Developing Network Applications for the Amiga

by Dale Larson

When you run a wire between two or more computers, you have a network. Big Deal. When your applications use that wire however, you have a revolution. Although some of the following software is only internal or experimental, these are things I can do now with my Amiga, with software that I have now:

- From my Amiga, I can transparently access filesystems on Suns, on the local Vax system (cbmvax), and on other Amigas.

- Whenever I print, I send my files to a network printer.

- I continuously receive mail on my Amiga--from as far as Seattle, Sydney and Denmark, and near as a desk next to mine.

- Every night when I go home, my Amigas at work (and several others) are used to do distributed graphics rendering. The process is started over the network and all data is sent over the network. A picture that would have taken a week can be finished overnight.

In the scheme of what is possible, this is only the tip of the iceberg.

- In a high school environment, a network could allow students to interactively participate in computer simulations. It could allow them to collaborate electronically. It could allow teachers to electronically monitor and assist students. It could save schools money because peripherals such as printers, hard-drives and CD-ROMs could be easily shared. Even the computational power of one expensive machine could be shared by the students.

- A small office can use a network for an email-like facility for phone messages and other notes. Another application might replace the intercom. Form letters can be kept in a central database accessed by a word processor. A distributed appointment calendar could allow a secretary to add a new appointment even as the boss is looking at what his afternoon schedule is. A distributed database application would allow access to such things as a central client database, outstanding orders and the present inventory.

- Imagine multi-player games that use the computational power of each machine connected by a high speed Local Area Network (LAN).

- In a software development environment, several programmers can work on the same project, updating the same sources. Debugging information could be sent over the network, or a debugger on one machine could control the programs on others (For example, there is a version of *Wack* that runs over a network).

- Multimedia applications might do any number of exciting things with the network. A few of the applications which have been experimented with on other machines are: real-time audio and video conferencing, interactive demos for groups, and shared electronic blackboards.

In much the same way as all applications are candidates for a GUI interface, all applications are candidates for becoming network applications. The GUI has only changed the ways in which people interact with their computers. Networks will change the ways in which people interact with each other.

This article introduces some of the principles of writing network programs using the AS225's Berkeley Socket interface. Even more so than in most of software development, networking seems simple in theory, but, in reality, gets complicated in a hurry. To develop network software for AS225, you will need to obtain the Network Developer's kit from CATS. It has all the necessary include files and Autodocs to develop for the AS225's *socket.library*. Also, you should plan read at least some of the material in the ''References'' section of this article.

Protocol Layers and the Berkeley Sockets Interface

Network applications should always be written to the Application Programmer's Interface (API) of a particular protocol, not to the network hardware. Network standards usually include several protocols layered one on top of another. These groups are often referred to as protocol stacks. At the lowest level, one of the protocols must interface to some network hardware. Each layer adds some abstraction to using the network on a lower level. This serves to make it easier to program network software as the developer doesn't have to deal with networking details that are well below the level of the software under development.

The International Standards Organization (ISO) has created a reference model on which to base new network layerings. The ISO 7-Layer Reference Model of Open Systems Interconnection looks like this:

Layer	Functionality
7	Application
6	Presentation
5	Session
4	Transport
3	Network
2	Data Link (Hardware Interface)
1	Physical Hardware Connection

ISO 7-Layer Reference Model

The only protocol stack for the Amiga which is currently available from Commodore is TCP/IP. Our AS225 software package includes the standard TCP/IP protocols and several standard Internet applications. It has the same API as most Unix machines running TCP/IP using the Berkeley Sockets interface.

Unix was designed to have a common method of accessing both files and devices. Before a Unix application can perform any I/O operations on a file, it has to open() it. The Unix open call returns an integer called a file descriptor that corresponds to the open file. The application uses this file descriptor to manipulate the file.

This method of I/O is not quite general enough for networking. Instead, there is the Berkeley Sockets interface. A socket is an entity used to send and receive data across a network. A socket can be ‘‘plugged in’’ or bound directly to the socket of some other application on another machine somewhere on the network. Like the Unix file system, applications access their sockets using a file descriptor, although it is typically referred to as a socket descriptor.

Thousands of network applications have been written to the socket interface. AS225 has been shipping since December, 1990, and everything needed to write network applications for AS225 is included on the 1991 DevCon disks and on the Network Developer’s Disk.

On the Amiga, layers 1 and 2 should always be the network hardware and SANA-II Network Device Driver (SANA-II defines the lowest level software interface to networking hardware). In AS225, layer 3 is the IP and ICMP protocols of the TCP/IP protocol stack. These protocols aren’t used directly by application developers. Essentially, they handle machine to machine communication. The transport layer uses the network layer (and the layers below it) to provide communication between individual processes on different machines. Most current network applications use transport protocols. The transport protocols in TCP/IP are TCP and UDP.

The TCP transport protocol is a connection-oriented, stream protocol. Basically, the socket of one application connects to the socket of another application and they communicate across the network in data streams. The two applications can be running on arbitrary machines on a network. The big plus of using

TCP is that it is a *reliable* protocol. If the data is put in one end of a TCP stream, it either gets to the other end intact, or not at all (which causes an error at the sender's end). This makes it easier to program applications because the application programmer does not have to worry about packet corruption.

UDP is a connectionless, datagram protocol. An application using UDP sends datagrams to some other application on the network. A datagram is a fixed-length message. Because the sockets of UDP applications are not connected, a datagram sent from a socket can be sent to an arbitrary UDP socket. Unlike TCP, UDP is not a reliable protocol, so an application that uses UDP has to account for errors that can occur during transmission.

The details of all the various protocols and how they behave are quite complex and are beyond the scope of this article. This article deals with the difference between connection-oriented stream protocols (sockets which are obtained as type `SOCK_STREAM`) and connectionless datagram protocols (type `SOCK_DGRAM`).

The Amiga Shared Socket Library (*socket.library*) is Commodore's implementation of the standard functions for manipulating, sending data to, and receiving data from sockets. Other network APIs for TCP/IP have been written for other platforms (most notably TLI on Unix SVR4 systems). Programs written using sockets on one machine can communicate just fine with programs written using another API (i.e., TLI) on another machine. Sockets are not specific to TCP/IP. They can be used with different ‘‘domains’’. TCP/IP is one domain, another network protocol is another domain and local Inter-Process Communication (IPC) is yet another. Our socket library currently supports only TCP/IP.

Layers 5 and 6 of the OSI model, the presentation and session layers, do not exist in TCP/IP, or most other protocol stacks. So with TCP/IP on the Amiga, the protocol stack looks like this:

Application
Transport (TCP, UDP)
Network (IP, ICMP)
Data Link (SANA-II Network Device Driver)
Hardware (ethernet, arcnet, etc.)

TCP/IP Protocol Stack

In spite of the fact that protocols come in a stack, your application will only come into direct contact with a protocol at the top of the stack. In the case of TCP/IP, this is the transport layer. In theory, you are not required to know protocols below the one used for your application. In practice, higher-level protocols are often described in terms of additions to lower-level protocols.

Network Applications

Most network applications are built around a client/server model. In the client/server model, a server application runs on one machine somewhere on a network. That server waits for a request for a particular service it provides. These requests come from client applications that are running on other machines on the network. A service can be as simple as echoing back text sent to the server or as complex as providing a remote login facility. For example, the ftp (*File Transfer Program*) application copies files between networked machines. Ftp actually consists of two programs, a client and a server. The server waits around for a client to request some service, like listing a directory or transferring a file.

On most networks, each machine is capable of running both client and server programs simultaneously, but on some networks a *machine* is either a client or a server and may only run programs of that type. The focus of this article is peer-to-peer networks (the former), not client-server networks (the latter).

Application Protocols

Every networked program must agree on how to send data across the network and on what meaning to attach to that data. Therefore, the application itself has a protocol. This is the application layer of the ISO reference model and TCP/IP protocol stack. The application-specific protocol can include such things as what transport protocol will be used, what initialization takes place, how any security is implemented, what format data will be in, etc.

For standard Internet applications (*ftp* for example), the application protocols are specified in detail in one or more reference documents called Requests for Comments, or RFCs. RFCs are the specifications for Internet protocols and standard applications. Even if you aren't implementing a standard Internet application, the RFCs offer insight into the complexities of application protocols and how they should be specified. See the ‘‘*References*’’ section of this article for more information on RFCs and standard Internet applications.

Normally application protocols are general enough that network applications can be ported to any machine which supports the network protocol. Neither the client nor the server knows (or cares) what type of machine is at the other end. Sometimes only one half of the application (client or server) is available for a given machine. For example, the currently released version of AS225 includes an NFS (Network File System) client program, but no NFS server program. To use NFS with AS225 requires access to any machine with an NFS server--it does not require any particular type of machine.

Kinds of Servers

There are two kinds of servers. Those which process one request at a time are called ‘‘iterative servers’’. Those which simultaneously service multiple requests (often by spawning a process to handle each request) are called ‘‘concurrent servers’’. Iterative servers are generally easier to write, but are only suitable for services which can be handled quickly and/or will not be accessed by multiple clients. Applications which use connectionless protocols (UDP) frequently have iterative servers, while applications with connection-oriented protocols (TCP) usually have concurrent servers.

Addresses

All data on the network is sent to and from network addresses. There are many different types of network addresses, at least one type for each layer of the protocol stack. For example, the Network layer of the TCP/IP protocol stack uses the IP and ICMP protocols which use 32-bit internet addresses (which are usually represented in ‘‘dotted decimal notation’’ e.g., 192.9.210.4) to talk to a specific machine at a specific internet-style address. When applications communicate with each other, they usually use a transport layer protocol, therefore the data is sent from one transport address to another. A transport address generally corresponds to a specific program running on a specific machine somewhere on a network.

A transport address consists of three parts: a protocol, a host address, and a process association. In AS225, the protocol in a transport address is either TCP or UDP. The host address is dependent upon the protocol, but in AS225, the host address is always the internet address of the host. The process association is also protocol dependent, and in AS225, the process association is a port number. TCP/IP port numbers are 16-bit integers that are used by the transport protocols on each host to direct network traffic to a specific process running on the machine at that internet address.

The set of port numbers is unique to each protocol. For example, port number 42 for UDP might belong to a different process than that which belongs to port number 42 for TCP. Without port numbers, multiple network programs could not run simultaneously.

Transport protocols are analogous in some ways to Amiga Exec devices. In such an analogy, there is a TCP device and a UDP device. Each device has about 65,000 units and none of the units can be opened in shared mode.

Once a socket is created, it has to be bound to a transport address. An application binds an address to a socket in one of two ways. The binding can be made explicitly by the program to a specific transport address (using the bind() call). Servers normally use this type of binding. In this case, the server uses a preset, ‘‘well-known’’ port number in its transport address. The port number is well-known because all of the server’s possible clients know what that particular server’s port number is. Because these clients know the server’s port number, the clients can construct a transport address for that particular service on any machine that runs that server. For example, the default, well-known port number for *ftp* is 21. If a client wants to use *ftp* to transfer files from a machine at the internet address 192.9.210.4, it can use *ftp’s* port number, the machine’s internet address, and the protocol (*ftp* uses TCP) to find the *ftp* server at 192.9.210.4. As long as there is an *ftp* server running at 192.9.210.4, the client should have no problem finding the server. This type of socket is analogous to a public message port.

The other way to bind a socket to a transport address is to let the socket library arbitrarily choose a port number for the application. Normally client applications bind this way because a client does not need a well-known address. The client supplies the server with its transport address when it sets up communication with the server. This type of socket is analogous to a private Exec message port.

Finding Servers

A client ‘‘finds’’ its server by the server’s transport address. As was mentioned eariler, the transport address consists of a protocol, a host address, and a port number. For the TCP/IP protocol stack, the protocol is either TCP or UDP.

The next thing the client needs to build after the transport address is an internet address for the server’s machine. Normally the client obtains that address from the user. The address can be in one of two forms, an internet address (in dotted decimal notation), or a host name which is an ASCII string that corresponds to the host’s internet address. If the client gets a host name, it asks the *socket.library* what internet address corresponds to the host name.

When a server starts, it opens a socket and bind()s that socket using the server’s well-known port number. There are two ways for the server’s well-known port to become well-known:

1) A server’s well-known port number can be hard-coded into both the client and server. This is recommended for prototyping new programs, but is a Very Bad Thing for programs which will be distributed. The port number is arbitrary, but must not be one of the reserved ports (see the next section) and must not conflict with a port number already in use.

2) Port numbers can be configurable. All distributed network applications should use configurable port numbers. In programs written for AS225, you should use the *inet:db/services* file to configure a port number. The function getservbyname() accepts a protocol (UDP or TCP) and the name of a well-known server and returns the port number of that service. This requires you to configure your application by adding an entry to the *inet:db/services* file on every machine which will use the application. Many standard Internet applications and Unix remote services are already in the *inet:db/services* file that comes with AS225. If your application isn’t already included, your installation scripts should add the entry for your application to *inet:db/services*. Offer a default value, but let your user actually pick the number since your port number must not conflict with another (pre-existing) port number.

Reserved Ports

Port numbers 1-255 are reserved for standard Internet applications (like ftp) and port numbers 256-1023 are reserved for standard Unix remote services (which are often available for machines other than Unix). You should never choose any of these port numbers for your application unless it is an implementation of a standard for which the port number is reserved. For more information on port numbering and reserved ports, see the ‘‘References’’ section at the end of this article.

Skeleton for Applications Using TCP (connection-based)

Aside from the special quirks of the *socket.library* (which is discussed in the ‘‘Shared Socket Library’’ section below), the basic outline of the core of most client/server model applications written with TCP starts with the server:

Create a socket:

```
int socket(int family, int type, int protocol)
```

Where *family* specifies the protocol family (which for the TCP/IP protocol stack is `AF_INET` from *<sys/socket.h>*), *type* specifies the abstract type of communication (either `SOCK_STREAM` for TCP or `SOCK_DGRAM` for UDP), and *protocol* is not generally used in the TCP/IP protocol stack and should be set to zero. If socket() fails, it returns a -1, otherwise it returns a socket descriptor.

Next, get the well-known port number of the server’s service:

```
struct servent *getservbyname(char *family, char *service)
```

where *family* is one of two strings, "tcp" or "udp". The *service* argument is the name of the service that this server provides. The function returns a `NULL` if there was an error. Otherwise, it returns pointer to a servent structure (from *<netdb.h>*):

```
struct    servent {
    char   *s_name;      /* official service name */
    char   **s_aliases;  /* alias list */
    int    s_port;       /* port # */
    char   *s_proto;     /* protocol to use */
 };
```

Next, build a sockaddr_in (from *<netinet/in.h>*) structure using the port number from the s_port field of the servent structure returned by getservbyname():

```
struct in_addr {
 u_long s_addr;
 };


struct sockaddr_in {
    short     sin_family;        /* address family.  Make this AF_INET for TCP/IP */
    u_short   sin_port;          /* the port number (the value from servent.s_port) */
    struct    in_addr sin_addr;  /* internet address.  For TCP/IP, make sin_addr.s_addr
                                    INADDR_ANY.  Bind will know what to do with this. */
    char      sin_zero[8];       /* unused by TCP/IP */
 };
```

The sockaddr_in structure is a TCP/IP specific version of the sockaddr structure.

Now give the sockaddr_in to bind() in order to attach a specific address to the socket:

```
int bind(int servsocket, struct sockaddr *name, int namelength)
```

where, *servsocket* is the socket descriptor of the socket that needs a specific address, *name* is a pointer to the sockaddr structure (or, for TCP/IP purposes, a sockaddr_in structure) that describes the address, and *namelength* is the length of the sockaddr structure.

Once the socket is bound to an address, the server calls listen() to indicate that it is waiting to receive connections.

The server is ready to start its main processing loop. The accept() function waits for and accepts a new connection at a socket:

```
int accept(int servsocket, struct sockaddr *name, int namelength)
```

where, *servsocket* is the socket (descriptor) on which the server is waiting for connections, *name* points to a buffer where accept() copies a sockaddr structure describing the client that is trying to connect to the server, and *namelength* is the length of the name buffer.

If accept() fails, it returns a negative value, otherwise accept() returns a socket descriptor of a new socket that is connected to the client's socket.

The server can communicate with the client using the new socket while it continues to listen to its original socket for new connections. The server may serve one client at a time (an iterative server), or give the new socket (the one returned by accept()) to a new process so it can handle talking to the client (a concurrent server).

Setting up the client is similar to setting up the server. You have to create a socket, find the service's well-known port number, and initialize a sockaddr_in structure that refers to the server. Creating the socket and finding the service's port number are identical to the method described above.

To initialize the sockaddr_in structure, fill in the sin_family (`AF_INET`) and sin_port (service's well-known port number) fields as in the server. To fill in the sin_addr field, the client needs to find the internet address of the server's machine. The client has to find this from either from an ASCII string of the IP address or the host name (either of which the client will probably get from the user). The inet_addr() function accepts an ASCII string of a numeric IP address and returns the internet address to put in the sockaddr_in's sin_addr.s_addr field. If the client only has the host name, it has to call gethostbyname() to find out the server's machine address:

```
struct hostent *gethostbyname(char *hostname)
```

The parameter hostname is an ASCII string naming the host. This function figures out the address of the host (usually by looking it up in the *inet:db/hosts* file). This function returns a pointer to a hostent structure:

```
struct     hostent {
    char   *h_name;       /* official name of host */
    char   **h_aliases;   /* alias list */
    int    h_addrtype;    /* host address type */
    int    h_length;      /* length of address */
    char   **h_addr_list; /* list of addresses from name server */
#define  h_addr  h_addr_list[0]  /* address, for backward compatibility */
 };
```

The #defined field, h_addr, contains a pointer to an in_addr structure, which contains the actual host address. Copy this value into the sockaddr_in.sin_addr.s_addr field.

Now the client needs to establish a connection between itself and the server. It does this with the connect() function():

```
int connect(int clientsocket, struct sockaddr *servname, int namelength)
```

where, clientsocket is the socket the client created earlier, servname points to the sockaddr_in structure the client just built, and namelength is the length of that sockaddr_in structure.

The server and client communicate using the send() and recv() functions:

```
int send(int socket, char *buf, int buflength, int flags)
int recv(int socket, char *buf, int buflength, int flags)
```

where, socket is the socket to send to/receive from, buf is a buffer that contains the data to transmit/is a place for recv() to put the data it receives, buflength is the length of buf, and
flags is beyond the scope of this article. For the moment, you can set it to zero.

The data is a continuous stream, with any meaning assigned by the application protocol (not to be confused with the network protocol). The data is always received either intact or not at all. The data almost always gets there unless there is a serious network or host machine problem. When the communications are finished, both sides s_close() the sockets which were connected.

Skeleton for Applications Using UDP (connectionless)

The basic outline of most client/server model applications written with UDP look something like this:

Server gets a socket with socket(), gets a port number with getservbyname(), builds a sockaddr_in structure describing the server, and gives that structure to bind() in order to attach a specific address to the socket. It then loops, waiting to recvfrom() any incoming datagrams and responding to any requests in those datagrams.

```
int recvfrom(int socket, char *buf, int buflength, int flags,
             struct sockaddr *clientname, int namelength)
```

The recvfrom() function is similar to the recv() function except it has two extra parameters. *clientname* is a buffer for a sockaddr_in structure. When recvfrom() receives a datagram from some application on the network, it fills in clientname with the transport address of that application. The size of the structure is *namelength*.

The client gets a server hostname from the user, gets a socket(), gets the server's port number with getservbyname(), builds a sockaddr_in structure describing the server, and sendto()s a datagram to the server's well-known address:

```
int sendto(int socket, char *buf, int buflength, int flags,
           struct sockaddr *servername, int namelength)
```

The client either waits for the server to send back a datagram to the client's socket, or give up because the server took too much time to reply. The client does this by calling select(). This function puts the client to sleep until either a particular set of sockets is ready for reading, writing, or exception processing, or after a timeout period has passed without any activity.

```
int select(int numsocks, fd_set *readsocks, fd_set *writesocks,
           fd_set *exceptsocks, struct timeval *timeout)
```

The numsocks parameter is 1 plus the number of sockets select() is waiting on. The readsocks, writesocks, and exceptsocks parameters are each a bitmask which tells select() which socket (or sockets) to wait for activity on. The fd_set structure (defined in *<sys/types.h>*), is basically a handle to one of these bitmasks. Programs cannot directly manipulate the bits in these masks. Instead, there are functions to do this:

```
FD_ZERO(struct fd_set *mymask)                /* clear all bits in mymask */
FD_SET(int mysocket, struct fd_set *mymask)   /* turn on the bit for mysocket
                                                 in mymask */
FD_CLR(int mysocket, struct fd_set *mymask)   /* turn off the bit for mysocket
                                                 in mymask */
FD_ISSET(int mysocket, struct fd_set *mymask) /* test if mysocket's bit
                                                 in mymask is set */
```

For the purposes of this article, the only relevant mask is readsocks, because the client is only waiting to read from a socket. Since the client isn't interested in the other masks, it makes writesocks and exceptsocks `NULL`.

The last parameter, timeout, sets the maximum amount of time that select() should wait for activity on the sockets it is watching.

When select() returns, its return value is either -1 if it failed, 0 if there was a timeout, or a positive number, which is the number of sockets that became ready. When select returns, it sets the bits in the bitmasks according to which socket (or sockets) became ready.

When select() returns, if a socket became ready, the client calls recvfrom() to get the datagram the server sent back. On timeout, the client might try to re-send the datagram since it may have been lost or corrupted. Datagrams can be also be received in an order different from that in which they were sent and can be received in duplicate.

Which Protocol Is Right For My Application?

Generally, if your application requires moving bulk data to far away places, you should be using TCP. For many other applications, TCP is also appropriate just because its reliability makes it easy to use. TCP is so easy to use because it provides so much functionality. The price paid for ease of use is performance.

TCP does a good job of moving bulk data from one side of the planet to another. For data which will only be sent across one physical network (one LAN), or for data sent in small pieces, TCP doesn't perform so well. A much more specific set of functionality can always provide better performance than the most general set can.

For performance-critical applications which don't move bulk data, UDP is usually the protocol of choice. Unfortunately, since UDP doesn't provide reliability, the application protocol must. This means a much more complicated application protocol. It isn't nearly as bad as it sounds, though, and Stevens (1990) offers two examples.

The Shared Socket Library

The primary goal of the Shared Socket Library is to provide a network API which is as compatible with standard Unix as possible. This makes porting many applications much easier, but it also creates many little quirks that cannot be ‘‘fixed’’. The justification behind this is: faithfully emulating Unix's quirks is better than creating new ones, since at least you can then write more portable software and only need to remember one set of quirks. Remember this when you wish that some function returned *void rather than *foo, etc. Expect to get a few spurious compiler warnings from your nice ANSI 'C' compiler.

Many functions in *socket.library* are only needed by those developers porting standard Unix remote services and probably should not be used by most Amiga applications. For example, all the functions dealing with user and group IDs belong in this category.

To use the *socket.library* functions, the first thing you have to do, of course, is open it. This library is a little unconventional because it returns a different library base for each OpenLibrary() call. The Shared Socket Library uses different library bases to keep track of some global data for each process that opens it. If you start a new process with a new context, the new process must open and initialize *socket.library*. Tasks should not access the *socket.library*, only processes should.

Before using any other function in the *socket.library*, you must call its function setup_sockets() to initialize the library:

```
ULONG retval = setup_sockets( UWORD max_sockets, int *errnop );
```

where max_sockets is the maximum number of sockets that can be open at once and errnop points to errno, a global integer that provides details about error conditions. This global value is used extensively by the standard socket functions. The standard Amiga C startup code (*c.o*) creates a global variable labelled ‘‘errno’’ which you can use as the global pointer.

The setup_sockets() call must be matched with the cleanup_sockets() call. This takes care of deallocating system resources that setup_sockets() allocates.

The *socket.library* assumes that all ints are 32-bit values. If you are using a compiler which doesn't use 32-bit ints, you must make sure that all ints are converted to longs by your code.

There are a couple of important differences between the AS225 *socket.library* and the standard Unix implementation. When writing softwarefor AS225, you cannot use the read(), write(), close(), and ioctl() functions on sockets. These functions come from Unix and apply both to files and sockets. To avoid confusion, *socket.library* does not contain these functions. Use the *socket.library* functions recv(), send(), s_close(), and s_ioctl() instead.

The standard Unix implementation has a series of get*() functions. These functions return a pointer to a static buffer. The buffer returned by a call to getX*() is cleared on the next invocation of getX*(). For example, the buffer pointed to by the return of gethostent() is cleared by another call to gethostent(), gethostbyaddr() or gethostbyname(), but not by a call to getprotoent(), etc. None of the get*ent(), set*ent() or end*ent() functions should normally be used except for porting existing programs.

The Shared Socket Library contains a function called selectwait(). This function combines the select() function with the *exec.library* Wait() function so that an Amiga networked application can wait on both Amiga events and network events at the same time.

This article, the examples from the Network Developer's disk, and the include files and the Autodocs should be enough to get you started. Writing network applications can be very complex and difficult, but is well worth the effort. This article only introduces you to writing network applications for the Amiga with AS225, and has left a lot unsaid about the socket interface and about networking in general. In addition to the Shared Socket Library include files and Autodocs, the following books and articles are all highly recommended. Several should be required reading for anyone seriously developing any Amiga and/or Unix network applications with TCP/IP:

References

Comer, D.E. (1991a), *Internetworking with TCP/IP, Volume I, 2d: Principles, Protocols, and Architecture*. Prentice-Hall, ISBN 0-13-468505-9

> If you want more detail on how the protocols work (especially how they support internetworking), this is the place to look. Little programming information is in this volume.

Comer, D.E. and Stevens, D.L. (1991b), *Internetworking with TCP/IP, Volume II: Design, Implementation and Internals*. Prentice-Hall, ISBN 0-13-472242-6.

> There is more detail here than most application developers will want or need, but some subjects (i.e., Out-Of-Band data) are covered here better than in any other text. The text includes a complete TCP/IP implementation for Xinu. It should be easily understandable by Amiga developers, in part, because Xinu happens to have a rather Exec-like IPC mechanism.

Stevens, W.R. (1990), *Unix Network Programming*. Prentice-Hall, ISBN 0-13-949876-1.

> This book starts at the beginning and methodically leads the reader through many advanced topics. If it weren't for the fact that it serves as a reference *and* a tutorial, it could be

thought of as the RKMs for AS225 software development. It introduces network protocols in some detail, and sockets in great detail. It includes source and discussion of several real-world examples: *ping*, *tftp*, *rlogin*, *lpr*, *rcmd*, *rmt*, etc. Everyone in the Amiga Networking group owns a copy.

RFCs

All Internet standards start life as Requests For Comments. They are still called RFCs even if they become required, recommended, or elective. If you wish to implement a standard Internet application, you should obtain any currently applicable RFC(s) and study them closely. Here is one way to obtain RFCs:

CSNET:

CSNET Coordination and Information Center (CIC)
Hotline: (617) 873-2777
10 Moulton Street, Cambridge, MA 02138
Email: cic@sh.cs.net
Info-Server requests to: info-server@sh.cs.net

The CSNET Info-Server stocks all RFCs with numbers higher than 900, unless (like RFC 900) they have been obsoleted by later RFCs. The Info-Server also stocks selected RFCs with numbers lower than 900.

The CSNET Info-Server is an automatic program that delivers information by electronic mail. To order a document from the Info-Server, send a message to ‘‘info-server@sh.cs.net’’. You do not need a subject field. The text of your message must be in a special format, such as:

```
REQUEST: rfc
 TOPIC: heLP
 Topic: RFC822
request: END
```

The text may any combination of upper-case and lower-case letters.

The above request asks for two documents ‘‘HELP’’ and ‘‘RFC822’’ from the collection ‘‘RFC’’. Your message must have a ‘‘REQUEST’’ line, and one or more ‘‘TOPIC:’’ lines to specify one or more documents. The optional statement ‘‘REQUEST: END’’ terminates your specification. Any subsequent text in the message is ignored by the Info-Server.

NOTICE: The Topic: field must be of the form ‘‘rfc822’’, and NOT ‘‘822’’ or ‘‘rfc822.txt’’.