A Shared Socket Library Server and Client

by John Wiederhirn and John Orr

Wednesday is gyro day in CATS, so we have to go out to lunch and eat gyros. Because David the Engineer used to work in CATS, we are morally obligated to bring him along. Like the rest of the engineers, David is usually working very hard and doesn't notice what time it is, so we have to remind him that it's time to leave for lunch. Unfortunately, David can't hear us yelling to him in the Engineering department, so we needed an alternate way to tell David that it is time for lunch. The only logical conclusion was to write a program that uses the Shared Socket Library to make a requester pop up on David's Workbench screen telling him it is lunch time.

To write a network application for the Shared Socket Library you will need two things:

1) An understanding the material in the article ''Developing Network Applications for the Amiga'' from the January/February 1992 issue of Amiga Mail.

2) The Shared Socket Library include files and Autodocs (which are on the Network Developer's Disk and the Denver/Milano 1991 Devcon disks)

This application has to be broken into two pieces, a client and a server. The client program will send out the notes. On Amiga A, the user runs the client (*SendNote*), passing it a note string (like ''David, it's time to eat'') and a machine address for Amiga B. The client then sends a note request off to the note server (*ShowNote*) on Amiga B. When the note server gets the note request, it pops up an EasyRequest containing the note on Amiga B. The server waits for the user to click an ''OK'' gadget, then sends back an acknowledgement to the client.

There are a couple of decisions to make about the application before coding anything. First, there are two transport protocols to choose from: TCP and UDP. To make things easier, this application uses TCP because it is a reliable protocol, so the application doesn't have to worry about making sure data makes it across the network. The application uses a client/server model, but there are two of those models to choose from: iterative and concurrent. Because this application does not need to handle more than one request at the same time, the iterative server is a better choice. This also makes coding easier.

The Application Protocol

The note program now needs an application protocol for sending information between the client and server. When the client sends a request message to the server, it needs to send the note string. It also would be nice to supply the text for the buttons that will pop up in the requester. On the return trip from *ShowNote* to the *SendNote* client, the protocol only needs to define some return codes so the client can tell if the request worked and which button on the requester the remote user clicked. Because the application needs to send different types of packets (one type containing a message request, the other type containing the response from the server), the protocol needs to have a way to specify a packet type.

The following structure is the packet that *ShowNote* and *SendNote* send back and forth to each other:

```
struct NetNote
{
    int nn_Code;
    int nn_Retval;
    char    nn_Text[200],
            nn_Button[40];
};
```

On the trip from client to server, the nn_Code field is the message request packet type (`NN_MSG`), nn_Text is the note for the remote user, and nn_button is the text for the EasyRequest buttons. On the return trip, nn_Code is either `NN_ACK`, if there was no error, or `NN_ERR`, if there was an error. If there was no error, nn_Retval contains the number of the EasyRequest button that the user selected.

In this application, the same packet is passed back and forth for both legs of the trip (client to server and vice versa). If the application required sending large chunks of data in one direction and small chunks of data in the other direction, it would be a good idea to use packets of different sizes. Otherwise, if the application used only one packet size, the size of the packet would be huge compared to the size of the small chunk of data, which would unnecessarily loading the network.

The *ShowNote* Server Application

Although the Amiga's Shared Socket Library is meant to be compatible with the Unix socket implementation, there are a couple of Amiga-specific quirks that an Amiga networked application has to take care of. First, an Amiga application has to open the Shared Socket Library (*socket.library*). Before calling any other functions in *socket.library*, the program has to call the socket.library function, setup_sockets(). *ShowNote.c* code has almost all of the network intialization material in the SS_Init() function.

```
/*
** Attempt to open socket library and initialize socket environment.
** If this fails, bail out to the non-returning AppPanic() routine.
*/

if (SockBase = OpenLibrary("inet:libs/socket.library",0L))
{
    setup_sockets( 3, &errno );
}
else
{
    AppPanic("Can’t open socket.library!",0);
}
```

Unlike most libraries, *socket.library* does not go in LIBS:. Instead, it goes with the network-specific support files, in INET:, in a LIBS directory. This location needs to be hardcoded into the application.

Next, create a socket:

```
/*
** Open the initial socket on which incoming messages will queue for
** handling.  While the server is iterative, I do it this way so that
** SIGBREAKF_CTRL_C will continue to function.
*/

int snum;

if ((snum = socket( AF_INET, SOCK_STREAM, 0 )) == -1)
{
    AppPanic("Socket Creation:",errno);
}
```

and figure out what the well-known port number of the server is. For testing purposes, *SendNote* and *ShowNote* use a hard-coded port number (8769). If the application was installed on a network, each machine that ran the server would need an entry for the ‘‘note’’ service in the *inet:db/services* file. In that case, both the server and the client would have to use getservbyname() to find the port number of the ‘‘note’’ service.

The server then needs to build a sockaddr_in structure describing itself. The sockaddr_in contains all the information needed to map an initialized socket to a specific transport address on the Internet (thus the _in suffix, other network protocols have different suffixes). Before the socket can be assigned an Internet transport address, it needs to be initialized:

```
struct sockaddr_in sockaddr;

memset( &sockaddr, 0, len ); /* clear sockaddr */
sockaddr.sin_family = AF_INET;
sockaddr.sin_port = 8769;
sockaddr.sin_addr.s_addr = INADDR_ANY;
```

Next, bind the socket to the well-known address in sockaddr.

```
if ( bind( snum, (struct sockaddr *)&sockaddr, len ) < 0 )
{
    AppPanic("Socket Binding:",errno);
}


/*
** Okay, the socket is as ready as it gets.  Now all we need to do is to
** tell the system that the socket is open for business.
*/

listen( snum, 5 );
```

The call to bind() takes the socket identifier and sockaddr_in structure and assigns a unique network identity (a transport address) to the socket, making it visible to the network. The listen() call tells *socket.library* the socket is ready to receive incoming messages. The ’5’ in the listen() call tells the system the size of the connection request queue.

A connection request that a client sends to the server’s socket goes into this queue and waits for the server to process it. If requests arrive at the server’s socket faster than the server can process them, the queue fills to capacity. While the queue is full, the system rejects any other incoming connection requests. *ShowNote* uses 5 because that is the maximum allowed by Berkeley Sockets.

Listening for Network and Amiga Events

Once back in the main() routine, the application is almost ready to start processing network events. The only thing remaining is to decide what kind of events the server needs to hear about. Most applications will need to be aware of both network and local Amiga events, which means separate masks need to be set up for both types of events. On the network side, the mask needs to contain information on which sockets need responses.

```
/* First, prepare the various masks for signal processing */
fd_set sockmask;

FD_ZERO( &sockmask );
FD_SET( socket, &sockmask );
```

The sockmask variable will be used as a template to indicate what network events the application notices. Since sockmask came off the stack and contains garbage, the FD_ZERO() call clears all its network signal bits. The FD_SET() call sets the mask to listen for events relating to the socket that the server created earlier. Everything is prepared, so the next step is entering the event loop itself.

```
long umask;
fd_set mask;

while(1)
{
    /*
    ** Reset the mask values for another pass
    */

    mask = sockmask;
    umask = SIGBREAKF_CTRL_C;

    /*
    ** selectwait is a combo network and Amiga Wait() rolled into
    ** a single call.  It allows the app to respond to both Amiga
    ** signals (CTRL-C in this case) and to network events.
    **
    ** Here, if the selectwait event is the SIGBREAK signal, we
    ** bail and AppPanic() but otherwise its a network event.
    */

    if (selectwait( 2, &mask, NULL, NULL, NULL, &umask ) == -1 )
    {
        AppPanic("CTRL-C:\nProgram terminating!",0);
    }
```

Before an event occurs, the mask variable tells selectwait() which network events the server wants to receive. After an event occurs, the mask variable indicates which socket triggered the network event. The sockmask variable is used to reset mask back to its original mask value at the top of each pass through the event loop.

In addition to waiting on network events, selectwait() also waits on Exec signals for the current task. For this example, the only Amiga event the server cares about is a Ctrl-C break (`SIGBREAKF_CTRL_C`). The selectwait() function has a simple purpose, but due to the wide scope of network events, the function has a myriad of parameters and configurations. This example uses the bare minimum of what's possible using selectwait(), and many network-friendly applications will be able to get by using only a small subset of selectwait()'s potential. In this case, the network event set is passed in mask, and the Amiga event mask is passed in umask.

If selectwait() returns with a value of -1, it means the Amiga event mask was the trigger. Otherwise, a network event caused the function to return. This example is simple enough that a Ctrl-C interrupt can be handled rather easily.

Identifying Network Events and Talking to the Client

If selectwait() returned as a result of a network event, then a bit more detective work is needed to determine which socket caused the event. The example only has to watch a single socket, so if that socket wasn't the cause, an error occurred. In more complex servers, the server might have to check each of several sockets using the FD_ISSET() macro to determine which one caused the event. There is also the possibility that more than a single event came in at the same time, so an application with many active sockets needs to take into account the possibility of multiple true results from FD_ISSET().

```
if (FD_ISSET( socket, &mask ))
{
    HandleMsg( socket );
}
else
{
    AppPanic("Network Signal Error!",0);
}
```

The FD_ISSET() macro checks if there was activity on a socket by checking if the socket's bit is set in the socket mask passed to selectwait(). In this code, there is only a single active socket, so if that socket isn't the trigger then there was an error.

When a client tries to talk with a server, it first attempts to get a connection between itself and the server. In this example, the server application returns from its selectwait() with that socket identified in the mask variable. The server has to accept the connection:

```
sockadd_in saddr;
int len;

if (!(nsock = accept( sock, (struct sockaddr *)&saddr, &len )))
{
    AppPanic("Accept:",errno);
}
```

When the server calls accept(), the function attempts to form a connection with the client. If it can do so, it returns a new socket identifier. The new socket identifier corresponds to a new socket where the all future communication between the client and server will occur. This allows concurrent servers to use an existing socket to establish new connections while carrying on other private client-server conversations.

The accept() function also passes back a sockaddr_in structure describing the client. In this example, the server uses this address to figure out the client's host name:

```
struct hostname *hent;
struct in_addr sad;
char *dd_addr, *hname, rname[80];
```

```
/*
** Get the internet address out of the sockaddr_in structure and then
** create a dotted-decimal format string from it.
*/

sad = saddr.sin_addr;
dd_addr = inet_ntoa(sad.s_addr);

/*
** Use the internet address to find out the machine’s name
*/

if ( !( hent = gethostbyaddr( (char *) &sad.s_addr,
                              sizeof(struct in_addr),
                              AF_INET )))
{
    AppPanic("Client resolution:\nAddress not in hosts db!", 0 );
}
hname = hent->h_name;
```

Right now, if the server cannot identify the client’s machine, it terminates with an error. While this may seem a bit drastic, it does prevent anonymous messages from being sent across the system. Coding a secure client and server requires more complex protocols and involves many other issues which are beyond the scope of this article.

After doing a little formatting of the address and name information, the HandleMsg() function begins the actual process of communicating with the client.

```
int nsock;
struct NetNote in;

/*
** Okay, now the waiting packet needs to be removed from the connected
** socket that accept() gave back to us.  Verify its of type NN_MSG and
** if not, set return type to NN_ERR.  If it is, then display it and
** return an NN_ACK message.
*/

recv( nsock, (char *)&in, sizeof(struct NetNote), 0 );
if (in.nn_Code == NN_MSG)
{
    DisplayBeep(NULL);  /* DisplayBeep() to get the user’s attention */
    DisplayBeep(NULL);
    retv = DoER( rname, (char *)&in.nn_Text, (char *)&in.nn_Button );
    in.nn_Code = NN_ACK;
    in.nn_Retval = retv;
}
else
{
    in.nn_Code = NN_ERR;
}
```

The recv() function removes a struct NetNote sized amount of data from the socket nsock and places that information in a buffer. Once the message packet is in the buffer, the server verifies it is a proper packet by checking the nn_Code. The checking isn’t really necessary since the example uses a reliable protocol (TCP) and there is only one type of packet the server can receive. After checking the packet, the server prepared the nn_Code field for the return trip to the client. If there was something wrong with the packet, the server sets nn_Code to `NN_ERR`, otherwise the server sets nn_Code to `NN_ACK`.

If there was no error, the server extracts the message and button text from the NetNote packet. The server passes them plus name and address information for the client to the DoER() routine. The DoER() routine creates a system EasyRequest, displays it, and returns a value which corresponds to the button which was pressed. The return information on which button was pressed is encoded into the nn_Retval field of the NetNote packet, which is then ready to be sent back to the client.

The prepared packet is sent back to the client using the send() function:

```
/*
** Having dealt with the message one way or the other, send the message
** back at the remote, then disconnect from the remote and return.
*/

send( nsock, (char *)&in, sizeof(struct NetNote), 0 );
s_close( nsock );
```

The HandleMsg() function then closes nsock using s_close(). The packet protocol between the client and server in this application is defined so there are no cases where the client would send another message, so it can close the socket and break the connection. The acknowledgement/error packet will arrive at the client regardless of whether the connection is still active or not, at least under TCP/IP. HandleMsg() returns to the main() routine and event loop.

Once back in the main event loop, the server will continue connecting and responding to client messages until it receives a Ctrl-C interrupt or an error condition occurs.

Starting the *ShowNote* Server

The ShowNote server should run as a background CLI process. To start it, type the following line at a CLI prompt:

```
run >nil: shownote
```

You can start up the server when you boot your Amiga by adding that line to *s:user-startup*.

The *SendNote* Client Application

The client application, *SendNote*, sends a message for a server to pop up on its display. While the server is an Intuition program in that it uses a requester to display the messages and get responses, the client (*SendNote*) is strictly a CLI-based application. *SendNote* parses its command line using the AmigaDOS 2.0 ReadArgs() call. For more information on ReadArgs(), see the *AmigaDOS Manual, 3rd Edition* from Bantam, the article ‘‘Standard Command Line Parsing’’ from the May/June 1991 issue of Amiga Mail, or the DOS library includes and Autodocs.

There is very little difference in the application-wide setup of sockets between the client and server. Both must open *socket.library* and call setup_sockets() to initialize the socket environment. One minor difference is the number of sockets the client initializes. The client only needs a single socket to establish a connection to the server, where the server requires at least two (one for receiving connection requests and one for talking to a client).

The FinalExit() routine is both the client's error handler and its shutdown routine. Since the client doesn't need to maintain so much operating information, the shutdown routine is rather simple, and can be used for both errors and normal termination.

Resolving the Target (Host) Address

The client gets a string back from the ReadArgs() call which contains the hostname in either dotted-decimal notation or ASCII form. The client needs to convert that string to a usable form.

```
struct sockaddr_in serv;
struct hostent *host;
char *hostnam, *text, *button;

/*
** First we need to try and resolve the host machine as an IP/Internet address.
** If that fails, fall back to searching the hosts file for it.  Later versions of
** gethostbyname() may use DNS to find a host name, rather than searching the hosts file.
*/



bzero( &serv, sizeof(struct sockaddr_in) );
if ( (serv.sin_addr.s_addr = inet_addr(hostnam)) == INADDR_NONE )
{
    /*
    ** Okay, the program wasn't handed a dotted decimal address,
    ** so we check and see if it was handed a machine name.
    */

    if ( (host = gethostbyname(hostnam)) == NULL )
    {
        printf("Host not found: %s\n",host);
        FinalExit( RETURN_ERROR );
    }

    /*
    ** It does indeed have a name, so copy the addr field from the
    ** hostent structure into the sockaddr structure.
    */

    bcopy( host->h_addr, (char *)&serv.sin_addr, host->h_length );
}
```

After clearing out the serv sockadd_in structure, the client tries to convert the host name string (hostnam) it got from its command line. It ues the inet_addr() function to convert a dotted-decimal address to an IP address block. If this fails, the server treats the string hostnam as an ASCII string containing a host name, and tries to get a normal IP address using gethostbyname().

This will search the *hosts* file (*inet:db/hosts*) for a matching entry. Future versions of gethostbyname() may use DNS (domain name system), which allows gethostbyname() to ask a server for host information rather than looking it up in a hosts file.

If it is successful, gethostbyname() returns a pointer to a hostent structure. It requires a little work to to convert this hostent structure to a sockaddr_in (IP socket address) structure. There is a sockaddr structure embedded inside the hostent structure which can be used as a sockaddr string in this case. The call to bcopy() copies that embedded sockaddr structure into the client's sockaddr_in buffer.

Locating the Server Port and Connecting to It

The next step is to find the server's port number. In the case of this example, the port name is hardcoded into the server and client. A real networked application should have an entry for the ‘‘note’’ service in the *inet:db/services* file. The code below finds the port number in the client machine's *inet:db/services* file.

```
struct servent *servptr;
char servnam[] = "note";

if ((servptr = getservbyname( servnam, "tcp" )) == NULL)
{
    printf("%s not in inet:db/services list!",servnam);
    FinalExit( RETURN_ERROR );
}
serv.sin_port = servptr->s_port;


/*
** This tells the system the socket in question is an Internet socket
*/

serv.sin_family = AF_INET;
```

Since the client and server are running on top of an IP (Internet Protocol) system, the client needs to specify `AF_INET` in its call to socket() just as the server did.

```
/*
** Initialize the socket
*/

if ( (sock = socket( AF_INET, SOCK_STREAM, 0 )) < 0 )
{
    printf("socket gen: %s\n", strerror(errno));
    FinalExit( RETURN_ERROR );
}
```

The client socket is initialized and ready. Because the client knows the IP address of the server's machine and the service number of the ‘‘note’’ service, the client knows the well-known transport address so it can attempt to establish a connection with the server:

```
/*
** Connect the socket to the remote socket, which belongs to the
** server, and which will "wake up" the server.
*/

if ( connect( sock,
              (struct sockaddr *) &serv,
              sizeof(struct sockaddr) ) < 0 )
{
    printf("connect: %s\n", strerror(errno));
    s_close( sock );
    FinalExit( RETURN_ERROR );
}
```

The connect() call contacts the server across the network and attempts to form a connection. There are many things which can go wrong at this point, and any return less than zero indicates an error condition. If an error does occur, the *socket.library* function strerror() converts the error number in errno into a readable error message, which is then displayed by the client. In case of an error, the client already has a socket open, so it then must close the socket and terminate using the application's error handler, FinalExit().

Once the connection is established, the client needs to prepare the note request packet. Since the call to ReadArgs() has already parsed everything, the client only has to copy the strings into a NetNote structure:

```
struct NetNote out;

out.nn_Code = NN_MSG;
strcpy( (char *)&out.nn_Text, text );
strcpy( (char *)&out.nn_Button, button );
```

The client has filled in all the relevant fields of the NetNote structure, so it is ready to send. The server will fill in the nn_Retval field before passing it back to the client, so the client doesn't need to fill it in for the client-to-server leg of the trip. All that remains is to transfer the packet across the network:

```
send( sock, (char *)&out, sizeof(struct NetNote), 0 );

printf("\nMessage sent to %s...waiting for answer...\n", hostnam );

/*
** Wait for either acknowledge or error.
*/

recv( sock, (char *)&out, sizeof(struct NetNote), 0 );
```

Now the client has to wait for the server to respond. This is one of the few points where the client can hang forever. If the server receives the message, and never replies back, the client will never stop waiting for a reply. A real application would time-out if the server didn't respond within a certain time interval. One way to do this is using a selectwait() which breaks when triggered by a *timer.device* event. We'll leave that as an exercise.

The call to recv() waits until the server sends back a reply. Once the client receives the reply, the client has to check the NetNote structure's nn_Code field. If it's `NN_ERR`, an error occurred, and the client terminates through the FinalExit() handler. If an `NN_ACK` packet comes back from the server, the nn_Retval field contains the number of the button the user pressed on the *ShowNote* requester.

The server gets the button number directly from the EasyRequest() function that the server uses to pop up the requester. The buttons on the requester are numbered from 1, increasing left to right, except for the rightmost button, which is button zero. If there is only one button, that button will return a zero (it's the furthest right).

After the client gets the packet back from the server, the client's task is complete. It only has to clean up after itself.

Starting the *SendNote* Client

The SendNote command has the following template, and uses the AmigaDOS 2.0 ReadArgs() call to parse the CLI command line:

```
SENDNOTE HOST/A,TEXT,BUTTON
```

The HOST argument is the address of the server machine. It is either a dotted-decimal notation address or its ASCII host name from the *inet:db/hosts* file. The TEXT argument is the string the server will pop up in its requester. The BUTTON argument is the ASCII string that will appear in the requester button. For example:

```
sendnote 123.123.123.1 "I've fallen and I can't get up!" "Help me!"
```

displays a requester on the machine whose address is 123.123.123.1, with the text of the requester saying "I've fallen and I can't get up!" and a button labeled "Help me!". If that machine is not running the server, then *SendNote* just displays an error message and terminates. If the *inet:db/hosts* file on the machine running *SendNote* has an entry that gives 123.123.123.1 the name "foo", then

```
sendnote foo "I've fallen and I can't get up!" "Help me!"
```

will have the same effect as the previous form of the command. There are also default values for both the TEXT and BUTTON arguments. If the user doesn't provide an argument for BUTTON, then *ShowNote* defaults to "OK". If both TEXT and BUTTON are missing, the TEXT argument defaults to "==PING!==". To give the user on the server machine several buttons to choose from, put several strings in the BUTTON argument delimited by '| ' characters, like this:

```
sendnote foo "Is it time yet?" "yes|no|maybe"
```

That's all there is to the client application. Because the *socket.library* routines take care of so much of the network ‘‘nitty gritty’’, the application code can deal with networks in a straightforward and simple manner. Two applications (client and server), each around 8K bytes in size, are able to implement a complete and working intranetwork communication system (albeit a very simplistic one). If you are interested in doing more serious development using the AS225 software (and thus IP and TCP/UDP), you should take a look at more advanced texts. A good start is *Unix Network Programming* by W. R. Stevens (Prentice-Hall, ISBN 0-13-949876-1). It covers many aspects of network protocol and application design, as well as explaining quite a bit about ‘‘Berkeley Sockets’’ which *socket.library* implements.