟩
}
if (stdinReady) {
  ⟨Read from standard input 9⟩
}◇
```
Macro referenced in scrap 7b.

To select the descriptors which are ready to be read, we construct an **fd_set** just as we did in the server. We set the bits we are interested in: **sock** for the socket, and 0 for the standard input. Then we use select to block until some data appears and to clear the "unready" bits in the **fd_set**. When select returns we set the Booleans **serverReady** and **stdinReady** to reflect the state of the two input streams.

⟨Select descriptors from which the client may read 8b⟩ ≡
```
fd_set ready;

FD_ZERO(&ready);
FD_SET(sock, &ready);
FD_SET(0, &ready);
if (select(sock+1, &ready, NULL, NULL, NULL) < 0) {
  perror("client");
}
if (FD_ISSET(sock, &ready)) serverReady = 1;
if (FD_ISSET(0, &ready)) stdinReady = 1;◇
```
Macro referenced in scrap 8a.

If the socket has data waiting, we read from the server through the socket, passing whatever text we read to the standard output. If we read zero bytes, then the server has closed the connection, so we quit.

⟨Read from server 8c⟩ ≡
```
char buf[1024];
int bytes = read(sock, buf, 1024);

if (bytes < 0) {
  perror("client");
} else if (bytes > 0) {
  buf[bytes] = '\0';
  fprintf(stdout, "%s", buf);
} else {
  fprintf(stderr, "Connection closed by server\n");
  done = 1;
}◇
```
Macro referenced in scrap 8a.

If the standard input is ready, we read from it and pass whatever we get to the server. If we read zero bytes, then the user has typed CTRL-D, so we quit.

⟨Read from standard input 9⟩ ≡

```
char buf[1024];
int bytes = read(0, buf, 1024);
if (bytes < 0) {
  perror("client");
} else if (bytes == 0) {
  done = 1;
} else if (bytes > 0) {
  buf[bytes] = '\0';
  if (write(sock, buf, bytes) < 0) {
    perror("client");
  }
}◇
```

Macro referenced in scrap 8a.