

TCP/IP Programming on Unix and Win32

Philip Bradley

Forward

This document represents a work in progress. At the moment, I'm reviewing the sections on signals and polling and will give more complete coverage to asynchronous and non-blocking I/O. An example for the readers/writers problem will be created also. Most of the diagrams were created using MS Word and these are slowly being replaced (using Xfig). Formatting will be addressed generally when these immediate issues are resolved.

This document was originally prepared using Microsoft Word and has since been transferred to OpenOffice. The PDF version is generated using ps2pdf.

If you have any questions or comments regarding this document, please send them to tcpip@mobiustech.net.

Phil Bradley
March 2003

1. Introduction and Overview

The Basis of Network Programming

- The client server Model
- Transport and Application Protocols.

Network APIs

- Berkeley Sockets
- Winsock
- Transport Layer Interface
- Win32 Named Pipes and Mailslots
- Remote Procedure Calls

Distributed Programming Concepts

- Processes and Threads
- Concurrency

2. Introduction to TCP/IP Networks

The Software Components of a Network

- A Layered View of Network Software
- Hostnames and IP Addresses
- Domain Name Service (DNS)

The TCP/IP protocol stack

- The MAC layer, Ethernet and Token Ring
- The IP layer: Routing
- Data delivery via TCP and UDP
- Related protocols, ARP, ICMP
- Monitoring Network Packets

How network services are located

- Servers: Daemons and Services
- Well Known Ports and Protocols
- Naming services: DNS, NIS, WINS
- Resolver Functions

3. Connection Oriented Socket Programming

Establishing a Connection

- The Socket Model
- Socket Domains and Types
- Data Representation
- Socket Addresses and Related Structures
- Socket Creation
- Binding Local Names
- Connection Establishment
- Data Transfer
- Discarding Sockets

A simple Client and Server

- The client
- The server

WinSock Programming

- Berkeley to WinSock: An Overview
- Running the Client and Server on Windows

4. Connectionless Sockets and Special Options

Connectionless Socket Programming

- When to use UDP
- Sending and Receiving Datagrams
- Broadcasting Datagrams
- Coping with Unreliability

Advanced Socket Mechanisms

- Setting Socket Options
- Working with Raw Sockets
- Signaling
- Sending Out of Band Data
- Delivering Data Efficiently using Multicasting

5. Handling Concurrency

Process and Thread Concepts

- Managing UNIX Processes, fork() and exec()
- Creating Threads in Win32
- Creating Threads in UNIX
- Thread Synchronisation

Concurrency in the Server

- Iterative and Concurrent Servers
- Multiprocessing vs. Multithreaded Servers
- An Example Multiprocessing and Multithreading Server

Asynchronous and Non-blocking Sockets

- Asynchronous Sockets
- Integrating Socket I/O with Windows Messaging: WSAAsyncSelect()
- Applying Overlapped I/O to Sockets
- Waiting for I/O with select()

6. Client Server Design

Fundamental Choices

- Connection Oriented vs. Connectionless Transport
- Choosing Between Single-Threaded, Multi-Threaded and Multi-Processing Designs
- Blocking vs. Non-Blocking I/O

Recent Developments

- New Features in Winsock 2
- Microsoft's WinInet Library
- The Impact of IPv6

1

Introduction and Overview

The Basis of Network Programming

- The client server Model
- Transport and Application Protocols.

Network APIs

- Berkeley Sockets
- Winsock
- Transport Layer Interface
- Win32 Named Pipes and Mailslots
- Remote Procedure Calls

Distributed Programming Concepts

- Processes and Threads
- Concurrency

The basis of network programming

The client server Model

The client-server model is a common method of implementing distributed applications. Before we proceed, we should define the following terms:

- A *service* is any functionality, which a program may make available to other programs.
- A *server* is any program, which offers a service.
- A *client* is a program that avails of the services offered by a server.

Figure 1-1 shows a typical networked environment where different services are provided and used by client and server processes.

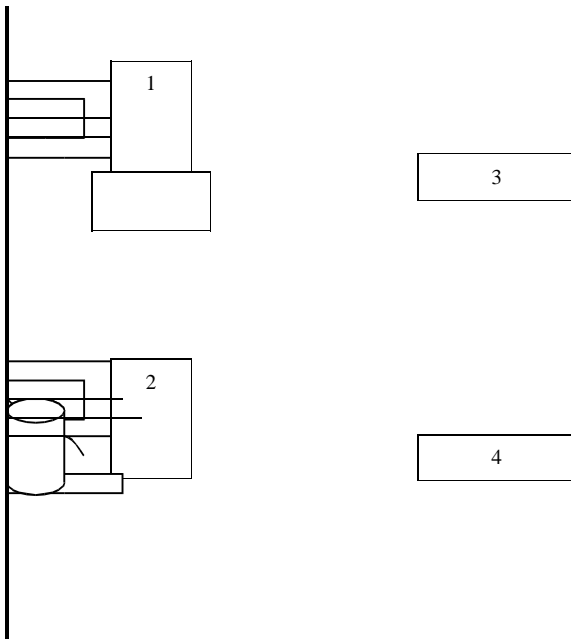


Figure 1-1 Typical Networked Environment

In the example above, hosts 1 and 2 are acting as servers. Host 1 is potentially a file server while host 2 is a database server. Host 3 is a client while host 4 is acting as both a client of hosts 1 and 2, and as a print server.

Note that a single machine may run multiple servers, for example, the file, print and database servers could all be run on the same host.

Transport and Application Protocols

A protocol is a set of conventions used by intercommunicating devices. For example, one protocol might specify the voltage levels and timing information used when communicating over the physical medium.

A different type of protocol might specify the communication primitives that the systems can use, for example the Simple Mail Transfer Protocol (SMTP) includes the following communication primitives:

```
HELO
MAIL FROM
RCPT TO
DATA
RSET
VRFY
EXPN
HELP
QUIT
```

These two protocols are obviously quite different in scope. In fact, there is no particular relationship between the communication primitives we use, and the underlying physical mechanism that transfers the data. We could change one without having any impact on the other.

This approach is called the *layered approach*. A number of models, which enumerate all of the potential layers at which we can define protocols, have been developed. The best known are the ISO/OSI stack and the TCP/IP stack. We explore these in detail in Chapter 2.

Two important layers are the *application layer* and the *transport layer*. SMTP is an example of an application layer protocol. Other examples of application layer protocols are HTTP and FTP. These protocols defined a language that the applications can use to talk to each other with. There is no consideration given to how the messages are delivered. This is left to lower level protocols.

The transport layer is significant because it is the level at which the programmer interfaces with the network. There are a number of different application programmer interfaces (API's) which provide the programmer with access to the transport layer. These are listed in the next section (Network API's). The services provided at the transport layer consist of a set of functions, which allow the programmer to exchange data between two machines. Delivery of this data may or may not be guaranteed, depending on the transport layer protocol selected. Transmission Control Protocol (TCP) provides reliable delivery. User Datagram Protocol (UDP) does not. TCP and UDP are discussed in detail in Chapter 2.

Network APIs

There are a number of different application programmer interfaces (APIs) designed for use in network programming. Each abstracts the communication between processes in slightly different ways.

The common interfaces are:

- Berkeley Sockets
- WinSock
- Transport Layer Interface
- Win32 Named pipes and mailslots
- Remote procedure calls

We will now examine these in more detail.

Berkeley Sockets

Sockets first appeared in the 4.1 release of the BSD (Berkeley Software Distribution) Unix operating system.

Sockets were designed to provide a consistent communication facility for processes regardless of whether or not the processes are running on the same machine. In addition, sockets were designed to hide as much detail as possible about how the underlying communication occurs.

The socket interface is protocol independent, that is, nothing in the interface requires the use of TCP/IP, however, they are used almost exclusively for TCP/IP communication on Unix systems. The sockets interface has been proposed as part of the POSIX 1003.12 Protocol Independent Interface standard.

The socket API is available on all BSD based systems and on most SVR4 systems (for example, Solaris, SCO Unix, Linux)

On BSD based systems, the sockets API is provided as a set of system calls. On other systems, the API is provided as a library. Apart from this, there should be no major issues porting code from one type of system to another.

WinSock

The Windows Sockets API (WinSock) is a specification originally developed by a consortium of companies to standardise TCP/IP communication on Windows.

WinSock is based on the sockets API on 4.3BSD and with some minor exceptions, provides the complete BSD interface, thus facilitating the porting of applications from Unix. WinSock also provides some extended (and non-portable) functions that are tailored specifically to the Win32 environment.

Microsoft ships a version of Winsock with Windows 95/98 and Windows NT. The advantages of these versions is that they are free and work directly with the operating systems' native network stacks. A few vendors still maintain alternative Winsock stacks, mainly on Windows 95/98 (for example Trumpet WinSock), but there are few advantages to these.

Windows sockets, like Berkeley sockets, provide protocol independence. In fact, WinSock serves as the programming interface in Win32 to several protocols, including:

- IPX/SPX from Novell
- NetBEUI (NetBIOS End User Interface)
- The AppleTalk protocol family for communication with Apple Macs.

Protocol Independence means that the application code consists of the same function calls regardless of what protocol is used. However, the programmer must specify the type of protocol at compile time.

WinSock takes the idea of protocol independence further with Protocol Transparency. This means that the protocol used can be determined at run time. This means that if a NetBEUI based network is replaced with a TCP/IP based network, then any existing network applications can run without modification.

The MFC classes CSocket and CAsyncSocket are built on top of the Win32 socket interface.

Transport Layer Interface

The Transport Layer Interface (TLI) is the primary networking interface used in Unix System V. The TLI presents a generic view of a network, corresponding to the transport level in the 7 layer OSI model (see chapter 2).

A transport level interface was chosen for two reasons:

- The transport layer is the lowest level that operates ‘*end to end*’. This means that it is the lowest level on the OSI stack at which an application can communicate over a network while remaining ignorant of it’s underlying topology.
- The services provided by the transport layer correspond to the services provided by many common network protocol families, for example, TCP/IP, XNS and SNA.

The architecture of the TLI consists of three elements as shown in figure 1-2.

The first element is the user-level library. This presents the programmer with the API and performs connection management and data transfer.

The second element is the transport provider (TPI). This is the module that actually performs the transport of data. In many cases, the TPI will just be the TCP/IP libraries.

The third element is the Transport Interface Module (TIMOD). The main function of TIMOD is to translate requests from TLI into requests to TPI. It is this architecture which establishes the protocol transparency of TLI.

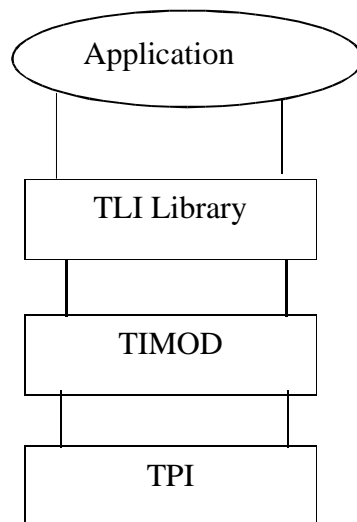


Figure 1-2 TLI Architecture

Win32 Named Pipes and Mailslots

The Named Pipes and Mailslots API's are high level, convenient programming interfaces for peer to peer communications in the Win32 environment (although OS/2 also supports these).

Named pipes and mailslots are implemented as Windows filesystem drivers. As a result, I/O takes place using the standard Win32 file I/O functions. The rest of the functions are concerned with making server applications available to clients and optimising certain types of exchange.

Named Pipes: The Server

In the Win32 environment, only NT can provide Named Pipes based server applications. Windows 9.x can only act as a client.

Named Pipes provide a connection-oriented service. This means that servers must be able to listen for incoming requests for communication. A Named Pipes server accomplishes this as follows:

- Create a Named Pipe with the call `CreateNamedPipe()`
- Wait for the client to request a connection by calling `ConnectNamedPipe()`

Named Pipes have a number of characteristics which are when the pipe is created, i.e. in the call to `CreateNamedPipe()`

These characteristics are as follows:

Direction: A named pipe can be two-way, client to server only or server to client only.

Read Mode: A named pipe can be read in byte or message mode. Byte mode, data is read and written as a stream of bytes. In message mode, each write to the pipe defines a discrete message, and each read must read the entire message.

Write Mode: A named pipe can write in either byte mode or message mode. A pipe may be written in message mode and read in byte mode, however, if the pipe is written in byte mode then it must be read in byte mode since there will be no message boundaries on the data.

Block Mode: Pipes can be either blocking or non-blocking. If the pipe is blocking the a call to `ConnectNamedPipe()` will block, otherwise, it will return immediately. Non-blocking pipes are generally considered obsolete and are only included for compatibility with OS/2.

Named Pipes: The Client

A client requests a connection to a named pipes based server by calling `CreateFile()` with an open named pipe.

A pipe is identified by its name and by the host on which the server is running. The format is the standard UNC format, that is:

```
\\hostname\PIPE\pipename
```

If the pipe is on the local host then you can use the notation:

```
\\. \PIPE\pipename
```

Technically there is no need to specify a local pipe this way since you could just use the first format and specify the local hostname however, there is a performance gain from using the second format due to the fact that this bypasses the network redirector.

The client can change the pipe attributes after calling `CreateFile()` by means of the `SetNamedPipeHandleState()` function call.

Having created the connection, the client and server may then communicate via the pipe by means of the `ReadFile()` and `WriteFile()` commands.

Mailslots

Mailslots provide a datagram extension to the connection-oriented service provided by named pipes. The interaction is very simple:

- The server creates a mailslot from which it can read only.
- The client opens a mailslot and can only write to it.

A mailslot can be created only on a local machine and the same process can actually use both the client and the server handle.

Mailslots can be created on Windows 9x and WinNT machines.

A server can create a mailslot using the `CreateMailslot()` function call. A client can access the mailslot using `ReadFile()`, as with named pipes.

The naming convention for mailslots is similar to that for named pipes, specifically, a mailslot is addressed as:

```
\\machinename\MAILSLOT\mailslotname
```

A mailslot may only be created locally, but may be accessed remotely. Thus, a call to `CreateMailslot()` must always specify a mailslot in the form:

```
\\. \MAILSLOT\mailslotname
```

Remote Procedure Calls (RPCs)

RPC is a method of client server communication that models the interaction between client and server as an API offered by the server in a completely location transparent way.

RPC is intended to be completely machine and operating system independent. For this reason, there must be a standard format for transmitting and receiving data structures. Converting arguments to and from this format is called *marshalling*.

The specific format that is used will depend on the type of RPC that you use. There are currently two main standards: Sun RPC and Microsoft RPC (based on OSF/DCE RPC). Unfortunately, the two are mutually incompatible. A remote procedure call works as follows:

The server offers its services by means of an API. Clients wishing to avail of these services make local function calls to the functions listed in the API. These functions do not implement the actual functionality described. Instead, they marshal the arguments, send (dispatch) them to the server and await a reply. On receiving the reply, the function unmarshals the results and returns them to the client.

These local functions are called *client stubs*. These client stubs can be generated automatically from an Interface Definition Language (IDL), for example, Sun RPC uses a standard called XDR (External Data Representation). Having specified the interface using the XDR language facilities, the client stubs may be generated using the `rpcgen` utility.

The following diagram illustrates the RPC process:

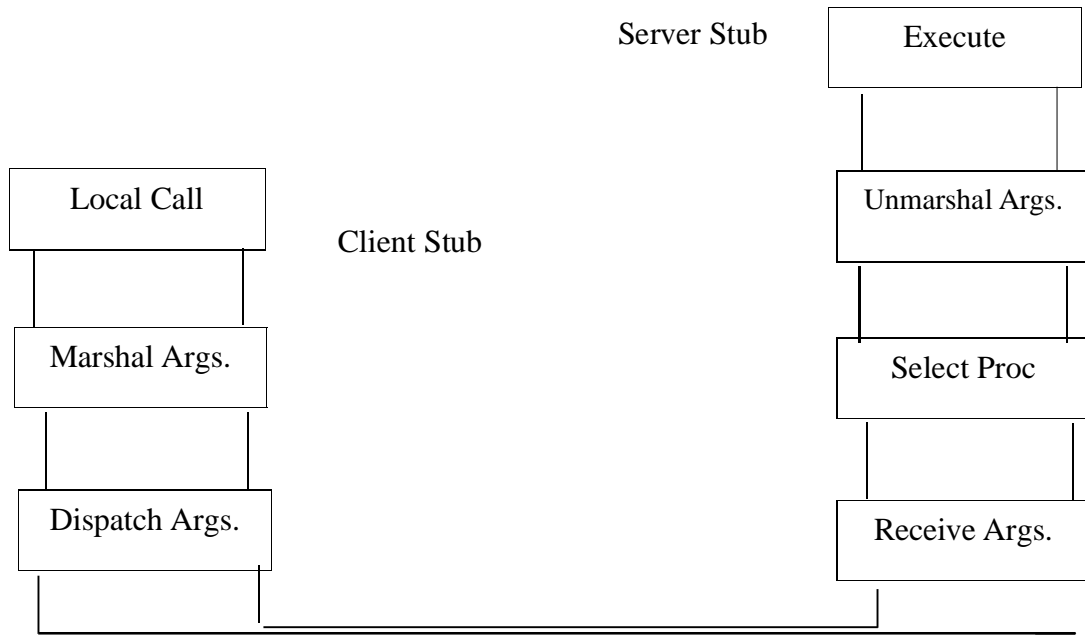


Figure 1-3 Structure of an RPC

Distributed Programming Concepts

Processes and Threads

When a program runs, it executes in the context of a process. A process consists of the program text (the actual instructions), the program data, a stack, a scheduling handle and some other state information.

A part of the kernel, called a scheduler, is responsible for allocating processor time to each process. Each process has a priority. The highest priority process gets run by the system. The priority of the running process is decreased over time so that it can't run indefinitely. This is called priority aging.

A process runs until its time expires or until it requests a resource that is unavailable. In the latter case it is said to *block*. Figure 1-4 below shows all of the possible process states and the state transitions that may occur.

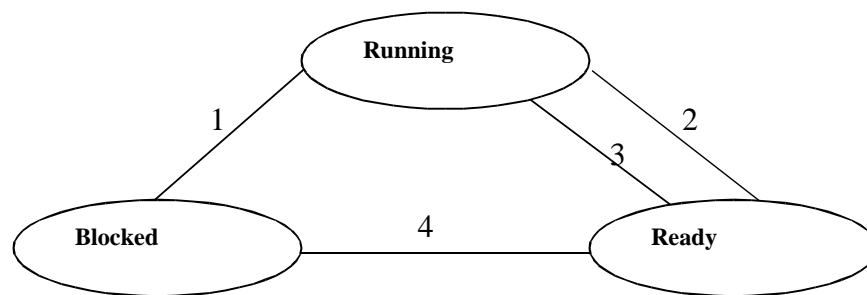


Figure 1-4 Process States and Transitions

The transitions have the following meaning:

1. Process blocks waiting for a resource
2. Scheduler picks another process
3. Scheduler picks this process
4. Resource becomes available

When a process blocks or its time expires, an interrupt is generated. This interrupt is serviced by a routine that loads the scheduler. The scheduler decides which process should run next and runs that. It is important to note that the scheduler is itself a process, that must be run periodically.

When the scheduler starts a process running, it replaces itself with that process. Before it does so, it sets a flag that causes an interrupt to be generated at some time in the future. This interrupt causes the scheduler to be loaded again thus ensuring that no process can lock out the system.

A process executes within an environment (also called a context). A context will contain the following information (this is not an exhaustive list)

- Registers
- Program Counter
- Process State
- Stack Pointer
- Time when process started
- CPU usage
- Process ID
- Pending signals
- Pointer to text segment
- Pointer to data segment

Every process has a separate context, and processes do not have access to the context of other processes. This means that the context must be saved when a process is stopped, and retrieved when started again. This (costly) operation is called a *context switch*.

Threads

Threads allow multiple tasks to be executed simultaneously within the context of a single process (threads are sometimes called lightweight processes or LWPs). For example, a user application might create a new thread to perform a task that is likely to take a long time for example, a database query. Execution of the query thread will be interleaved with execution of the main program. This means for example, that the user interface will be able to respond to input while the query is still running.

One of the reasons why threading is efficient is that a thread context is much smaller than a process context. The thread context consists of a local data segment, a program counter and little else so a thread switch is much less resource intensive than a process switch.

Threads execute within a process and so share the same global data. This gives rise to the standard concurrency problems described in the next section.

Concurrency

If our program uses multiple processes or multiple threads then it is said to be concurrent. Concurrency implies that a number of activities are being carried out simultaneously. Strictly speaking, this won't be the case if the machine is a single CPU machine. However, the operating system interleaves the execution of the tasks thus giving the user the illusion of simultaneous execution. In addition, the operating system ensures that what gets executed is logically equivalent to what would be executed if the tasks did run simultaneously.

When we use concurrency in our programs, we have to deal with the problem of *synchronisation*. Synchronisation means co-ordinating access to shared data so that programs don't interfere the operation of other programs.

If we use multiple processes then synchronising data access between these is not an issue. Each process has its own private data. If we want shared data then it must be explicitly created (for example, using Unix System V shared memory system calls) and accessed as such. The kernel ensures that this access is co-ordinated.

If we are using multiple threads then any data held by the controlling process is freely accessible to any of the threads as if it were local data. Access to this data must be explicitly co-ordinated (synchronised) by the programmer.

There are many synchronisation methods available, although the specific set will vary depending on the threads library that you use. We look at some of the common synchronisation mechanisms below.

Mutexes

A mutex is a structure that enforces Mutual Exclusion. A mutex can serve as a lock on a shared resource. Whenever a thread accesses the resource, it first sets the mutex. When it finishes, it releases the mutex. If the mutex is already set when a thread tries to set it, then it waits until the holding thread has released it.

A typical usage is shown below:

```
int shared_integer;
mutex shared_integer_mutex;

thread1                                thread2
lock(shared_integer_mutex)              // block until mutex is available
lock(shared_integer_mutex)
// do something with shared_integer     // do something with shared_integer
unlock(shared_integer_mutex)
unlock(shared_integer_mutex)
```

Semaphores

Semaphores are another mechanism of synchronisation. A standard problem that is amenable to being solved with semaphores is the Producer-Consumer problem.

Suppose we have multiple threads writing data to an area of shared memory (producers) and multiple threads removing data from the same area (consumers). A good real world example of this would be if multiple threads were accessing a linked list.

Producers can only add to the data if the data area is not full. Consumers can only remove from the area if the area is not empty. Obviously, we want to co-ordinate access so that add and remove operations don't interfere with each other.

Co-ordinating access is simple, we just set a mutex before we either add or remove, and release it when the operation is complete. This does nothing to solve the big problem however, how can we be sure that there will be any data in the list before we try to remove it, and how can we be sure that the list is not full before we add to it?

One attempt at this problem is shown in figure 1-5 below. We assume that the `add_element()` and `retrieve_element()` functions use a lock a mutex before they start, and unlock it when they finish.

<u>Producer</u>	<u>Consumer</u>
<pre>while (TRUE) do while (full) do // do nothing done e = produce_element() add_element(e) done</pre>	<pre>while (TRUE) do while (empty) do // do nothing done e = retrieve_element() consume_element(e) done</pre>

Figure 1-5 First Attempt at Solving the Producer Consumer Problem

This solution might appear to work at first glance however, it is unacceptable. Each thread loops until it's condition is satisfied and this is resource intensive. A better solution would be to block while waiting for the condition. Semaphores allow us to do this.

A semaphore acts a bit like a counter which keeps track of the number of threads that have reserved a resource for use. The programmer has no direct access to this count however. There are two primitives associated with semaphores. These are wait() and signal() (sometimes called P() and V()).

The wait() operation checks the value of the count within the semaphore. If it is non-zero then it is decremented, and the program continues as normal. If it is zero, then the thread sleeps.

The signal() operation increments the count within the semaphore. If there are any sleeping threads waiting on the semaphore then one is chosen at random by the system. (The system may or may not implement a FIFO scheme with waiting threads. This behaviour cannot be assumed and your program should not depend on it)

Now that we have a means of putting our producers and consumers to sleep while they are waiting, we will re-write the code from figure 1-5. This is shown in figure 1-6 below.

```
semaphore empty = 0
semaphore full = MAX_ELEMENTS
```

Producer

```
while (TRUE)
do
    // Block if full
    wait(full)

    e = produce_element()
    add_element(e)

    signal(empty)
done
```

Consumer

```
while (TRUE)
do
    // Block if full
    wait(empty)

    e = retrieve_element()
    consume_element(e)

    signal(full)
done
```

Figure 1-6 Solution to the Producer Consumer Problem with Semaphores

There are many other methods of synchronisation available but mutexes and semaphores are generally sufficient. The specific API calls available to programmers in Win32 and Unix are described in Chapter 5.

2

Introduction to TCP/IP Networks

The Software Components of a Network

- A Layered View of Network Software
- Hostnames and IP Addresses
- Domain Name Service (DNS)

The TCP/IP protocol stack

- The MAC layer, Ethernet and Token Ring
- The IP layer: Routing
- Data delivery via TCP and UDP
- Related protocols, ARP, ICMP
- Monitoring Network Packets

How network services are located

- Servers: Daemons and Services
- Well Known Ports and Protocols
- Naming services: DNS, NIS, WINS
- Resolver Functions

The Software Components of a Network

TCP/IP is a set of protocols that can provide either a reliable, streams based connection or an unreliable, packet (or datagram) based connection. The two protocols that provide this are TCP and UDP. Both use the underlying protocol IP, which is datagram based and unreliable.

User Datagram Protocol (UDP)

This allows programs to send data as packets. The packets are not guaranteed to arrive in the order in which they are sent. Some packets may not arrive at all. It is the responsibility of the application programmer to ensure that data is received and transmitted properly. This form of communication is called *Connectionless* (abbreviated to CLTS).

Transmission Control Protocol (TCP)

TCP provides a means for programmers to send data with the guarantee that the data will arrive in the same order in which it was transmitted. This is accomplished by means of a mechanism called *Positive Acknowledgement with Re-transmission* (PAR). It also provides error detection and correction. This form of communication is called *Connection Oriented* and is usually abbreviated to COTS.

A Layered View of Network Software

The International Standards Organisation has defined an architectural model used to describe the structure and function of data communications protocols.

This model is called the *Open Systems Interconnection Reference Model (OSI)*

The ISO/OSI reference model contains seven layers that define the functions of data communications protocols. Each layer represents a set of functions that is performed when data is transferred between applications across an intervening network.

The 7 layer model is shown in figure 2-1 below:

Application Layer: Protocols used by network applications for example FTP, HTTP.
Presentation Layer: Standardise Data Presentation to the applications (e.g XDR), compression, encryption.
Session Layer: Manages Sessions between applications.
Transport Layer: Provides end-to-end connection, flow control and error correction. (e.g TCP)
Network Layer: Manages connections across the network for upper layers.
Data Link Layer: Provides reliable data delivery across the physical link.
Physical Layer: Defines the physical characteristics of network medium.

Figure 2-1 The OSI 7 Layer Model

While the OSI model is useful, TCP/IP protocols do not match it's structure exactly. TCP/IP interacts with the OSI model in the following way:

- **Application layer**
The application layer is the level of the protocol hierarchy where user applications reside.
- **Presentation Layer**
This functionality is usually performed by TCP/IP applications for example, using htons/ ntohs and similar functions (see chapter 3 for details). Alternatively, protocols such as MIME or XDR may be used for presentation.
- **Session Layer**
The session layer is not identifiable as a separate layer in the TCP/IP stack. A session represents the path over which processes communicate. In TCP/IP, the session is represented as a socket and a port.
- **Transport Layer**
The transport layer in the OSI model provides end-to-end communication with full error correction. This functionality is provided by the Transmission Control Protocol (TCP) however, TCP/IP defines a further transport layer protocol called User Datagram Protocol (UDP). UDP provides a datagram based service with no guarantees about delivery.
- **Network Layer**
The Network Layer manages connections across the network, insulating upper layers from the network. It also provides addressing and delivery of data. This corresponds most closely to IP.
- **Data Link Layer**
This layer handles the reliable delivery of data across the physical network. An example of a data link layer protocol is IEEE 802.3 (Ethernet).
- **Physical Layer**
The physical layer defines the hardware required to transmit the data. Characteristics that might be defined at the layer include type of wiring, voltage levels, connection type etc.

Data is transferred from upper layers to lower layers until it is transmitted using the physical layer protocols across the network. On the receiving side, data is passed up through the layers until it reaches the application level. For this reason, the ISO/OSI model is sometimes referred to as the *protocol stack*.

TCP/IP is generally viewed as a stack with fewer layers than the OSI model. There is no universal agreement on how to represent TCP/IP as a stack but the diagram in figure 2-2 below is generally accepted.

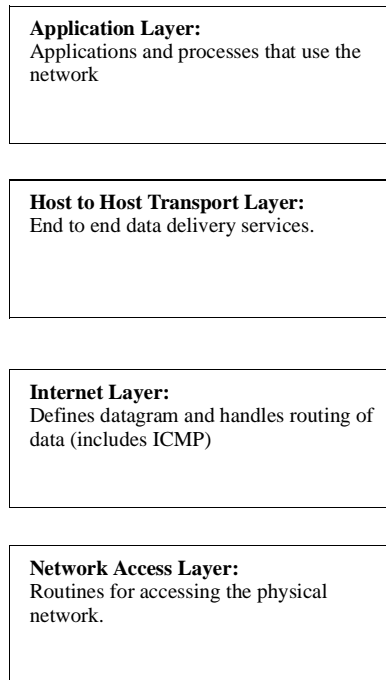


Figure 2-2 The TCP/IP Protocol Stack

Each layer in the stack adds control information to the data it receives. This information is placed in front of the data and is thus called a header.

The header contains information that allows each layer to do its job, for example, the IP adds an IP header to the data. The IP header contains information such as, the IP address of the source and destination host.

The transport layer adds sequence numbers (along with some other information) that allows it to keep track of the order of data as it arrives. These also allow it to detect duplicate packets.

Hostnames and IP Addresses

TCP/IP networks use a numbering scheme to provide a unique identification to each host on the networks. The number is a 32 bit number that is usually written as 4 *octets*, that is, 4 numbers between 0 and 255 (that is, $2^8 - 1$).

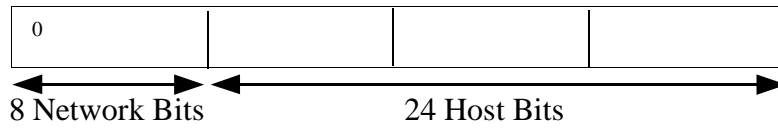
Thus logically, any combination between 0.0.0.0 and 255.255.255.255 is a valid IP address. In practice, not all combinations can be used as host IP addresses, since some have particular significance.

An IP address consists of a network portion and a host portion. Exactly where the boundary between the network portion and the host portion lies, is determined by the *class* of the network. IP uses the following rules to determine the class of the address:

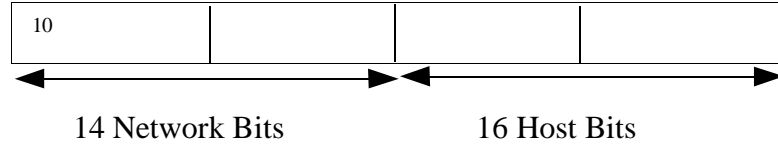
- If the first bit is 0 then it is an address in a class A network. The next seven bits identify the network and the remaining 24 bits identify the host. There are fewer than 2^7 (128) class A networks, each having up to 2^{24} (16 million) hosts.
- If the first 2 bits of the address are 10, then it is an address in a class B network. The next 14 bits identify the network and the remaining 16 bits identify the hosts. There can be a maximum of 2^{14} (16384) class B networks with each network having up to 2^{16} (65536) hosts.
- If the first 3 bits of the address are 110 then it is an address in a class C network. The next 21 bits identify the network. There can be a maximum of 2^{21} (2 million) class C networks with each network having 254 hosts.

The address class structure is summarised in figure 2-3 below.

Class A



Class B



Class C

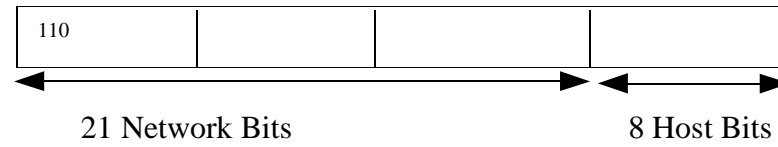


Figure 2-3 IP Address Class Structure

The IP class scheme is inflexible and makes inefficient use of the available namespace. An alternative scheme is called Classless InterDomain Routing (CIDR). Using this scheme, the boundary is determined using a mask. This is more flexible but has a greater administrative overhead.

Domain Name Service (DNS)

TCP/IP networks allow a host to be assigned a name. This name may be used instead of the IP address when establishing communication with the host.

In order to use hostnames, there must be mechanism for translating hostnames into IP addresses (and vice versa). In the early days of the Internet, this was done on Unix machines using a file called `/etc/hosts`. This file listed all the known hosts on the internet in a lookup table.

As the number of hosts grew, this scheme became unwieldy and has now been largely replaced with the DNS (Domain Name Service) system.

The DNS system is a distributed database. No single DNS server contains the DNS information for all hosts on the Internet. Central to the idea of DNS, is that a DNS server need only retain information about the hosts that are in its domain.

For example, suppose a host *clnt1* in domain `alpha.com` requires the IP address of machine *serv1* in domain `beta.com`. `clnt1.alpha.com` requests its local DNS server for the IP address of `serv1.beta.com`. Assuming that the local DNS server does not have this information, it will contact the authoritative server for `.com`.

The most likely response from the authoritative server will be the IP address of the DNS server for `beta.com`. The local DNS server can now query the DNS server at `beta.com` for the address of `serv1.beta.com`. This type of query is called a recursive query.

The local DNS server will then cache the reply for later use.

The TCP/IP protocol stack

The MAC layer, Ethernet and Token Ring

The MAC (Media Access Control) layer is the lowest layer above the actual physical medium (that is, the interface devices, transmission medium etc.) The MAC layer is described in the ISO model as the lower sub-layer of the Data Link Layer (the upper layer is the Logical Link Controller, or LLC).

The MAC mediates access to the physical medium, encapsulating all knowledge of the medium so that the LLC can ignore issues of medium control.

MAC addresses identify network entities in LANs that implement the IEEE MAC addresses of the data link layer. MAC addresses are unique for each LAN interface.

MAC addresses are 48 bits in length and are expressed as 12 hexadecimal digits. The first 6 hexadecimal digits, which are administered by the IEEE, identify the manufacturer or vendor and thus comprise the Organisational Unique Identifier (OUI). The last 6 hexadecimal digits comprise the interface serial number, or another value administered by the specific vendor.

Ethernet

Ethernet was developed at Xerox PARC in 1973. This was the first demonstrably feasible high-speed local area network (LAN). Ethernet is formally described in IEEE/ISO standard 802.3. This description is summarised below.

Ethernet operates at a transmission rate of 10Mb/s using low loss coaxial cable. The Ethernet is a bus network consisting of 1 or more segments of coaxial cable linked by repeaters. Nodes on the network are connected to the main segment by means of a drop cable, which attaches to the main cable by means of a T-connection or *tap*. There may be up to 4 repeaters (5 segments) of 100m in length. Thus an Ethernet based network may be up to 500m in total length.

Ethernet is a member of a class of networks described by the phrase '*Carrier Sensing, Multiple Access, Collision Detection*', abbreviated to CSMA/CD. CSMA/CD is implemented in Ethernet using the following scheme:

- Before a station attempts to send a packet, it checks to see if the cable is free (carrier sensing). If it senses another packet then it waits until it is free, otherwise it sends the packet.
- This mechanism cannot prevent all collisions, since two stations might still transmit at the same time. This is why a collision detection mechanism is used. Whenever a packet is transmitted, the network interface also listens on a loopback channel. The loopback channel returns exactly what the other stations on the network see. By comparing the data received on the loopback with the data transmitted, the station can determine whether a collision occurred.
- In the case where a collision has occurred, the sending station sends a *jamming packet* to alert the other stations of the collision and then waits a random amount of time before re-sending the data.

Ethernet is more commonly implemented using a bus-star topology. This hybrid scheme consists of nodes connecting to a central hub. The hub actually contains the same logic as a bus, collapsed into a single box. This topology results in greater network availability and easier administration since a single node can be removed without affecting the rest of the network. This is not the case with a simple bus topology.

This scheme (commonly called 10BaseT) requires an Ethernet hub, with individual connections from each station to the hub. Category 3 (Cat3) unshielded twisted pair (UTP) cabling is typically used to achieve a speed of 10Mb/s.

Fast Ethernet (IEEE 802.3u)

The 802.3u standard is an extension to the original Ethernet specification, it is not intended as a replacement. There are several implementations of Fast Ethernet. The most popular implementation is called 100BaseTX.

The 802.3u standard specifies a star topology using either twisted pair or fibre optic cable. There is no provision for using a bus topology or coaxial cable.

Fast Ethernet operates at 100Mb/s and is most commonly implemented using unshielded twisted pair (UTP) category 5 (Cat5) cable. Note that even if the hub operates at 10Mb/s, many installations will still use Cat5 cable. This is to allow an easy upgrade to 100Mb/s at a future date.

One disadvantage of Fast Ethernet is that, because of its higher speed, it is more sensitive to cabling defects. As a result, a single segment may not be longer than 100Mb/s. In addition, there may be only 2 repeaters with a maximum distance of 5m between them. This allows for a maximum network length of 205m.

Switched Ethernet

The original specification of Ethernet described a broadcast based system, that is, a station would send a packet which would be received by all the other stations on the network.

This results in a lot of needless broadcasting. A modification to this scheme allows for the hub to interpret the packets it receives and forward them only to their intended station. This results in a reduction in traffic and hence an increase in efficiency.

This scheme is called switched Ethernet.

Token Ring

Token ring is an alternative LAN technology to Ethernet. A standard for token ring is described in IEEE/ISO 802.5 specification.

A token ring based network consists of a ring based topology, around which, a special packet called a *token* is circulated. The token is circulated in one direction from station to station. When a station wishes to transmit, it waits until it receives the token (a 3 byte packet). It attaches its data as well as some header information (source and destination address, check sequence etc.) to the token and forwards it to the next station.

When the token is freed for use again, it is said to be *released*. In 4Mb/s networks, the original sending station performs the token release when it receives the last bit of the packet that it originally sent. In 16 Mb/s networks, the token is released after the last bit of the packet has been transmitted (early token release).

The IP layer: Routing

Routing is the act of moving information across an internetwork from a source to a destination. Along the way, at least one intermediate node typically is encountered.

Gateways are hosts that route data between networks. Hosts, even when they are not routers, generally must make some routing decisions. This is usually as follows:

- If the destination host is on the local network, then deliver it directly.
- If the destination host is on a different network then deliver it to the gateway.

The choice of gateway to deliver to is determined by means of a *routing table*. This table lists gateways to use for particular network addresses as well as a default route. The gateways that it lists are on networks directly connected to the system. As a result, the routing table cannot specify complete routes. It can only specify the *next hop*.

As the data moves from one gateway to another, it should eventually reach a gateway that is directly connected to the destination network. This gateway then delivers the data directly to the destination host.

The original Internet was built on top of the ARPANET. The ARPANET was the backbone of the system and this, along with the gateways the interconnected it, was referred to as the core. The core gateways maintained routing information for the entire Internet. They exchanged information using the *Gateway to Gateway Protocol (GGP)*.

This architecture is now obsolete. The new routing model is based on a collection of *Autonomous Systems*. An autonomous system is a collection of networks and gateways with its own internal mechanism for collecting routing information. These autonomous systems are also referred to as *Routing Domains*. Routing domains exchange information with one another using exterior routing protocols such as Border Gateway Protocol (BGP). This information is also called *reachability information*. It is important to note that there is no central authority that can be used to determine the best route to take.

BGP is an example of an exterior routing protocol. Within an autonomous system, interior routing protocols are used to collect routing information.

Routing protocols may be classified as either link state or distance vector protocols. The Routing Information Protocol (RIP) is a distance-vector protocol that uses hop count as its metric. RIP is widely used for routing traffic in the global Internet and is an interior gateway protocol (IGP), which means that it performs routing within a single autonomous system.

OSPF is a link-state routing protocol that calls for the sending of link-state advertisements (LSAs) to all other routers within the same area. Routers accumulate link-state information, which they use to calculate the shortest path to each node.

Data delivery via TCP and UDP

Applications that require the transport protocol to provide reliable data delivery use TCP because it verifies that the data is delivered across the network accurately. TCP is a reliable, connection oriented, byte stream protocol. TCP provides reliability using a method called *Positive Acknowledgement with Retransmission* (PAR).

Under this scheme, the sender keeps resending data until it receives an acknowledgement. The data is sent along with a checksum, which the receiver uses in order to check that no errors occurred during transmission. If an error has occurred then the receiver silently drops the packet.

Every packet sent contains a *sequence number*. Duplicate packets will have the same sequence number. This allows the receiver to detect duplicate packets, which are also dropped silently. The sequence numbers also allow TCP to correctly order packets regardless of what order they arrive in.

A connection is established by means of a *three-way handshake*. This process is summarised in figure 2-4 below.

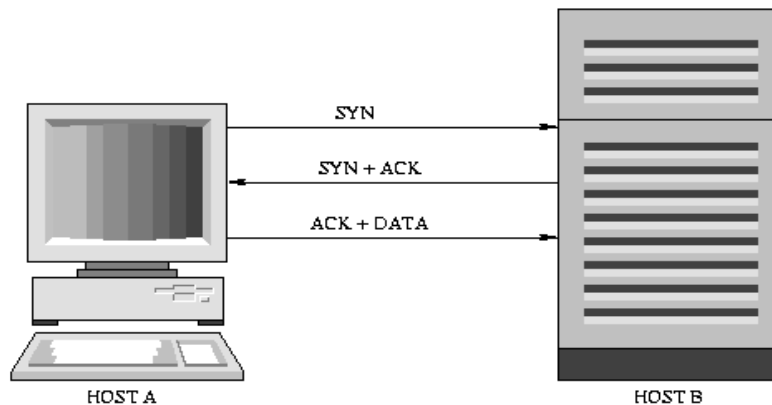


Figure 2-4

The client initiates the connection by sending a segment with the Synchronise sequence number (SYN) bit, set, and the sequence number at which it wants to start. The server responds with a segment that also has the SYN bit set and it provides the client with its starting sequence number. Subsequent segments exchanged between client and server will not have the SYN bit set. All packets after the first sent, will have the ACK bit set.

UDP is an unreliable, connectionless datagram based protocol. UDP allows applications to exchange messages over the network with a minimum of protocol overhead. The lack of error detection and guarantee of delivery means that UDP is often considered a *lightweight* protocol.

Application programmers often choose UDP when the amount of data being transmitted is small and doesn't justify the overhead of TCP. Applications that employ a query response scheme are also candidates for using UDP. Generally the response serves as an acknowledgement while the client just re-transmits the request if it doesn't receive the response within a timeout period.

Related protocols, ARP, ICMP

ARP

IP datagrams are delivered to the correct network using the IP address and the routing table. The physical network has its own addressing scheme and the IP address of any incoming packet must be mapped to its corresponding physical network address. This is the MAC address referred to earlier.

The protocol that performs the translation of IP addresses to Ethernet addresses is called *Address Resolution Protocol* (ARP), and is defined in RFC 826.

ARP maintains a lookup table of translations from IP addresses to Ethernet addresses. This table is built up dynamically. When ARP receives a request, it performs a lookup in its table. If the IP address is found then it returns the corresponding MAC address. If it is not found, then it broadcasts a packet containing the IP address it wishes to locate. If a receiving host identifies the IP address as its own then it responds by sending its MAC address. This response is then cached in the lookup table.

On Unix systems, the arp command can be used to display the local ARP table. On some systems, you may have to use the arp -a command to see the complete table.

ICMP

Internet Control Message Protocol (ICMP) is the part of IP used to co-ordinate the transfer of data between hosts on a TCP/IP based network. ICMP specifies a set of messages used for this purpose. These messages are used to perform the following functions:

Flow Control: When a host receives data faster than it can process, it sends an ICMP *Source Quench Message* to the sending station. This causes the sender to pause the sending of datagrams.

Detecting unreachable destinations: When a destination is found to be unreachable by a system, it sends a *Destination Unreachable Message* to the sender. If the destination in question is a host or network, then the sender of the ICMP message will be the intermediate gateway. If the destination is a port, then the sender of the message will be the destination host.

Redirecting Routes: Gateways use the *ICMP Redirect Message* to tell the sender to use a different gateway. A gateway will send a redirect message if it considers the other gateway to be a better choice, for example, if the other gateway is lightly loaded.

Checking remote hosts: A host can send an *ICMP Echo Message* in order to check if a remote system is properly running IP. When a host receives an ICMP Echo packet, it responds with that packet. The *ping* command uses this message.

Monitoring Network Packets

The 802.3 (Ethernet) standard is described in the section on Ethernet. In this scheme, a central hub is used to connect the network nodes. A simple (shared) hub works as follows:

The sending station sends its message to the hub. The packet contains the MAC address of the recipient. When the hub receives the message, it sends it to all the stations that it is connected to. Each receiving station examines the MAC address on the packet. By default, stations drop any packets not addressed to them.

It is possible to monitor network traffic on an Ethernet by changing the way in which the Ethernet card on your host operates. The card can be set to operate in *promiscuous mode*. When in this mode, the card will display all traffic that it receives, not just traffic addressed to it.

On Unix, a card can be set to promiscuous mode using the *ifconfig* utility (see man pages for details). Traffic can then be monitored using the *snoop* utility. Note that you need root access on the host in order to change the configuration of the card and to run snoop.

Servers: Daemons and Services

On Unix systems, most servers are implemented as *daemons*. A daemon is a non-interactive process, with no associated terminal, that listens for network connections. In order to avail of a network service on a Unix machine, the daemon that provides the service must be running. Examples of daemons commonly found on Unix systems are ftpd, telnetd, httpd etc.

Daemons can be configured to start automatically at boot time. Alternatively, they can be configured to start up when a client requests the service. This latter option is achieved using a generic server called the Internet daemon (or inetd). The inetd is an example of a *port monitor*. This means that it can listen on multiple ports for requests. Inetd starts up the appropriate server only when it receives a request for it.

On NT, servers are usually implemented as *services*. These are the NT equivalent of daemons. Services can be configured so that they are automatically started at boot time.

A service is really only a console application that follows some conventions in order to conform to the NT Service Control Manager scheme. Most NT services are implemented using RPC (Remote procedure calls) although this is not strictly necessary.

Well Known Ports and Protocols

When data arrives at a specific host, it must be sent to the correct process. The TCP/IP mechanism allows for multiple processes using multiple protocols to communicate over the same physical medium. In order to deliver each incoming packet to the appropriate process, the TCP/IP software examines the protocol number and the port number.

The protocol number indicates which transport level protocol is being used. On a Unix system, these numbers are defined in the `/etc/protocols` file. An extract from such a file is shown below.

```
Ip      0   IP      # Internet protocol
icmp    1   ICMP    # Internet Control Message Protocol
ggp     3   GGP     # Gateway Gateway Protocol
tcp     6   TCP     # Transmission Control Protocol
udp     17  UDP     # User Datagram Protocol
```

Using the above table, the data is passed to the appropriate transport layer protocol. The transport layer examines the port number and passes the data to the process that is listening on that port.

There is a conventional mapping between port numbers and standard services similar to that between protocol numbers and protocols. The sender of the data does not know for certain what server (if any) they are going to connect to. They only know that they are connecting to whatever server process listens to the port they specify. However, there is a good chance that the host they connect to follows the standard conventions. For example, the FTP server usually listens on port 21. It is reasonably safe to assume that by connection to port 21, you will connect to the FTP server on the destination machine.

On Unix systems, the mapping between port numbers and services is maintained in the `/etc/services` file. An extract from a standard services file is shown below:

```
echo      7/tcp
discard   9/tcp
discard   9/udp
chargen   19/tcp
ftp-data  20/tcp
ftp       21/tcp
telnet    23/tcp
smtp      25/tcp
```

A complete listing of protocol and port numbers is available in the *Assigned Numbers* RFC.

Naming services: DNS, NIS, WINS

DNS

The implementation of DNS on most systems is the Berkeley Internet Name Domain (BIND) server. While originally written for Unix, BIND is also used on many NT systems.

A DNS system is conceptually divided into two components: a resolver and a name server. The resolver is a set of functions used by client programs that need to make DNS queries. The resolver functions form the query and present it to the name server. The name server looks up the address and makes the appropriate response.

Hosts that use the resolver functions but not the name server are called *resolver only* systems.

NIS

Network Information Service (NIS) is an administrative database system, developed by Sun Microsystems. NIS provides centralised control and automatic dissemination of important system information for Unix systems. Much information that is typically held in files the `/etc` directory on standalone machines, can be stored in a NIS database (also called a NIS map). A NIS server performs a role analogous to a Primary Domain Controller (PDC) on an NT network.

Typical examples of information that might be stored in NIS maps include the contents of the `/etc/hosts` and `/etc/networks` files. Usernames and passwords may also be stored in NIS maps. Once the account has been created in the NIS database, the user may log on to any NIS client in that NIS domain.

NIS may also provide mapping between IP addresses and hostnames and is often used in conjunction with DNS. However, NIS is not an alternative to DNS. NIS was designed only to provide service to LANs, not to the Internet as a whole.

Sun Microsystems have developed a system called NIS+ that was intended to improve on NIS. While NIS+ was a complete redesign, it can emulate NIS for backward compatibility. Adoption of NIS+ has been very slow as it is generally considered to be overly complex and error-prone.

WINS

The NetBEUI suite (NetBIOS Extended User Interface) is the foundation of Microsoft networking. NetBEUI consists of the following components:

The NetBIOS API

This is the principal API for Microsoft networking. NetBIOS was originally developed as a set of BIOS routines but is now implemented in software.

Server Message Block (SMB) protocol

This defines a series of commands that can be used to communicate network information between hosts. SMB is in some ways analogous to ICMP in TCP/IP networks.

NetBIOS Frame Buffer (NBF) protocol

Builds NetBIOS frames for transmission over the network.

The terms NetBEUI and NetBIOS are frequently used interchangeably although they are not synonymous.

NetBEUI was designed for use on small homogeneous networks. It is lightweight and fast, but it doesn't scale to enterprise or WAN applications. One of the main reasons for this is that NetBEUI is non-routable and hence, is constrained to a single physical network. One solution to this lack of scalability that allows users to continue using NetBEUI is to run the NetBIOS over TCP/IP (NetBT) protocol.

In a NetBEUI network hosts are identified by their Windows computer name (also called NetBIOS computer name), while on TCP/IP networks, IP addresses are used. When using NetBT, there must be some means of translating between the two naming schemes. On small networks, this can be done by maintaining a table in a file called LMHOSTS. This scheme does not scale well however. A more scalable solution is the Windows Internet Name Service (WINS).

Many Microsoft networks make use of DHCP (Dynamic Host Configuration Protocol). As a result there is no permanent mapping between NetBIOS names and IP addresses. WINS can be configured so that whenever a host boots, it *registers* with the WINS service. This allows hosts to be properly identified despite not having a permanent IP address, without any ongoing administration overhead. WINS is specifically designed to work with the Microsoft DHCP server to accomplish this.

Resolver Functions

The architecture of the BIND system is described in the previous section on DNS. DNS clients use the resolver functions in order to make DNS queries. These are of the form getXbyY. These functions are described below. Note that the prototypes and structures described are common to BSD and Winsock implementations.

Host resolver functions

The host resolver function prototypes are listed below.

```
struct hostent* gethostbyaddr(char* addrp, int len, int type)
struct hostent* gethostbyname(char* name)
```

The gethostby* functions return a pointer to a hostent structure. A hostent structure has the following definition:

```
struct hostent {
    char*   h_name;           // Hostname
    char*   h_aliases;       // NULL-terminated array of alternate names
    short   h_addrtype;      // The type of address being returned
    short   h_length;        // The length of each address in bytes
    char*   h_addr_list;     // NULL-terminated list of host addresses
};
```

Addresses are returned in network byte order. The macro h_addr is defined to be h_addr_list[0] for compatibility with older software.

Network resolver functions

The network resolver function prototypes are listed below.

```
struct netent* getnetbyaddr(long net, int type)
struct netent* getnetbyname(char* name)
```

The netent structure has the following definition.

```
struct netent {
    char*   n_name;          // Official name of network
    char**  n_aliases;      // Alias list
    int     n_addrtype;     // Network number type
    long    n_net;          // Network number
}
```

Service resolver functions

The prototypes for these functions are listed below.

```
struct servent* getservbyport(int port, int proto)
struct servent* getservbyname(char* name, char* proto)
```

These functions return a pointer to a servent structure. servent has the following definition:

```
struct servent {
    char*    s_name;        // official service name
    char**   s_aliases     // alias list
    int     s_port         // port at which the service listens
    char*    s_proto       // protocols to use
}
```

Protocol resolver functions

The prototypes for these functions are listed below.

```
struct protoent* getprotobynumber(int proto)
struct protoent* getprotobyname(char* name)
```

These functions return a pointer to a protoent structure. protoent has the following definition:

```
struct protoent {
    char*    p_name;        // official protocol name
    char**   p_aliases     // alias list
    int     p_proto;       // protocol number
}
```

3

Connection Oriented Socket Programming

Establishing a Connection

- The Socket Model
- Socket Domains and Types
- Data Representation
- Socket Addresses and Related Structures
- Socket Creation
- Binding Local Names
- Connection Establishment
- Data Transfer
- Discarding Sockets

A simple Client and Server

- The client
- The server

WinSock Programming

- Berkeley to WinSock: An Overview
- Running the Client and Server on Windows

Establishing a Connection

The Socket Model

The basic building block for network communication is the socket. A socket is an endpoint of communication to which a name may be bound. Each socket in use has a type and one or more associated processes. Sockets exist within communication domains. A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets.

Sockets were designed to implement the client-server model of interprocess communication where:

- The interface to network protocols needs to accommodate multiple communicating protocols, such as TCP/IP, XNS and UNIX domain.
- The interface to network protocols needs to accommodate server code that waits for connections and client code that initiates connections.
- The application must operate differently depending on whether communication is connectionless or connection oriented.

A socket descriptor is a value associated with a open network connection. It is similar in function to a file descriptor. A socket descriptor is created using the `socket ()` function call.

Socket Domains and Types

When you create a socket, you must specify a domain for that socket. A domain is sometimes called an *address family* or *protocol family*. When programming TCP/IP based applications, the domain to use will be the Internet Domain, represented by the constant `AF_INET`. Table 3-1 below lists some commonly used domains.

Constant	Protocol
<code>AF_INET</code>	Internet communication (TCP/UDP etc.)
<code>AF_UNIX</code>	Local Unix hosts communication
<code>AF_IPX</code>	NetWare (IPX/SPX) communication
<code>AF_APPLETALK</code>	AppleTalk communication
<code>AF_NETBIOS</code>	NetBIOS communication

Table 3-1

Having selected the domain, you must now select what *type* of socket you want. The two most common types are `SOCK_STREAM` and `SOCK_DGRAM`.

Selecting `SOCK_STREAM` creates a *Stream Socket*. Stream sockets are reliable two-way connected communication streams. If you output two items into the socket in the order "1, 2", they will arrive in the order "1, 2" at the opposite end. They will also be error free. Stream sockets use TCP in order to provide the connection oriented service.

Selecting `SOCK_DGRAM` creates a datagram socket. This provides a connectionless, unreliable service. If you send a datagram, it may not arrive, or it may arrive out of order. If the datagram does arrive then the data within will be error free. Datagram sockets use UDP to provide this service.

Another type of socket, called a *Raw Socket* may also be used. These are created as type `SOCK_RAW`. Raw sockets are rarely used in regular application. They are sometimes used when developing new protocols or testing esoteric features of existing protocols. On Unix systems, only applications running as root can create raw sockets.

Finally, a *Sequenced Packet Socket* may be created by setting the type to be `SOCK_SEQPACKET`. A sequenced packet socket is similar to a stream socket, with the exception that record boundaries are preserved. Sequenced packet sockets are very rarely used and are only mentioned for completeness.

Data Representation

Data elements that consist of more than one byte may be represented using one of two possible *byte orderings*. These are Most Significant Byte first (MSB) and Least Significant Byte first (LSB).

MSB is also called Network Byte Order (NBO) since this is the order in which data is transmitted over the network. The host may store its data natively in either MSB or LSB. The host format is called Host Byte Order (HBO).

Certain values need to be explicitly converted to NBO before transmission. These will be pointed out wherever they occur.

If your hardware represents data in NBO anyway then technically you don't need to convert between NBO and HBO. It is good practice to explicitly convert to network order anyway since this makes your code more portable.

The routines used for converting between HBO and NBO are:

- htons() "Host to Network Short"
- htonl() "Host to Network Long"
- ntohs() "Network to Host Short"
- ntohl() "Network to Host Long"

Socket Addresses and Related Structures

The structure `sockaddr` holds socket address information for many types of sockets. It looks as follows:

```
struct sockaddr {
    unsigned short    sa_family;    /* address family, AF_xxx */
    char              sa_data[14]; /* 14 bytes of protocol address */
};
```

`sa_family` is the address family, or domain as described earlier. `sa_data` contains a destination address and port number for the socket.

This scheme is cumbersome since two pieces of information are being stored in the `sa_data` array. An alternative to this structure is the `sockaddr_in` structure described below (the `in` suffix signifies *internet*). Note that these two structures are the same size. The `sin_zero` field is used as padding to ensure this.

```
struct sockaddr_in {
    short int         sin_family;   /* Address family */
    unsigned short int sin_port;    /* Port number */
    struct in_addr    sin_addr;     /* Internet address */
    unsigned char     sin_zero[8]; /* Same size as struct sockaddr */
};
```

The `sin_addr` field is intended to contain an IP address. For historical reasons, rather than just declare `sin_addr` as the appropriate type, `sin_addr` is declared as a `struct in_addr`. This structure is shown below:

```
// Internet address (a structure for historical reasons)
struct in_addr {
    unsigned long s_addr;
};
```

As a result of this peculiarity, in order to access the IP address of a socket address, we must use the construct below:

```
struct sockaddr_in s;  
...  
s.sin_addr.s_addr = value;
```

Notice that `sockaddr_in` contains the same information as `sockaddr`. The difference is that the port and address are separated into two fields in `sockaddr_in`.

The `sin_zero` field is used to pad the structure so that it is the same size as a `sockaddr`. This means that a `sockaddr_in*` can be cast as a `sockaddr*`. This comes in quite useful later.

Socket Creation

To create a socket the *socket* system call is used:

```
s = socket(domain, type, protocol);
```

This call requests that the system create a socket in the specified domain and of the specified type. A particular protocol may also be requested. If the protocol is left unspecified (a value of 0), the system will select an appropriate protocol from those protocols which comprise the communication domain and which may be used to support the requested socket type.

The user is returned a descriptor (a small integer number) which may be used in later functions which operate on sockets.

To create a stream socket in the Internet domain the following call would generally be used:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

This call would result in a stream socket being created with the TCP protocol providing the underlying communication support. To create a datagram socket the call would generally be:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The default protocol (used when the protocol argument to the socket call is 0) should be correct for most every situation. However, it is possible to specify a protocol other than the default; this will be covered later.

The `socket()` call returns -1 if an error occurs. There are several reasons a socket call may fail. Aside from the rare occurrence of lack of memory (ENOBUFS), a socket request may fail due to a request for an unknown protocol (EPROTONOSUPPORT), or a request for a type of socket for which there is no supporting protocol (EPROTOTYPE).

Binding Local Names

A socket is created without a name. Until a name is bound to a socket, processes have no way to reference it and, consequently, no messages may be received on it.

Communicating processes are bound by an association. In the Internet domain, an association is composed of local and foreign addresses, and local and foreign ports. There may never be duplicate <protocol, local address, local port, foreign address, foreign port> tuples.

The `bind()` function allows a process to specify the local address and local port. The `connect` and `accept` primitives are used to complete a socket's association.

Binding names to sockets can be fairly complex. Fortunately, it is usually not necessary to specifically bind an address and port number to a socket, because the `connect()` and `send()` calls will automatically bind an appropriate address if they are used with an unbound socket. Generally it makes sense to bind in a server but not to in a client.

The `bind()` function is used as follows:

```
bind(s, name, namelen);
```

The bound name is a variable length byte string which is interpreted by the supporting protocol(s). `bind()` returns -1 on error.

Specifically, the `bind()` function is called as follows:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in sin;
...
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

Notice the casting of the `sin` variable. We have not yet said what values should be contained in the `sockaddr_in` structure. We will examine this now.

The `sockaddr_in` structure has three fields, `sin_family`, `sin_port` and `sin_addr`. We must provide a value for `sin_family`. This is the address family, or domain in which we want to create our socket. Valid values for this field are listed in table 3-1.

`sin_port` is the local port number to which we want to bind. We may set this value to 0 in which case, the system will select a random non-privileged (ie. >1024) port.

`sin_addr` is the IP address of the local machine. We may set the address to be `INADDR_ANY`. The system interprets this to mean any valid IP address for the machine. This is particularly useful if the machine has more than one network interface since a server can listen on any of the networks.

Connection Establishment

Connection establishment is usually asymmetric, with one process acting as a client, and the other acting as a server.

The server, when willing to offer its advertised services, binds a socket to a well-known address associated with the service and then passively *listens* on its socket. It is then possible for an unrelated process to rendezvous with the server. The client requests services from the server by initiating a *connection* to the server's socket.

On the client side the connect call is used to initiate a connection. This might appear as below:

```
struct sockaddr_in server;  
...  
connect(s, (struct sockaddr *)&server, sizeof (server));
```

where server in the example above would contain the Internet address and port number of the server to which the client process wishes to speak.

If the client process's socket is unbound at the time of the connect call, the system will automatically select and bind a name to the socket if necessary. This is the usual way that local addresses are bound to a socket.

If the connection was unsuccessful then -1 is returned and `errno` is set (any name automatically bound by the system, however, remains). Otherwise, the socket is associated with the server and data transfer may begin.

In order for the server to receive a client's connection it must perform two steps after binding its socket. The first is to indicate a willingness to listen for incoming connection requests:

```
listen(s, 5);
```

The second parameter to the listen call (*backlog figure*) specifies the maximum number of outstanding connections which may be queued awaiting acceptance by the server process; this number may be further limited by the system.

With a socket marked as listening, a server may accept () a connection:

```
struct sockaddr_in from;
...
fromlen = sizeof (from);
newsock = accept(s, (struct sockaddr *)&from, &fromlen);
```

A new descriptor is returned on receipt of a connection (along with a new socket). If the server wishes to find out who its client is, it may supply a buffer for the client socket's name.

The parameter `fromlen` is initialized by the server to indicate how much space is associated with `from`, then modified on return to reflect the true size of the name. If the client's name is not of interest, the second parameter may be a null pointer.

As might be expected, both `listen()` and `accept()` return -1 on error and set `errno` appropriately.

The procedures for creating client and server application are summarised in Figure 3-1 below:

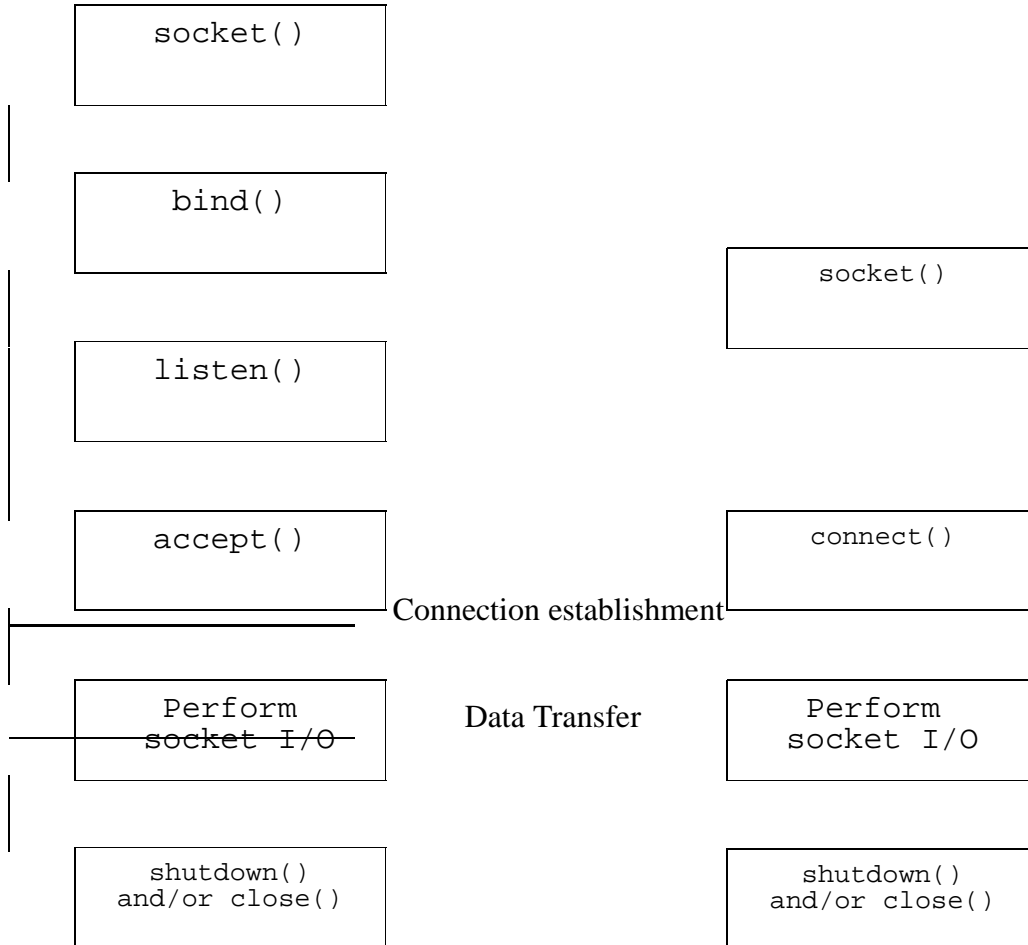


Figure 3-1

While there are quite a number of steps involved, the interface is reasonably intuitive and quite consistent. These simple steps are the basis of all client/server sockets programming.

Data Transfer

A socket descriptor is treated similarly to a file descriptor by the Unix kernel. This means that we perform the same *file* type operations on sockets. In particular, we can use the normal `read()` and `write()` system calls:

```
write (sockfd, buf, sizeof(buf));
read (sockfd, buf, sizeof(buf));
```

While the above system calls will work, it is better to use the socket specific `send()` and `recv()` system calls. These have the following usage:

```
send (sockfd, buf, sizeof(buf), flag);
recv (sockfd, buf, sizeof(buf), flag);
```

As you can see, the usage is almost identical to that of `read/write`. The only difference is the flag argument. The flag argument can usually be set to 0. The flags argument may take 1 or more of the following arguments:

`MSG_PEEK`

Peek at the incoming data. The data is copied into the buffer but is not removed from the input queue.

`MSG_OOB`

Process out-of-band data (See chapter 4 for details).

`MSG_DONTROUTE`

Send data without routing packets

You can pass multiple arguments by OR'ing the values together. As before, you should check for a return of -1 and use `errno` to discover which error occurred.

Discarding sockets

When a socket is no longer of use, it may be discarded by using the close operation.

```
close(sockfd);
```

If data is associated with a socket, which promises reliable delivery (e.g. a stream socket) when a close takes place, the system will continue to attempt to transfer the data. However, after a fairly long period of time, if the data is still undelivered, it will be discarded.

In order to control how the socket handles pending data, you can call `shutdown()` before you call `close()`. This call is of the form:

```
shutdown(sockfd, how);
```

where *how* can take the following values:

- 0 don't read any pending data
- 1 don't send any pending data
- 2 don't send or receive any pending data

A simple Client and Server

The Client

Having seen all the main function calls required to set up sockets, we are now in a position to create a basic client and server. We will create a client first. The following code will work on most Unix machines. We look at creating a Win32 client later.

```
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

/* Connect to a predefined port on a user specified host */

const int PORT=3490;          /* the port client will be connecting to */
const int MAXDATASIZE=100;   /* max number of bytes we can get at once */

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in remote_addr; /* connector's address information */

    if (argc != 2) {
        fprintf(stderr,"usage: clnt hostname\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) { /* get the host info */
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    remote_addr.sin_family = AF_INET; /* host byte order */
    remote_addr.sin_port = htons(PORT); /* short, network byte order */
    remote_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(remote_addr.sin_zero), 8); /* zero rest of the struct */

    /* Connect to address in remote_addr */
    if (connect(sockfd, (struct sockaddr *)&remote_addr, sizeof(struct sockaddr)) == -1) {
        perror("connect");
        exit(1);
    }

    if ((numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1) {
        perror("recv");
        exit(1);
    }

    buf[numbytes] = '\0'; /* We must null terminate the buffer */
    printf("Received: %s",buf);

    close(sockfd);
    return 0;
}
```

Example 3-1 Basic Client

The code on the previous page requires some explanation. The basic function of the client is as follows:

- Accept the hostname of a machine to connect to.
- Retrieve the details of the host (including the IP address)
- Create a socket
- Create an address consisting of the IP address of the user-supplied host, and the predefined port number.
- Connect the socket to the address
- Read from the socket
- Destroy the socket

Excercise 3-1

Create a file in your Unix home directory called *clnt.c* with the above C code. Try compiling the code.

Excercise 3-2

Try running the compiled program.

You should get a message something like "Connection refused". Why is this?

Excercise 3-3

Try changing the `PORT` definition to that of an existing service. Look in the `/etc/services` file on your Unix machine for a list of standard services. You will need to recompile your code every time you change the definition of `PORT`.

Excercise 3-4

Change the code so that it accepts the port as well as the hostname as a command line argument.

Solution 3-1

The compilation string should look something like the following:

```
$ cc clnt.c -o clnt -lnsl -lsocket
```

Most systems require you to explicitly to include the socket and nsl (network services) libraries.

If you are using the GNU C compiler, then substitute `gcc` for `cc` in the above.

Solution 3-2

If the program has compiled successfully then you will get a message something like "Connection refused". This is because the client is attempting to connect to a server at port 3780. It is likely that there is no server listening on this port, hence the message.

Solution 3-4

You need to make the following changes to the original code. The code should compile as before.

```
...
/* Connect to a user specified port on a user specified host */
#define MAXDATASIZE 100      /* max number of bytes we can get at once */
int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in remote_addr; /* connector's address information */
    unsigned short int port;

    if (argc != 3) {
        fprintf(stderr, "usage: clnt hostname port\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) { /* get the host info */
        perror("gethostbyname");
        exit(1);
    }

    port = htons(atoi(argv[2]));

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    remote_addr.sin_family = AF_INET; /* host byte order */
    remote_addr.sin_port = port; /* short, network byte order */
    remote_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(remote_addr.sin_zero), 8); /* zero rest of the struct */

    /* The rest of the code remains unchanged */
}
```

The Server

We now look at creating a basic server. In the code that follows, we create a socket that listens for connections. When a connection is received, it creates a new socket and send a message to the server over that socket. It then prints out anything that it receives and closes down the socket when the client terminates the connection.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

const int MYPOR = 3490; // The port users will be connecting to
const int BACKLOG = 5; // How many pending connections queue will hold

void main()
{
    int sockfd, new_fd; // listen on sock_fd, new connection on new_fd
    struct sockaddr_in my_addr; // my address information
    struct sockaddr_in remote_addr; // connector's address information
    int sin_size;
    char rbuf[1024];
    char sbuf[1024];
    int rval;

    sin_size = sizeof(struct sockaddr_in);
    strncpy (sbuf, "Welcome to Your New Server\r\n", sizeof(sbuf));

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET; // host byte order
    my_addr.sin_port = htons(MYPOR); // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
    memset(&(my_addr.sin_zero), 8, 0); // zero the rest of the struct

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
    {
        perror("bind");
        exit(1);
    }

    printf ("Socket listening on port %#d.\n", ntohs(my_addr.sin_port));

    // Start accepting connections
    if (listen(sockfd, BACKLOG) == -1)
    {
        perror("listen");
        exit(1);
    }
}
```

```

do {
    if ((new_fd = accept(sockfd, (struct sockaddr *)&remote_addr, &sin_size)) == -1)
    {
        perror("accept");
    }
    else
    {
        printf("Created new connection\n");
        if (rval = send(new_fd, sbuf, sizeof(sbuf), 0) == -1)
        {
            perror("Writing message stream");
        }

        do {
            memset(rbuf, 0, sizeof(rbuf));

            if ((rval = recv(new_fd, rbuf, sizeof(rbuf), 0)) == -1)
            {
                perror("Reading message stream");
            }

            if (rval != 0)
            {
                printf("--->%s\n", rbuf);
            }
        } while (rval != 0);

        printf("Ending connection.\n");

        // close the socket
        close(new_fd);
    }
} while(1);

close(sockfd);
exit(0);
}

```

Example 3-2 Basic Server that Accepts One Connection

Excercise 3-5

As with the client, create a file in your home directory called `server.c` containing the above code. It should compile with the same compiler options as the client.

Note that if there are multiple people running the same server on the same machine you will have to change the port on which your server listens so that it is unique.

Connect to your server using the client that you compiled earlier. Ensure that the client is connecting to the port on which your server is listening. You can simplify matters by using the modified client from excercise 3-4.

Excercise 3-6

The client in the previous example only receives data. Try connecting to your server using a more generic client, *telnet*. The arguments to *telnet* take the following form:

```
telnet hostname [portnumber]
```

Now you may type at the telnet session and see the output at the server.

Excercise 3-7

Try opening more than one telnet session to your server. What happens? Why is this? What solutions can you think of to this problem?

Excercise 3-8

Modify the client so that it reads from the socket first, and then prompts the use for input. Any input should be sent to the server.

Solution 3-7

When you make a second connection to the server, it will block, waiting for the first client to disconnect. As soon as the first client disconnects, the server creates a connection for the second client.

The above behaviour is not very useful for a real world server. There are a number of solutions to this.

- Create a new process for every socket that's created.
- Create a new thread for every socket that's created.
- Use non-blocking sockets

We will defer examination of these methods until chapter 5, Handling Concurrency.

Solution 3-8

The following code implements the requirements of exercise 3-8.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

/*
 * Connect to a predefined port on a user specified host
 */

const int PORT=3490;          // the port client will be connecting to
const int MAXDATASIZE=255;   // max number of bytes we can get at once

int main(int argc, char *argv[])
{
    int sockfd;
    int numbytes, rval;
    char rbuf[MAXDATASIZE];
    char sbuf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in remote_addr;    /* connector's address information */

    if (argc != 2) {
        fprintf(stderr, "usage: clnt hostname\n");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
    }
```

```

        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    remote_addr.sin_family = AF_INET;           // host byte order
    remote_addr.sin_port = htons(PORT);        // short, network byte order
    remote_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(remote_addr.sin_zero), 8,0);      // zero rest of the struct

    // Connect to address in remote_addr
    if (connect(sockfd, (struct sockaddr *)&remote_addr,\
                sizeof(struct sockaddr)) == -1)
    {
        perror("connect");
        exit(1);
    }

    if ((numbytes=recv(sockfd, rbuf, MAXDATASIZE, 0)) == -1) {
        perror("recv");
        exit(1);
    }

    // We must null terminate the buffer
    rbuf[numbytes] = '\0';
    printf("Received: %s",rbuf);

    while (1)
    {
        printf("-->");
        gets(sbuf);

        if (rval = send(sockfd, sbuf, sizeof(sbuf), 0) == -1)
        {
            perror("Writing message stream");
        }
    }

    shutdown(sockfd, 2);
    close(sockfd);

    return 0;
}

```

WinSock Programming

We now examine using the WinSock API. There are currently two widely used versions of WinSock. These are: WinSock 1.1 and WinSock2.

Windows 95 comes with WinSock 1.1 as standard while Windows98 and NT 4 come with WinSock2 by default. The main difference is that WinSock2 contains some additional MicroSoft proprietary extensions.

In this section we will assume the use of WinSock 1.1 unless otherwise stated since anything written with WinSock1.1 should work without modification with WinSock2.

Berkeley to WinSock: An Overview

Standard Berkeley socket calls will port to Windows Sockets with the following exceptions:

- You must call `WSAStartup()` to initialise the Windows Sockets DLL.
- You can use `read` and `write` to receive and send data but you must manually convert the socket descriptor to a file descriptor, using the `_open_ofshandle()` call. You can use `send()` and `recv()` in the standard manner.
- You must close a socket using the `closesocket()` call, instead of `close()`.
- You must call `WSACleanup()` at the end of your program in order to shut down the WinSock DLL.

Bearing these points in mind, we now look at implementing the code in examples 3-1 and 3-2.

Running the Client and Server on Windows

Notice that the following examples are hardcoded to check for WinSock 1.1.

In order to use WinSock, you must explicitly link the WinSock library. If you are using WinSock1.1 then use wsock32.lib. If you are using WinSock2, then use ws2_32.lib.

Details of how to link the library will vary depending on what development environment you use. When using MS DevStudio '97 you can add libraries by selecting Project, Settings. Click on the Link tab and select the Object/library modules entry. Add the required library to the existing list.

The following two code listings show examples 3-1 and 3-2 with the minimum changes required to make them work in Windows.

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <windows.h>
#include <winsock.h>

// Connect to a predefined port on a user specified host

const int PORT=3490;          /* the port client will be connecting to */
const int MAXDATASIZE=100;   /* max number of bytes we can get at once */

int main(int argc, char *argv[]){
    SOCKET sockfd;
    int numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in remote_addr;    /* connector's address information */
    WORD winsock_vers;
    WSADATA wsa_data;
    int err;

    if (argc != 2) {
        fprintf(stderr,"usage: clnt hostname\n");
        exit(1);
    }

    winsock_vers = MAKEWORD( 1, 1 );
    err = WSASStartup( winsock_vers, &wsa_data );

    if ( err ) {
        perror("Version Requested");
        WSACleanup();
        exit(1);
    }

    // Confirm that the WinSock DLL supports 1.1
    if ( LOBYTE( wsa_data.wVersion ) != 1 || HIBYTE( wsa_data.wVersion ) != 1 ) {

        // We couldn't find a usable WinSock DLL.
        perror("Winsock version");
        WSACleanup( );
        exit(1);
    }

    // The WinSock DLL is acceptable. Proceed.
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        WSACleanup();
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {
        perror("gethostbyname");
        WSACleanup();
        exit(1);
    }

    remote_addr.sin_family = AF_INET;          /* host byte order */
    remote_addr.sin_port = htons(PORT);       /* short, network byte order */
    remote_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(remote_addr.sin_zero), 8,0);     /* zero rest of the struct */

    // Connect to address in remote_addr
    if (connect(sockfd, (struct sockaddr *)&remote_addr, sizeof(struct sockaddr)) == -1)
    {
        perror("connect");
        WSACleanup();
        exit(1);
    }
}

```

```
if ((numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1)
{
    perror("recv");
    WSACleanup();
    exit(1);
}

buf[numbytes] = '\0';          /* We must null terminate the buffer */
printf("Received: %s",buf);

shutdown(sockfd, 2);
closesocket(sockfd);

WSACleanup();

return 0;
}
```

Example 3-3 Basic Client Modified for WinSock

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <windows.h>
#include <winsock.h>

const int MYPORT = 3490; // The port users will be connecting to
const int BACKLOG = 5; // How many pending connections queue will hold

void main()
{
    SOCKET sockfd, new_fd; // listen on sock_fd, new connection on new_fd
    struct sockaddr_in my_addr; // my address information
    struct sockaddr_in remote_addr; // connector's address information
    int sin_size;
    char rbuf[1024];
    char sbuf[1024];
    WORD winsock_vers;
    WSADATA wsa_data;
    int err;
    int rval;

    sin_size = sizeof(struct sockaddr_in);
    strncpy (sbuf, "Welcome to Your New Server\r\n", sizeof(sbuf));

    // Check for the WinSock 1.1 dll
    winsock_vers = MAKEWORD( 1, 1 );
    err = WSStartup( winsock_vers, &wsa_data );

    if ( err ) {
        perror("Version Requested");
        exit(1);
    }

    // Confirm that the WinSock DLL supports 1.1
    if ( LOBYTE( wsa_data.wVersion ) != 1 || HIBYTE( wsa_data.wVersion ) != 1 )
    {
        // No suitable WinSock DLL
        perror("Winsock version");
        WSACleanup( );
        exit(1);
    }

    // The WinSock DLL is acceptable. Proceed.
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        WSACleanup( );
        exit(1);
    }

    my_addr.sin_family = AF_INET; // host byte order
    my_addr.sin_port = htons(MYPORT); // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
    memset(&(my_addr.sin_zero), 8, 0); // zero the rest of the struct

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
    {
        perror("bind");
        WSACleanup( );
        exit(1);
    }

    printf ("Socket listening on port %#d.\n", ntohs(my_addr.sin_port));

    // Start accepting connections
    if (listen(sockfd, BACKLOG) == -1)
    {
        perror("listen");
        WSACleanup( );
        exit(1);
    }
}

```

```

do {
    if ((new_fd = accept(sockfd, (struct sockaddr *)&remote_addr, &sin_size)) == -1)
    {
        perror("accept");
    }
    else
    {
        printf("Created new connection\n");

        if (rval = send(new_fd, sbuf, sizeof(sbuf), 0) == -1)
        {
            perror("Writing message stream");
        }

        do {
            memset(rbuf, 0, sizeof(rbuf));

            if ((rval = recv(new_fd, rbuf, sizeof(rbuf), 0)) == -1)
            {
                perror("Reading message stream");
                break;
            }

            if (rval != 0)
            {
                printf("--->%s\n", rbuf);
            }
        } while (rval != 0);

        printf("Ending connection.\n");

        // Shutdown socket for both sending and receiving
        shutdown(new_fd, 2);
        closesocket(new_fd);
    }
} while(TRUE);

// Shutdown original socket for both sending and receiving
shutdown(sockfd, 2);
closesocket(sockfd);

WSACleanup();
exit(0);
}

```

Example 3-4 Basic Server modified for WinSock

Final Notes

Most system calls return -1 on failure. Always check this return code and use the perror function to print out the associated error message.

If you are debugging a server then it is likely that the server will be stopping abruptly. In this case the system may not release the port properly. Use setsockopt as follows:

```
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (void*)&on, sizeof(on))
```

This allows the server to reuse the port even though it has not been properly de-allocated.

4

Connectionless Sockets and Special Options

Connectionless Socket Programming

- When to use UDP
- Sending and Receiving Datagrams
- Broadcasting Datagrams
- Coping with Unreliability

Advanced Socket Mechanisms

- Setting Socket Options
- Working with Raw Sockets
- Signaling
- Sending Out of Band Data
- Delivering Data Efficiently using Multicasting

Connectionless Socket Programming

When to use UDP

UDP is connectionless, and hence it is more lightweight than TCP. If the probability of packet loss is low, then the overhead of TCP may not be justified. This may be the case if a highly reliable transmission medium is being used (for example FDDI).

The application must still provide for packet loss but if it doesn't occur very often then it is possibly advantageous for the program code to deal with this explicitly.

UDP may also be desirable for simple request/response interactions. In such cases, the traffic involved in setting up a TCP connection may outweigh the actual data to be transported. Also, it must be noted that if the communication is taking place over the loopback interface (that is, all communication is within the same host) then the data is guaranteed to be delivered.

Applications that use UDP as their transport provider include Trivial File Transfer Protocol (TFTP), BOOTP and Dynamic Host Configuration Protocol (DHCP). All of these protocols are designed for transferring small amounts of data in the form of a request/response mechanism.

Sending and Receiving Datagrams

To this point we have been dealing with connection oriented communication, using stream sockets. However, TCP/IP also provides us with a connectionless, packet-based interface using datagram sockets.

A datagram socket provides for symmetric data exchange, that is, there need not be a client/server relationship between the communicating processes. Instead, each message includes the destination address.

Datagram sockets are created in much the same way as stream sockets. A typical call is shown below:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

If a particular local address is needed, the `bind()` operation must precede the first data transmission. Otherwise, the system will set the local address and/or port when data is first sent.

To send data, the `sendto()` primitive is used,

```
sendto(s, buf, buflen, flags, (struct sockaddr *)&to, tolen);
```

The `s`, `buf`, `buflen`, and `flags` parameters are used as before. The `to` and `tolen` values indicate the address of the recipient.

When using an unreliable datagram interface, it is unlikely that any errors will be reported to the sender. When it is possible to detect delivery errors (for instance when a network is unreachable), the `sendto()` call will return `-1` and the global value `errno` will be set.

To receive messages on an unconnected datagram socket, the `recvfrom()` primitive is provided:

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *)&from, &fromlen);
```

As before, the `fromlen` parameter initially contains the size of the from buffer, and on return, indicates the actual size of the address from which the datagram was received.

In addition to `sendto()/recvfrom()`, datagram sockets may also call `connect()` to associate a socket with a specific destination address. In this case, any data sent on the socket will automatically be addressed to the connected peer, and only data received from that peer will be delivered to the user. This means that you can use `send()` and `recv` instead of `sendto()` and `recvfrom()`.

A socket may only be connected to a single address. Subsequent calls to `connect` will over write the previous setting. A `connect` to the null address (family `AF_UNSPEC`) will cause a disconnection.

`Connect` requests on datagram sockets return immediately. This is different from stream sockets where a `connect` request initiates establishment of an end to end connection. This is because all that the `connect` does is record the peers address. `accept()` and `listen()` are not used with datagram sockets.

While a datagram socket is connected, errors from recent `send` calls may be returned asynchronously. These errors may be reported on subsequent operations on the socket. A special socket option, `SO_ERROR`, may be used with the `getsockopt()` call to interrogate the error status. An example connectionless server is shown in figure 4-1 below.

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

const int MYPOR=4350 // the port users will be sending to
const int MAXBUFL=255 // Maximum amount we can receive

int main(){
    int sockfd;
    struct sockaddr_in local_addr;
    struct sockaddr_in remote_addr;
    int addr_len, numbytes;
    char buf[MAXBUFL];

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    local_addr.sin_family = AF_INET; // host byte order
    local_addr.sin_port = htons(MYPOR); // short, network byte order
    local_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
    memset(&(local_addr.sin_zero), 8, 0); // zero the rest of the struct

    if (bind(sockfd, (struct sockaddr *)&local_addr, sizeof(struct sockaddr)) == -1) {
        perror("bind");
        exit(1);
    }

    addr_len = sizeof(struct sockaddr);
    if ((numbytes=recvfrom(sockfd, buf, MAXBUFL, 0, (struct sockaddr *)&remote_addr,
    &addr_len)) == -1) {
        perror("recvfrom");
        exit(1);
    }

    printf("Received packet of length: %d from %s\n", numbytes,
    inet_ntoa(remote_addr.sin_addr));

    buf[numbytes] = '\0'; // Null terminate the buffer
    printf("Packet contains: \"%s\"\n", buf);
    close(sockfd);
}

```

Figure 4-1 Basic Connectionless Server

The client is equally straightforward and is shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>

const int MYPORT=4350    // the port users will be sending to

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in remote_addr; // connector's address information
    struct hostent *he;
    int numbytes;

    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        exit(1);
    }

    // get the remote host info
    if ((he=gethostbyname(argv[1])) == NULL) {
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    remote_addr.sin_family = AF_INET;           // host byte order
    remote_addr.sin_port = htons(MYPORT);      // short, network byte order
    remote_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(remote_addr.sin_zero), 8);        // zero the rest of the struct

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0, \
        (struct sockaddr *)&remote_addr, sizeof(struct sockaddr))) == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to %s\n",numbytes,inet_ntoa(remote_addr.sin_addr));

    close(sockfd);
    exit(0);
}
```

Figure 4-2 Basic Connectionless Client

Broadcasting Datagrams

Using datagram sockets, it is possible to send broadcast packets, that is, send a packet to every station on the local network. Note that the network itself must be capable of broadcasting, the system provides no simulation of broadcast in software.

Broadcast messages can place a high load on a network since they force every host on the network to service them. Consequently, the ability to send broadcast packets has been limited to sockets which are explicitly marked as allowing broadcasting. Broadcast is typically used for one of two reasons:

- To find a resource on a local network without prior knowledge of its address
- Important functions such as routing require that information be sent to all accessible neighbours.

To send a broadcast message, a datagram socket should be created:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

The socket is marked as allowing broadcasting,

```
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));
```

Next, you have to decide what kind of broadcast you want to use. The simplest type is a packet sent to IP address 255.255.255.255.

Simple broadcasts have a number of problems, however, so to make sure your broadcasts work properly, you should do a "directed broadcast" instead.

To construct a directed broadcast address, you need to know the LAN's network address and the netmask for that LAN. Mathematically, the directed broadcast address is the network address bitwise OR'd with the one's complement of the netmask. We work out the network address below and use it to find the directed broadcast address.

```
u_long host_addr = 0xC0A8724C;           // 192.168.114.76
u_long net_mask = 0xFFFFE000;          // 255.255.224.0
u_long net_addr = host_addr & net_mask; // 192.168.96.0
u_long dir_bcast_addr = net_addr | (~net_mask); // 192.168.127.255
```

Finally, a port number should be bound to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof (sin));
```

On Unix systems, the Internet domain supports a shorthand notation for broadcast on the local network. The macro `INADDR_BROADCAST`, usually defined in `<netinet/in.h>`, is used for this purpose.

On most systems, `INADDR_BROADCAST` is set to `255.255.255.255`. If this is the case then you may want to manually work out the directed broadcast address as above.

Excercise 4-1

The code shown in figure 4-2 is a basic connectionless client. Modify this client so that it sends packets to the directed broadcast address.

Solution 4-1

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>

const int MYPOR=4350 // the port users will be sending to
const ulong NET_MASK = 0xFFFFE000; // 255.255.224.0, change this to suit your own
netmask

int main(int argc, char *argv[]){
    int sockfd;
    struct sockaddr_in bcast_addr;
    struct hostent *he;
    int numbytes;
    int on = 1;
    u_long host_addr;
    u_long net_addr;
    u_long dir_bcast_addr;
    char bcast_msg = "Broadcast!\n";

    // get the local IP address
    if ((he=gethostbyname("localhost")) == NULL) {
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    // Mark the socket as allowing broadcasts
    setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof (on));

    // Work out the broadcast address
    host_addr = he->h_addr;
    net_addr = host_addr & NET_MASK;
    dir_bcast_addr = net_addr | (~NET_MASK);

    bcast_addr.sin_family = AF_INET; // host byte order
    bcast_addr.sin_port = htons(MYPOR); // short, network byte order
    bcast_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    bzero(&(bcast_addr.sin_zero), 8);
    bind(sockfd, (struct sockaddr *) &bcast_addr, sizeof(bcast_addr));

    // Send the broadcast
    if ((numbytes=sendto(sockfd, bcast_msg, strlen(bcast_msg), 0, \
        (struct sockaddr *)&bcast_addr, sizeof(struct sockaddr))) == -1) {
        perror("sendto");
        exit(1);
    }

    close(sockfd);
}
```

Coping with Unreliability

If you decide to use a connectionless transport in your application then you have to decide what to do about the unreliability of that transport. If the transmission medium is sufficiently good then you might decide to ignore the unreliability completely. In general, though, this is not the case. We examine some of the alternatives below.

The most common scheme is where the sender sends a packet and waits for an acknowledgement. This waiting should expire after a timeout period. If the waiting times out then the sender should re-transmit the packet.

The response could be the actual packet that was sent but it is more efficient to just respond with a message that identifies the packet that was received. This might be some sort of checksum or hash of the sent data. (Note that UDP packets do not contain sequence numbers like TCP segments do). Alternatively, if the incoming packet is a request for information then the response may serve as an acknowledgment.

This scheme is obviously not foolproof. The packet may have been delivered without any problems while the response may be lost. You may consider requiring an acknowledgement-acknowledgment from the original sender however this buys us nothing and increases the complexity greatly.

Assuming that we cannot be sure that an acknowledgement will be delivered, we have to provide for duplicate packets at the receiving end. Duplicate packets may not be a problem if they do not change the state of the server. For example, if the packets are requests for information, then multiple requests will cause multiple responses to be sent. This is not a problem for the server and it may be more efficient to just allow duplicates than to check and filter them out.

If the packet is a request for an operation to be carried out (for example, increase a bank balance by £100) then multiple requests may be harmful. In this case, the server may have to filter out duplicates. One workaround is to make sure that the operation is idempotent, that is, executing it multiple times will have the same effect as executing it once (for example, set a bank balance to be £1,500).

Advanced Socket Mechanisms

Socket Options

It is possible to set and get a number of options on sockets via the `setsockopt()` and `getsockopt()` system calls. These options include such things as marking a socket for broadcasting, not to route, to linger on close, etc. The general forms of the calls are:

```
setsockopt(sockfd, level, optname, optval, optlen);  
and  
getsockopt(sockfd, level, optname, optval, optlen);
```

The parameters to the calls are as follows:

<code>sockfd</code>	The socket on which the option is to be applied.
<code>level</code>	The protocol layer on which the option is to be applied; in most cases this is the <i>socket level</i> , indicated by the symbolic constant <code>SOL_SOCKET</code> .
<code>optname:</code>	This is the actual option, and is a symbolic constant. These are listed below.
<code>optval:</code>	This points to the value of the option.
<code>optlen:</code>	This is the length of the value of the option.

For `getsockopt()`, `optlen` is a value-result parameter, initially set to the size of the storage area pointed to by `optval`, and modified upon return to indicate the actual amount of storage used.

The example below shows one use for `getsockopt()`, determining the type of a socket.

```
#include <sys/types.h>  
#include <sys/socket.h>  
  
int type, size;  
int sockfd;  
  
size = sizeof (int);  
  
if (getsockopt(sockfd, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0)  
{  
    ...  
}
```

In the above code, `type` will be assigned a value representing the socket type. For example, if the socket is a datagram socket then `type` will have the value corresponding to `SOCK_DGRAM`.

In the example above, we called `getsockopt` at the *socket level*. Some common socket options available at the socket level are listed in table 4-1 below:

<i>Option</i>	<i>Return type</i>	<i>Description</i>
SO_DEBUG	int/BOOL	Debugging is enabled.
SO_REUSEADDR	int/BOOL	The socket can be bound to an address which is already in use.
SO_KEEPALIVE	int/BOOL	Keepalives are being sent.
SO_DONTROUTE	int/BOOL	Routing is disabled.
SO_LINGER	struct linger/LINGER	Returns the current linger options.
SO_BROADCAST	int/BOOL	Socket is configured for the transmission of broadcast messages.
SO_OOBINLINE	int/BOOL	Out-of-band data is being received in the normal data stream.
SO_SNDBUF	int	Buffer size for sends
SO_RCVBUF	int	Buffer size for receives
SO_ERROR	int	Retrieve error status and clear.
SO_ACCEPTCONN	int/BOOL	Socket is listening.
SO_TYPE	int	The type of the socket (for example, SOCK_STREAM).

Table 4-1 Socket level Options common to Unix and Win32

Where two return types are shown, the former is the Unix return type, the latter is the Win32 return type.

There are likely to be many other options available on your system. Consult your documentation for details.

Working with Raw Sockets

A raw socket provides users access to the underlying communication protocols which support socket abstractions. These sockets are normally datagram oriented, though their exact characteristics are dependent on the interface provided by the protocol. In the case of TCP/IP, IP provides the delivery.

Raw sockets are not intended for the general user; they have been provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol.

On a Unix system to require super-user privileges to create a raw socket. Creation of a raw socket is shown below:

```
s = socket(AF_INET, SOCK_RAW, 0);
```

The `send()` and `recv()` calls can be used to transmit data over the connection. It is the responsibility of the programmer to construct the appropriate header and footer information depending on the protocol being implemented. TCP/IP packets can be built up using the structures defined in the system header files, for example, the `tcphdr` structure defined in the Linux kernel.

Signals

Signals are the software equivalent of hardware interrupts. Signals can be sent from one process to another to indicate the occurrence of a particular event. There are a lot of different signals that can be sent although the specific set varies from system to system.

Signals are typically used to reflect user interrupts or else system events. Some commonly used signals in the Unix environment are listed in table 4-2 while table 4-3 lists the signals available in the Win32 environment.

How a process reacts when it receives a signal is determined by that signal's disposition. The system provides a default disposition for signals. This is normally either *terminate* or *ignore*. However, a program can change the disposition of a signal towards it by providing an *interrupt handler*.

Signaling in the Unix Environment

There are a number of different methods of signal handling in Unix. The original signal interface consisted of the `signal()` system call. The `signal` system call had a number of problems and is now obsolete. The `sigset()` system call has the same syntax and fixes many of the problems with `signal()`.

The prototype of the `sigset()` system call is shown below:

```
void ( sigset( int sig, void (disp)(int)))(int);
```

The arguments are, the signal number (one of the values listed in table 4-2), and the address of a function that takes an integer argument and has no return value. Alternatively, the second argument may be `SIG_IGN` (ignore the signal) or `SIG_DFL` (use the system default).

A typical usage of signals is shown below:

```
int main()
{
    void handle_sig(int);

    /* Program will respond to the interrupt key
     * by executing the handle_sig() function.
     */
    sigset(SIGINT, handle_sig);
}

void handle_sig(int sig)
{
    printf("Received Interrupt\n");
}
```

A program may explicitly raise a signal using the kill system call. The kill system call has the following form:

```
kill (process_id, SIGNAL);
```

Note that the POSIX P1003.1 standard describes a more flexible and portable signal handling interface that is available on most versions of Unix. See the `sigaction()` man pages for more details. We described the `sigset()` method here for simplicity.

Signal	Description
SIGHUP	Controlling terminal hangup
SIGINT	Interrupt the process
SIGQUIT	Quit execution
SIGKILL	Kill the process (cannot be caught or ignored)
SIGPIPE	Write on a pipe with no reader
SIGALRM	Alarm clock has gone off
SIGTERM	Software termination signal
SIGUSR1	Application defined signal
SIGUSR2	Application defined signal
SIGCHLD	Child status change
SIGPOLL	Pollable event occurred

Table 4-2 Commonly Used Signals in Unix

Signaling in the Win32 environment

The Win32 environment provides a signaling interface similar to that in Unix. The `signal()` function call provides similar functionality to the Unix `signal()` system call. The prototype is shown below:

```
void (*signal(int sig, void(__cdecl *func)(int sig [,int subcode])))  
(int sig);
```

The usage is the same as with `sigset()` as previously described. Note that every time the signal handler is executed, the value of `func` is reset to `SIG_DFL`. This behaviour causes many problems and is one of the reasons why the `signal()` system call is obsolete in Unix.

The signals available are listed in table 4-3. Signals can be raised using the `raise()` function call.

Win32 signaling is described here mainly for completeness, it has little use in Win32 network programming.

Signal	Description
SIGABRT	Abnormal termination
SIGFPE	Floating-point error
SIGILL	Illegal instruction
SIGINT	CTRL+C signal
SIGSEGV	Illegal storage access
SIGTERM	Termination request

Table 4-3 Available signals in Win32

Sending Out of Band Data

It is sometimes necessary to send high priority data over a connection that may have unread low priority data at the other end. The stream socket abstraction includes the notion of *out of band* (OOB) data to support this. Out of band data is a logically independent transmission channel associated with each pair of connected stream sockets. Out of band data is delivered to the user independently of normal data.

For example, a user interface process may be interpreting commands and sending them on to another process through a stream connection. The user interface may have filled the stream with as yet unprocessed requests when the user types a command to cancel all outstanding requests. Rather than have the high priority data wait to be processed after the low priority data, it is possible to send it as out-of-band data.

Out of Band Data on Unix Systems

A program can send out of band data simply by adding the `MSG_OOB` flag to the `send()` command as below:

```
send (fd, buf, sizeof(buf), MSG_OOB);
```

The recipient must take a number of steps before it can successfully handle out of band data. Firstly, it is important to note that when out of band data is received, a signal (`SIGURG`) is generated.

We must tell the system that whenever events associated with the socket generate signals, those signals should be sent to us. We do this by using `fcntl` with the `F_SETOWN` command.

```
fcntl(sockfd, F_SETOWN, getpid())
```

Now, whenever out of band data is received, a `SIGURG` signal is sent to our process. If we want to handle out of band data in a special way (which is likely) then we must install a signal handler. Assuming that we have defined a function called `handle_oob()`, we install the signal handler as below:

```
sigset(SIGURG, handle_oob);
```

Figure 4-2 below shows a server that handles out of band data by means of a signal handler as described above. The server is based on the single connection server in Chapter 3. Modifying it to handle multiple connections is trivial.

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/file.h>

const int MYPOR = 3490; // The port users will be connecting to
const int BACKLOG = 5; // How many pending connections held
const int MAXDATASIZE=255; // max number of bytes we can get at once
int new_fd;

void oob_handler(int sig)
{
    int rval;
    char rbuf[MAXDATASIZE];

    printf("Received OOB data:");
    if ((rval = recv(new_fd, rbuf, sizeof(rbuf), 0)) == -1)
    {
        perror("Reading message stream");
    }
    printf("%s\n", rbuf);
    fflush(stdout);
}

void main()
{
    int sockfd; // listen on sock_fd
    struct sockaddr_in my_addr; // my address information
    struct sockaddr_in remote_addr; // connector's address information
    int sin_size;
    char rbuf[MAXDATASIZE];
    char sbuf[MAXDATASIZE];
    int rval;
    int on = 1;

    // Install signal handler
    sigset(SIGURG, oob_handler);

    strncpy (sbuf, "Welcome to Your New Server\r\n", sizeof(sbuf));

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }
}

```

```

    my_addr.sin_family = AF_INET;           // host byte order
    my_addr.sin_port = htons(MYPORT);      // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY;  // auto-fill with my IP
    memset(&(my_addr.sin_zero), 8, 0);     // zero the rest of the struct

// Set REUSEADDR, this is useful when developing a server
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (void*)&on, sizeof(on));

if (bind(sockfd, (struct sockaddr *)&my_addr, \
        sizeof(struct sockaddr))
    == -1)
    {
        perror("bind");
        exit(1);
    }

    sin_size = sizeof(struct sockaddr_in);

-1) if (getsockname(sockfd, (struct sockaddr *) &my_addr, &sin_size) ==
    {
        perror("Getting socket name");
        exit(1);
    }

printf ("Socket listening on port %#d.\n", ntohs(my_addr.sin_port));

// Start accepting connections
if (listen(sockfd, BACKLOG) == -1)
    {
        perror("listen");
        exit(1);
    }

```

```

do
{
    if ((new_fd = accept(sockfd, (struct sockaddr *)&remote_addr, \
                        &sin_size)) == -1)
    {
        perror("accept");
    }
    else
    {
        printf("Created new connection\n");

        // Set the process receiving SIGURG signals to be us
        if (fcntl(new_fd, F_SETOWN, getpid()) < 0) {
            perror("fcntl F_SETOWN");
            exit(1);
        }

        if (rval = send(new_fd, sbuf, sizeof(sbuf), 0) == -1)
        {
            perror("Writing message stream");
        }

        do
        {
            memset(rbuf, 0, sizeof(rbuf));

            if ((rval = recv(new_fd, rbuf, sizeof(rbuf), 0)) == -1)
            {
                perror("Reading message stream");
            }

            if (rval != 0)
            {
                printf("---->%s\n", rbuf);
            }
        } while (rval != 0);

        printf("Ending connection.\n");

        // close the socket
        close(new_fd);
    }
} while(1);

exit(0);
}

```

Figure 4-2 Server that handles Out of Band Data using a Signal Handler

The server described above will handle out of band data but, at the moment we have no particular way of sending out of band data. The example below is a basic client that connects on a fixed port.

While running, the client prompts the user for input. Any input is sent to the server and printed to the terminal associated with the server.

Any line of data that begins with OOB is considered to be out of band data by the client. The client sends this data (without the OOB characters) to the server marked as out of band. The OOB signal handler in the server prints out this data.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

/*
 * Connect to a predefined port on a user specified host
 */

const int PORT=3490;          // the port client will be connecting to
const int MAXDATASIZE=255;   // max number of bytes we can get at once

int main(int argc, char *argv[])
{
    int sockfd;
    int numbytes, rval;
    char rbuf[MAXDATASIZE];
    char sbuf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in remote_addr; // connector's address information

    if (argc != 2) {
        fprintf(stderr, "usage: clnt hostname\n");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
}
```

```

remote_addr.sin_family = AF_INET;        // host byte order
remote_addr.sin_port = htons(PORT);     // short, network byte order
remote_addr.sin_addr = *((struct in_addr *)he->h_addr);
memset(&(remote_addr.sin_zero), 8,0); // zero rest of the struct

// Connect to address in remote_addr
if (connect(sockfd, (struct sockaddr *)&remote_addr,\
             sizeof(struct sockaddr)) == -1) {
    perror("connect");
    exit(1);
}

if ((numbytes=recv(sockfd, rbuf, MAXDATASIZE, 0)) == -1) {
    perror("recv");
    exit(1);
}

rbuf[numbytes] = '\0'; // We must null terminate the rbuffer
printf("Received: %s",rbuf);

while (1)
{
    printf("-->");
    gets(sbuf);

    if (strncmp(sbuf, "OOB", 3) == 0)
    {
        // Strip off first three characters before we send
        if (rval = send(sockfd, (sbuf + 3), (sizeof(sbuf) -3), MSG_OOB)
== -1)
        {
            perror("Writing message stream");
        }
    } else {
        if (rval = send(sockfd, sbuf, sizeof(sbuf), 0) == -1)
        {
            perror("Writing message stream");
        }
    }
}

shutdown(sockfd, 2);
close(sockfd);

return 0;
}

```

Figure 4-3 Client that Send Data Prefixed with OOB as Out of Band

Excercise 4-2

The client in figure 4-3 only quits when the user types Ctrl-C. This causes the program to abort and relies on the system to clean up. Add a signal handler to the code that traps this signal (SIGINT) and performs cleanup before quitting.

Solution 4-2

You must add a signal handler similar to the one described below:

```
void handle_sigint(int sig)
{
    printf("Closing down.\n");
    shutdown(sockfd, 2);
    close(sockfd);
}
```

In the main function, you must install a signal handler as follows:

```
int main()
{
    void handle_sigint(int);

    /* Program will respond to the interrupt key
     * by executing the handle_sigint() function.
     */
    sigset(SIGINT, handle_sigint);
}
```

Note also, that sockfd must be declared outside the main function so that it is accessible to the signal handler.

Delivering Data Efficiently Using Multicasting

IP multicasting provides a mechanism for sending a single datagram to a group of systems. Normally, only systems that have joined the multicast group process the datagrams.

Multicast datagrams are transmitted and delivered with the same "best effort" reliability as regular unicast IP datagrams. That is, the datagrams are not guaranteed to arrive intact at all members of the destination group or in the same order as the datagrams were sent.

Membership in a multicast group is dynamic. That is, systems can join and leave groups at any time. A system remains a member of a multicast group until the last socket that joined the group is closed or has dropped membership in the group. A system can be a member of more than one group at a time.

Joining a Multicast Group

In order to receive multicast datagrams, a system must join the multicast group. An application can request that the system join a group by using the `IP_ADD_MEMBERSHIP` socket option. For example:

```
#include <netinet/in.h>

struct ip_mreq mreq;
mreq.imr_multiaddr.s_addr = inet_addr("224.1.2.3");
mreq.imr_interface.s_addr = INADDR_ANY;
setsockopt(s, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

The `ip_mreq` structure is also defined in `netinet/in.h`. It has the following definition:

```
struct ip_mreq {
    struct in_addr  imr_multiaddr; // multicast address of group
    struct in_addr  imr_interface; // local IP address
};
```

If `imr_interface` is `INADDR_ANY`, the membership is joined on the system default interface. Otherwise, `imr_interface` should be the IP address of a local interface.

Note that in Win32, the `ip_mreq` structure and the `IP_*` options are defined in `winsock.h`.

Leaving a Multicast Group

An application automatically leaves a multicast group when it terminates, either normally or abnormally. Alternatively, the application can leave a group by using the `IP_DROP_MEMBERSHIP` socket option. For example:

```
#include <netinet/in.h>

struct ip_mreq mreq;
mreq.imr_multiaddr.s_addr = net_addr("224.1.2.3");
mreq.imr_interface.s_addr = INADDR_ANY;
setsockopt(s, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

Note that `imr_interface` must match the field that was used when the `IP_ADD_MEMBERSHIP` socket option was specified for `imr_multiaddr`.

Sending IP Multicast Datagrams

To send a multicast datagram, the application specifies an IP multicast address as the destination address in a `sendto()` call. For example:

```
#include <netinet/in.h>

struct sockaddr_in servaddr;
servaddr.sin_family = AF_INET;
servaddr.sin_port = 12345;
servaddr.sin_addr.s_addr = inet_addr("224.1.2.3");
sendto(s, buf, buflen, 0, &servaddr, sizeof(servaddr));
```

It is not necessary for the system to join a multicast group in order to send multicast datagrams.

Each multicast datagram is sent from one network interface at a time, even if the system has joined the destination multicast group on more than one interface. The system administrator configures a default interface from which to send multicast datagrams. An application can override the default by setting a socket option `IP_MULTICAST_IF` to specify the outbound interface for datagrams sent through that socket.

Normally, multicast datagrams are sent only to systems directly connected to the same network that the sending interface is on. If multicast datagrams are intended for distant networks, a special multicast router must be present on the local and intermediate networks. A socket option `IP_MULTICAST_TTL` controls the number of intermediate systems through which a multicast datagram can be forwarded (TTL stands for time to live, that is, the number of gateways that the packet may pass through).

The TTL may be set as below:

```
#include <netinet/in.h>

unsigned char ttl = 64;
setsockopt(s, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

If a multicast datagram is sent to a port from which an application on the local system is reading, normally a copy of the datagram is looped back and delivered to the application. A socket option `IP_MULTICAST_LOOP` allows the sending application to disable loopback for datagrams sent through that socket.

Receiving IP Multicast Datagrams

In order to receive multicast datagrams, an application must bind to the port number to which multicast datagrams will be sent. Additionally, the system must join the multicast group on the interface on which the multicast datagrams arrive.

An application automatically leaves a multicast group when it terminates, either normally or abnormally. Alternatively, the application can leave a group by using the `IP_DROP_MEMBERSHIP` socket option. However, the system remains a member of a multicast group as long as at least one application is a member.

If more than one application binds to the same port number on a system, each application must set the `SO_REUSEPORT` socket option before binding to the port. In that case, every application will receive all multicast datagrams sent to the port number.

5

Handling Concurrency

Process and Thread Concepts

- Managing UNIX Processes, fork() and exec()
- Creating Threads in Win32
- Creating Threads in UNIX
- Thread Synchronisation

Concurrency in the Server

- Iterative and Concurrent Servers
- Multiprocessing vs. Multithreaded Servers
- An Example Multiprocessing and Multithreading Server

Asynchronous and Non-blocking Sockets

- Asynchronous Sockets
- Integrating Socket I/O with Windows Messaging: WSAAsyncSelect()
- Applying Overlapped I/O to Sockets
- Waiting for I/O with select()

Process and Thread Concepts

Managing UNIX processes, fork() and exec()

On Unix systems, a process can create a copy of itself, using the `fork()` system call. The created process is referred to as a *child process*. The process that calls `fork()` is referred to as a *parent process*.

In order to execute different tasks in a child process, a process must first call `fork()` and the child process must call `exec()`. `exec()` overlays the existing code segment with a new code segment. This code is then executed.

Similar functionality on Win32 systems is provided using the `_spawn()` and `_exec()` functions.

The code extract in figure 5-1 illustrates typical usage of the `fork()` system call on Unix. Note that `fork` returns different values to the parent and child process. The value returned to the parent is the process id of the child. The value returned to the child is always 0. In the case of an error, -1 is returned.

```
pid_t    pid;

if ((pid = fork()) < 0)
{
    perror("Fork failed");
    exit(1);
}

if (pid == 0)
{
    // This is the child process
    execl("childtask", NULL);
    fatal("Exec failed.");
}

// This is the parent
if (waitpid(pid, NULL, 0) < 0)
    fatal("waitpid failed");
}
```

Figure 5-1 Typical Usage of fork() and exec()

Creating threads in Win32

Windows provides us with a comparatively simple mechanism for creating threaded applications. The general approach is summarised in pseudo-code in figure 5-2 below:

```
function f1()
function f2()

main() {
    spin off f1()
    spin off f2()

    do_main_task()
}
```

Figure 5-2 General Approach to Multithreading Applications

In order to spin off a function, the Win32 environment provides us with two function calls that we can make. These are `CreateThread()`, and `_beginthread()`. Both of these functions require a handle to a function to be passed among their arguments.

`CreateThread()` works as long as your program makes nothing but Win32 function calls. If your program make C runtime library calls then you must use `_beginthread()`. Since there is no particular advantage to using `CreateThread()` we will look only at `_beginthread()`.

`_beginthread()` has the following prototype:

```
unsigned long _beginthread(
    void (*lpThreadEntryPoint)(void *lpArgList),
    unsigned uStackSize,
    void *lpArgList);
```

`lpThreadEntryPoint` is the address of the function which will be executed as a thread (this is called the thread entry point). The entry point should be a function with the following prototype:

```
void ThreadedFunction (void*)
```

The second argument is the number of bytes that will be allocated to the stack of the thread being started. Passing this argument as zero causes the thread to have the same stack size as the calling thread.

The third argument is the argument list to be passed to the thread entry point. This may be `NULL` if no parameters are required. If a single parameter is required then it can be cast as `void*`. If multiple parameters are required then they can be passed in a structure which can be cast to `void*`.

The return value from `_beginthread()` is the thread handle. This can be used with other thread manipulation functions such as `SuspendThread()`, `ResumeThread()` and `SetThreadPriority()`.

The `_endthread()` function is provided to allow threads to terminate themselves. This function is not strictly necessary since a thread will be properly terminated by the system when it returns.

We now look at an example of a threaded program in Win32. The following example defines a function which takes a number as its input, and prints it out 5 times, at intervals of 1 second. The main function calls this function inside a thread, with values 9 to 0. It waits for 1½ seconds before it starts each new thread.

```

#include <windows.h>
#include <process.h>
#include <stddef.h>
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>

// Print the input number out five times at 1-second intervals
void printout(void* i){
    int j = (int)i;    // Cast required for MS VC++ 5.x
    int count = 5;

    while (count-->0)
    {
        // Print out message and sleep for 1 second
        printf("Output from printout called with %d\n", j);
        Sleep(1000);
    }
}

void main() {
    int index=10;
    while (index-->0)
    {
        // Start the new thread
        _beginthread(printout, 0, (void*)index);
        Sleep(1500);
    }

    // Sleep for 10 seconds
    Sleep(10000);
}

```

Figure 5-3 Basic Win32 Threaded Application

When a thread exits, the memory allocated to its stack is released and any DLLs that the process is using are notified of the threads disappearance. Unfortunately, there is no clean way for one thread to shut down another. Win32 provides the `TerminateThread()` function but, this is an abortive action. The stack is not de-allocated and the DLLs are not notified.

Creating threads in UNIX

There are a number of threading interfaces available on Unix. The most common interface is the POSIX Threads interface, more commonly called Pthreads. Pthreads is available on most Unix platforms and this is the interface that we examine here.

Pthreads provides us with an interface similar to the Win32 interface described above. The `pthread_create()` call is the pthreads analog to the Win32 `_beginthread()` call. The prototype is

```
pthread_create(pthread_t* thread_id,
              pthread_attr_t* tattr,
              (*thread_entry_point)(void *args),
              void* args)
```

The code in figure 5-4 below shows the basic threaded application as described in figure 5-3 with modifications to allow it to use Pthreads.

```
#include <stdio.h>
#include <pthread.h>

void printout(void* i){
    int j = (int)i;
    int count = 5;

    while (count-->0)
    {
        // Print out message and sleep for 1 second
        printf("Output from printout called with %d\n", j);
        sleep(1);
    }
}

void main(){
    int index=10;
    pthread_t tid;
    int ret, status;

    while (index-->0)
    {
        if (ret = pthread_create(&tid, NULL, printout, (void*)index))
            perror("thread_create");

        sleep(1);
    }

    // sleep for 10 seconds just to make sure that
    // all the threads have finished before we exit
    sleep(10);
}
```

Figure 5-4 Basic Threaded Application using Pthreads

Thread Synchronisation

Thread synchronisation methods are outlined in general in Chapter 1 in the section on concurrency. We now look at how we actually implement these synchronisation schemes using the functions provided by Pthreads and by the Win32 threading library.

Mutual Exclusion

We use mutexes to enforce mutual exclusion whenever threads require access to global resources. As an example, consider the basic threaded application described in figures 5-3 and 5-4. When these are run, we should see that the output from all of the different threads is interleaved on the screen.

Suppose we want to change this behaviour so that when a thread runs, it has exclusive access to the screen. Each thread must wait until it can grab exclusive access. When a thread finishes, it should indicate that it no longer requires the screen.

We can accomplish this behaviour using a mutex. Note that the mutex is not inherently bound to a resource. The programmer just decides on a name for a mutex that reflects its use and then uses it accordingly.

In Win32, a mutex can be created using the `CreateMutex()` function. This has the following prototype:

```
HANDLE CreateMutex
(
    LPSECURITY_ATTRIBUTES MutexAttributes,    // attributes
    BOOL InitialOwner,                       // initial ownership
    LPCTSTR Name                             // mutex-object name
);
```

The arguments have the following meanings:

MutexAttributes: These are the security attributes of created thread. Generally this can be set to `NULL`.

InitialOwner: If this is set to `TRUE`, the calling thread requests ownership of the mutex. Otherwise, the mutex is unowned.

Name: A null-terminated string specifying the name of the mutex.

Once a mutex has been created, a thread can wait for it using `WaitForSingleObject()`. When a thread is finished with the resource, it calls `ReleaseMutex()`.

In the following example, we use the mutex `myConsoleMutex` to protect access to the console.

```
#include <windows.h>
#include <process.h>
#include <stddef.h>
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>

HANDLE myConsoleMutex;

void printout(void* i)
{
    int j = (int)i;
    int count = 5;

    WaitForSingleObject( myConsoleMutex, INFINITE );
    while (count-->0)
    {
        // Print out message and sleep for 5 seconds
        printf("Output from printout called with %d.\n", j);
        Sleep(1000);
    }
    ReleaseMutex( myConsoleMutex );
}

void main()
{
    myConsoleMutex=CreateMutex (NULL, FALSE, NULL);
    int index=10;

    while (index-->0)
    {
        _beginthread(printout, 0, (void*)index);
        Sleep(1500);
    }

    // Sleep for 40 seconds just to make sure that
    // all the threads have finished before we exit
    Sleep(40000);
}
```

Figure 5-5 Protecting Access to the Console with a Mutex in Win32

In the Pthreads system, a mutex can be created using the `pthread_mutex_init()` function. The prototype is shown below:

```
int pthread_mutex_init(pthread_mutex_t *mp,  
                       const pthread_mutexattr_t *mattr);
```

The `mattr` variable corresponds to a set of attributes that will be applied to the mutex to be initialised. This may be `NULL` if only the default attributes are required. A thread can wait on a mutex using the `pthread_mutex_lock()` function. The mutex is unlocked using the `pthread_mutex_unlock()` function. The prototypes for these functions are shown below:

```
int pthread_mutex_lock(pthread_mutex_t *mutex)  
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

We now look at applying mutual exclusion to the Pthreads code in figure 5-4. These changes are shown in figure 5-6 below.

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t myConsoleMutex;

void printout(void* i) {
    int j = (int)i;
    int count = 5;

    pthread_mutex_lock(&myConsoleMutex);
    while (count-->0) {
        // Print out message and sleep for 1 second
        printf("Output from printout called with %d\n", j);
        sleep(1);
    }
    pthread_mutex_unlock(&myConsoleMutex);
}

void main() {
    int index=10;
    pthread_t tid;
    int ret, status;

    if (ret = pthread_mutex_init(&myConsoleMutex, NULL)) {
        perror("Mutex");
        exit(1);
    }

    while (index-->0) {
        if (ret = pthread_create(&tid, NULL, printout, (void*)index))
            perror("thread_create");

        sleep(1);
    }

    // sleep for 40 seconds just to make sure that
    // all the threads have finished before we exit
    sleep(40);
}
```

Figure 5-6 Basic Threaded Application using Mutual Exclusion with Pthreads

Semaphores

We won't examine semaphores in as much detail as mutual exclusion since they are less commonly used. We summarise their usage in Win32 and in Pthreads below.

In Win32 a semaphore is created using the `CreateSemaphore()` call. The prototype is shown below:

```
HANDLE CreateSemaphore( LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  
                        LONG lInitialCount,  
                        LONG lMaximumCount,  
                        LPCTSTR lpName  
                        );
```

The parameters have the following meaning:

`lpSemaphoreAttributes`:

Pointer to a `SECURITY_ATTRIBUTES` structure that determines whether child processes can inherit the returned handle. If `lpSemaphoreAttributes` is `NULL`, the handle cannot be inherited.

`lInitialCount`

Specifies an initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to `lMaximumCount`.

`lMaximumCount`

Specifies the maximum count for the semaphore object. This value must be greater than zero.

`lpName`

Points to a null-terminated string specifying the name of the semaphore object. If `lpName` is `NULL`, the semaphore object is created without a name.

The `wait()` and `signal()` functions are implemented using `WaitForSingleObject()` and `ReleaseSemaphore()`.

When using Pthreads, a semaphore is initialised using the `sem_init()` function. The prototype is shown below:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

The `pshared` variable indicates whether the semaphore can be shared between processes. A non-zero value indicates that it can.

The `value` variable is the initial value of the internal count.

The `wait()` and `signal()` functionality is provided by the `sem_wait()` and `sem_signal()` functions.

Excercise 5-1

Write a Unix program that creates a child process. The child process should print out a message saying "This is the child process id" followed by its process id. It should then exit. The parent should print out it's process id similarly.

Note, use the `getpid()` system call to retrieve your process id.

Excercise 5-2

Compile and run the multithreading examples in figures 5-3 and 5-4. Also, compile and run the synchronisation examples in figures 5-5 and 5-6. Compare the execution of the two.

Excercise 5-3

At the moment, the multithreaded examples use a sleep at the end to avoid quitting before the last thread has finished. Obviously this is not a good permanent solution. Rewrite one of examples 5-5 and 5-6 so that it waits until all of the threads are finished and then quits.

HINT: One approach would be to use recursive semaphores.

Concurrency in the Server

Iterative and Concurrent Servers

In practice a server must be able to handle more than one client connection. There are a couple of mechanisms that allow us to do this. These mechanisms are examined in detail in the next section (Asynchronous and Non-blocking Sockets).

One basic design decision to make is whether to use an iterative or concurrent server. The basic algorithm for an iterative server is shown in figure 5-7 below. This can be implemented using the `select()` system call described in the next section.

```
listen_for_connections()

while (not finished)
do
  foreach connection
  do
    check_connections()
    perform_io()
  done
done
```

Figure 5-7 Basic Algorithm for Iterative Server

An example of this type of approach is shown in figure 5-11.

The basic algorithm for a concurrent server is described in figure 5-8 below. For each new connection that is created, a new job is created to handle communication over that connection. A new job may be a process or a thread. The basic algorithm is the same regardless of which is used.

```
listen_for_connection()

foreach connection
do
  create_new_job(handle_connection)
done
```

Figure 5-8 Basic Algorithm for Concurrent Server

Multiprocessing vs. Multithreaded Servers

Creating a new process to handle each connection is generally a bad idea. This doesn't scale very well since the number of processes can rapidly multiply. The high overhead of context switching between processes makes this an unattractive approach.

The alternative to this, is to create a new thread to handle each new connection. This is a much more attractive solution since threads consume much fewer resources than processes. Thus, a multi-threaded concurrent server should scale much better than a multi-processing concurrent server.

Concurrency can lead to simpler, more scalable design however, when using multiple threads, the development overhead of co-ordinating thread access to shared resources must be considered.

An Example Multiprocessing and Multithreading Server

In Chapter 3, we created a basic server that could handle a single connection at a time. If multiple requests were made then the requests were queued and each client had to wait for the previous client to quit before it could proceed.

Having examined multiprocessing (using the `fork()` system call) and multithreading, we are now in a position to implement a server that can handle multiple simultaneous connections.

We will examine the multiprocessing option first, that is, the server creates a new process to handle each new connection. The code for Unix systems is shown in figure 5-9 below.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

const int MYPORT = 3490;      // The port users will be connecting to
const int BACKLOG = 5;      // Number of pending connections

void main() {
    int sockfd, new_fd;      // listen on sock_fd,
                           // new connection on new_fd
    struct sockaddr_in my_addr; // my address information
    struct sockaddr_in remote_addr; // connector's address information
    int sin_size;
    char rbuf[1024];
    char sbuf[1024];
    int rval;

    strncpy (sbuf, "Welcome to Your New Server\r\n", sizeof(sbuf));

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;      // host byte order
    my_addr.sin_port = htons(MYPORT);  // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
    memset(&(my_addr.sin_zero), 8, 0); // zero rest of the struct

    if (bind(sockfd, (struct sockaddr *)&my_addr,
             sizeof(struct sockaddr)) == -1) {
        perror("bind");
        exit(1);
    }
}
```

```

sin_size = sizeof(struct sockaddr_in);
printf ("Socket listening on port %#d.\n",ntohs(my_addr.sin_port));

// Start accepting connections
if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
}
do {
    if ((new_fd = accept(sockfd, (struct sockaddr *)&remote_addr, \
        &sin_size)) == -1)
    {
        perror("accept");
    }
    else
    {
        if (!fork())
        {
            printf("Created new connection\n");
            if (rval = send(new_fd, sbuf, sizeof(sbuf), 0) == -1) {
                perror("Writing message stream");
            }

            do {
                memset(rbuf, 0, sizeof(rbuf));

                if ((rval = recv(new_fd, rbuf, sizeof(rbuf), 0)) ==
-1)
                {
                    perror("Reading message stream");
                }

                if (rval != 0)
                {
                    printf("--->%s\n", rbuf);
                }
            } while (rval != 0);

            printf("Ending connection.\n");

            // close the socket
            close(new_fd);
        } else {
            close(new_fd); // parent doesn't need this
        }
    }
} while(1);
close(sockfd);
exit(0);
}

```

Figure 5-9 Basic Multiprocessing Server

If you compare this code to that in Chapter 2, you should see that the only significant difference is the addition of the `'if (!fork)'` line. This simple addition has dramatically increased the usefulness of our server.

We now look at the same server using threads instead of processes to handle additional connections. The code in figure 5-10 below shows the multithreaded version of the server. The server was written for a Unix environment using Pthreads.

One point to note is that we have made a minor structural change to the code. The code that handles new connections is placed inside a function called `handle_connection()`. This is so that this part of the server can be put inside a thread. Apart from this, everything else should look familiar.

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <unistd.h>

const int MYPORT = 3490;      // The port users will be connecting to
const int BACKLOG = 5;
char sbuf[1024];             // sbuf should be visible to all threads

void handle_connection(void* sockfd){
    int rval;
    char rbuf[1024];
    int sock_fd = (int)sockfd;

    printf("Created new connection\n");
    if (rval = send(sock_fd, sbuf, sizeof(sbuf), 0) == -1) {
        perror("Writing message stream");
    }

    do {
        memset(rbuf, 0, sizeof(rbuf));

        if ((rval = recv(sock_fd, rbuf, sizeof(rbuf), 0)) == -1) {
            perror("Reading message stream");
        }

        if (rval != 0) {
            printf("---->%s\n", rbuf);
        }
    } while (rval != 0);

    printf("Ending connection.\n");

    // close the socket
    close(sock_fd);
}

```

```

void main() {
    int sockfd, new_fd;           // listen on sock_fd
    struct sockaddr_in my_addr;   // my address information
    struct sockaddr_in remote_addr; // connector's address information
    int sin_size;
    int rval;
    pthread_t tid;
    int ret;

    strncpy (sbuf, "Welcome to The Threaded Server\r\n", sizeof(sbuf));

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;           // host byte order
    my_addr.sin_port = htons(MYPORT);      // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY;  // auto-fill with my IP
    memset(&(my_addr.sin_zero), 0, 8);     // zero rest of the struct

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) \
        == -1)
    {
        perror("bind");
        exit(1);
    }

    sin_size = sizeof(struct sockaddr_in);

    printf ("Listening on port %#d.\n", ntohs(my_addr.sin_port));

    // Start accepting connections
    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    do {
        if ((new_fd = accept(sockfd, (struct sockaddr *)&remote_addr, \
&sin_size)) == -1) {
            perror("accept");
        } else {
            if (ret = pthread_create(&tid, NULL, handle_connection, \
(void*)new_fd))
                perror("thread_create");
        }
    } while(1);
    close(sockfd);
    exit(0);
}

```

Figure 5-10 Basic Server Using Pthreads

Excercise 5-4

Adapt the multithreaded server in figure 5-10 to work in Win32. This should be a simple case of replacing the Pthreads `pthread_create()` function with it Win32 equivalent, `_beginthread()`.

Asynchronous and Non-blocking Sockets

Asynchronous Sockets

A socket can be either blocking or non-blocking. By default, sockets are blocking. This means that when you call a function (for example, `send()`, `recv()` etc.) on the socket, it will not return until it has completed its task, or has failed while trying.

Calls on non-blocking sockets return immediately, even if they cannot complete their task. Although this allows the program to do other things while the network operations finish, it requires that the program frequently *poll* the network to keep apprised of its current state. When you call `recv()` on a non-blocking socket that has no data to be read, it will return an error and set the `errno` to `EWOULDBLOCK`.

A socket can be made non-blocking using the `fcntl` call to set the `FNDELAY` flag using the `F_SETFL` command. A potential (but not recommended) usage of non-blocking sockets is shown in figure 5-11 below.

```
#include <fcntl.h>

...

int sockfd, numbytes;
char buf[MAXDATASIZE];
...
sockfd = socket(AF_INET, SOCK_STREAM, 0);
...

// Set the socket to be non-blocking
if (fcntl(sockfd, F_SETFL, FNDELAY) < 0)
{
    perror("fcntl F_SETFL, FNDELAY");
    exit(1);
}
...

// Wait for input to arrive
while (numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1)
{
    // Handle any errors
    if (errno != EWOULDBLOCK)
        handle_error();

    // Do something while we're waiting for input
    perform_some_task();
}

// Received input
handle_input(buf);
```

Figure 5-11: An inefficient use of non-blocking sockets

The code in figure 5-11 is very resource intensive due to the main while loop. One solution is to use another class of sockets called *asynchronous sockets*. These are similar to non-blocking sockets in that calls on them will return immediately. The difference is that the kernel sends the program a signal whenever an event occurs.

Asynchronous sockets are handled quite differently on Unix and Win32 environments. In Unix, we perform the following two steps:

- The process registers its process id with the socket so that the socket can send a signal (SIGIO is used on most systems and most portable).
- The socket is set to be asynchronous.

The socket can be read inside a signal handler. An example use of this is shown in figure 5-12. In Win32, a similar effect is achieved using WSAAsyncSelect() (see section Socket I/O with Windows Messaging).

```

#include <fcntl.h>
...

void get_async_input(int sig)
{
    if (numbytes=recv(sockfd, buf, MAXDATASIZE, 0)) == -1)
    {
        perror("recv");
    }
    printf("Received input -->%s\n", buf);
}

int main()
{
    int sockfd, numbytes, flags;
    char buf[MAXDATASIZE];
    ...

    void get_async_input(int);

    // Set get_async_input() to be our signal handler
    sigset(SIGIO, get_async_input);

    ...
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    ...

    // Register this process so we can receive signals
    if (fcntl(sockfd, F_SETOWN, getpid()) < 0)
    {
        perror("fcntl F_SETOWN");
        exit(1);
    }

    // Get the existing flags
    if ((flags = fcntl(sockfd, F_GETFL, 0)) == -1)
    {
        perror("fcntl F_GETFL 0");
        exit(1);
    }

    // Set the FASYNC flag
    if ((flags = fcntl(sockfd, F_SETFL, (flags|FASYNC))) == -1)
    {
        perror("fcntl F_SETFL FASYNC");
        exit(1);
    }

    // Rest of program

```

Figure 5-12: Using asynchronous sockets in Unix

Integrating Socket I/O with Windows Messaging: WSAAsyncSelect()

Socket activity can be integrated with Windows messaging by means of the `WSAAsyncSelect()` function. This function has two effects: it puts the socket into nonblocking mode, and it asks to have a message posted to a window whenever certain events occur. It has the following prototype:

```
int WSAAsyncSelect(SOCKET s, HWND hWnd, unsigned int wMsg, long lEvent);
```

The arguments have the following meaning:

`s` A descriptor identifying the socket we are monitoring

`hWnd` A handle to the window that will receive a message when any of the network events occurred.

`wMsg` The message to be received when a network event occurs.

`lEvent` A bitmask that specifies a combination of network events in which the application is interested.

The possible events that can occur are listed in table 5-1 below. To receive notification of multiple events, you can OR the flags together. Setting the event to zero cancels notification for the socket.

Value	Meaning
FD_READ	Data available for reading
FD_WRITE	Socket can be written to
FD_OOB	Out-of-band data has arrived
FD_ACCEPT	A client has requested a connection
FD_CONNECT	A client has successfully connected to the socket
FD_CLOSE	A peer has closed the connection
FD_QOS	Quality of Service (QOS) has changed (Winsock 2.0 only)
FD_GROUP_QOS	QOS of the sockets group has changed (Winsock 2.0 only)

Table 5-1 Events listened for by WSAAsyncSelect()

It is not necessary to reissue the `WSAAsyncSelect` after every event. It is automatically set when an *enabling function* is called.

In the case of `FD_READ` or `FD_OOB`, the enabling functions are `read()`, `recv()`, `recvfrom()`, `ReadFile()`, `ReadFileEx()`, `WSARecv()` and `WSARecvFrom()`.

In the case of `FD_WRITE`, they are `write()`, `send()`, `sendto()`, `WriteFile()`, `WriteFileEx()`, `WSASend()` and `WSASendTo()`.

`WSAAsyncSelect()` is an essential part of Windows sockets programming as it implicitly multithreads an application. This avoids the programming overhead of explicitly managing threads.

A typical use of `WSAAsyncSelect` by a client and server is described below:

Use of `WSASyncSelect()` in a server

- Create a socket and bind your address to it.
- Call `WSAAsyncSelect()` and request `FD_ACCEPT` notification.
- Call `listen()` and either perform other tasks or sleep.
- When a connection is requested, the specified window receives the message and the `FD_ACCEPT` notification. Call `accept()` to complete the connection.
- Call `WSAAsyncSelect()` on the socket created by `accept`. Specify read, out-of-band or close notification, that is, `FD_READ | FD_OOB | FD_CLOSE`.
- Call the appropriate read function (`recv()`, `ReadFile()` etc.) on `FD_READ` or `FD_OOB` notification.
- Call `closesocket()` on `FD_CLOSE` notification.

Use of of WSASyncSelect() in a client

- Create a socket.
- Call `WSASyncSelect()` and request `FD_CONNECT` notification.
- Call `connect()` and either perform other tasks or sleep.
- When `FD_CONNECT` notification is received, request `FD_WRITE | FD_READ | FD_OOB | FD_CLOSE` notification on the socket.
- When `FD_WRITE` notification is received, the client is ready to send requests to the server.
- Call the appropriate read function (`recv()`, `ReadFile()` etc.) on `FD_READ` or `FD_OOB` notification.

Applying overlapped I/O to sockets

Events

Events are synchronisation devices used in the Win32 environment. Events can be in one of two states - signalled and non-signalled. When used with I/O, they are set to non-signalled when the I/O begins and set to signalled when the operation completes.

Events are created using the `CreateEvent()` call. A thread may wait on an event to become signalled by calling `WaitForSingleObject()` with the event handle. If the event is signalled then the thread can proceed, otherwise it will sleep until the event becomes signalled.

Events come in two types - auto-reset and manual reset. An auto-reset signal allows one thread to proceed and immediately resets. A manual reset event allows any thread to run once it is in the signalled state. It becomes non-signalled only when a new operations commences or when `ResetEvent()` is called.

Overlapped I/O

Overlapped I/O is a mechanism that allows your programs to initiate I/O on a file, socket or other communication device, and then continue with other operations.

For each I/O operation, you must create a manual-reset event using `CreateEvent()`. When the operation commences, Windows automatically puts the event into the non-signalled state. When the operation completes, Windows signals the event. The result can be examined by calling `GetOverlappedResult()`.

Reading and writing is performed using the `ReadFile()` and `WriteFile()` functions respectively. In the case of overlapped I/O being used, these functions may return false. This does not always indicate that an error has occurred. Use `GetLastError()` to examine the last error. If the last error is `ERROR_IO_PENDING`, then the operation is proceeding and no error has occurred.

`ReadFile` and `WriteFile` take the same arguments:

```
BOOL ReadFile(
    HANDLE          hFile,          // File handle to read from
    LPVOID          lpBuffer,       // Buffer to read into
    DWORD           dwBytes,       // Number of bytes to read
    LPDWORD         lpBytes,       // Number of bytes read
    LPOVERLAPPED   lpOverlapped);

BOOL WriteFile(
    HANDLE          hFile,          // File handle to write to
    LPVOID          lpBuffer,       // Buffer to write from
    DWORD           dwBytes,       // Number of bytes to write
    LPDWORD         lpBytes,       // Number of bytes written
    LPOVERLAPPED   lpOverlapped);
```

The `HANDLE` may be a socket descriptor. If the socket has been created with the Winsock 1.1 `socket()` function then it is automatically usable. If you are using Winsock 2 then you must use `WSASocket()` and set the `WSA_FLAG_OVERLAPPED` bit.

The last argument is the most significant. It is a pointer to an `OVERLAPPED` structure. This structure has the following definition:

```
typedef struct _OVERLAPPED {
    DWORD   Internal;
    DWORD   InternalHigh;
    DWORD   Offset;
    DWORD   OffsetHigh;
    HANDLE  hEvent;
} OVERLAPPED;
```

The members of the structure have the following meaning:

Internal

Reserved for operating system use. When the operation is complete this should contain the value of any error that occurred.

InternalHigh

Reserved for operating system use. When the operation is complete this should specify the length of the data transferred.

Offset

Specifies a file position at which to start the transfer. The file position is a byte offset from the start of the file. This value is ignored when reading from or writing to sockets and other communications devices.

OffsetHigh

Specifies the high word of the byte offset at which to start the transfer. This value is ignored when reading from or writing to sockets and other communications devices.

hEvent

Identifies an event set to the signaled state when the transfer has been completed. The calling process sets this member before calling the `ReadFile()` or `WriteFile()` functions.

Waiting for I/O with `select()`

One facility often used in developing applications is the ability to multiplex I/O requests among multiple sockets and/or files. This is done using the `select()` call. `select()` provides a synchronous multiplexing scheme.

```
#include <sys/time.h>
#include <sys/types.h>
...

fd_set readmask, writemask, exceptmask;
struct timeval timeout;
...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

`select()` gives you the power to monitor several sockets at the same time. The arguments to `select()` are:

- `nfds` The range of file descriptors to be examined. To examine all open file descriptors, set this value to be one plus the largest descriptor.
- `readmask`: A set of file descriptors that we are interested in reading from.
- `writemask`: A set of file descriptors that we are interested in writing to.
- `exceptmask`: A set of file descriptors that we are interested in detecting exceptional conditions on. The only exceptional condition currently implemented by the socket is out of band data.
- `timeout`: This specifies a timeout period for the call. If the timeout is 0, then the call returns immediately. If the argument is `NULL`, then the call blocks indefinitely. Otherwise the call will timeout if none of the sockets in the sets it monitors have been accessed.

The timeout value supplied is actually a struct, as shown below:

```
struct timeval
{
    int tv_sec;        // seconds
    int tv_usec;      // microseconds
};
```

Note that the second field is microseconds, not milliseconds as might be expected. Also, the standard Unix timeslice is 100 milliseconds so you may not have microsecond time resolution.

The `readmask` parameter identifies the sockets that are to be checked for readability. If the socket is currently in the *listen* state, it will be marked as readable if an incoming connection request has been received such that an *accept* is guaranteed to complete without blocking. For other sockets, readability means that queued data is available for reading such that a call to `recv` or `recvfrom` is guaranteed not to block.

For connection-oriented sockets, readability can also indicate that a request to close the socket has been received from the peer. If the virtual circuit was closed gracefully, then a `recv` will return immediately with zero bytes read. If the virtual circuit was reset, then a `recv` will complete immediately with an error code such as `WSAECONNRESET`. The presence of out-of-band data will be checked if the socket option `SO_OOBINLINE` has been enabled (see `setsockopt()`).

The `writemask` parameter identifies the sockets that are to be checked for writability. If a socket is processing a connect call (nonblocking), a socket is writable if the connection establishment successfully completes. If the socket is not processing a connect call, writability means a `send` or `sendto` are guaranteed to succeed. However, they can block on a blocking socket if the `len` parameter exceeds the amount of outgoing system buffer space available. It is not specified how long these guarantees can be assumed to be valid, particularly in a multithreaded environment.

The `exceptmask` parameter identifies the sockets that are to be checked for the presence of out-of-band data or any exceptional error conditions.

The sets of file descriptors that we pass to `select()` are of type `fd_set` (`FD_SET` in Win32). How exactly this type represents the set of file descriptors is system dependent, but they are manipulated in the same way, using the following macros:

```
FD_ZERO(fd_set *set)           // Clears a file descriptor set
FD_SET(int fd, fd_set *set)    // Adds fd to the set
FD_CLR(int fd, fd_set *set)    // Removes fd from the set
FD_ISSET(int fd, fd_set *set)  // Tests to see if fd is in the set
```

Note that if the user is not interested in certain conditions (i.e., read, write, or exceptions), the corresponding argument to the `select` should be a null pointer.

The `select` function returns the total number of socket handles that are ready and contained in the `fd_set` structures, zero if the time limit expired, or -1 if an error occurred. When `select()` returns, the code must check every file descriptor (using `FD_ISSET`) that it is interested in, in each set.

The example in figure 5-13 below reads data from two sockets, `s1` and `s2`, as it is available from each, and writes to socket `s3`. There is a ten-second timeout.

```

fd_set read_set, write_set;
struct timeval wait;
int s1, s2, s3;
...
for (;;)
{
    wait.tv_sec = 10;           // 10 seconds
    wait.tv_usec = 0;

    FD_ZERO(&read_set);
    FD_ZERO(&write_set);

    FD_SET(s1, &read_set);
    FD_SET(s2, &read_set);
    FD_SET(s3, &write_set);

    nb = select(FD_SETSIZE, &read_set, &write_set, (fd_set *) 0,
&wait);

    // An error occurred
    if (nb < 0)
        handle_error();

    if (FD_ISSET(s1, &read_set))
    {
        // Socket s1 is ready to be read from.
    }

    if (FD_ISSET(s2, &read_set))
    {
        // Socket s2 is ready to be read from.
    }

    if (FD_ISSET(s3, &write_set))
    {
        // Socket s3 can be written to,
        // ie. a connection has been requested
    }
}

```

Figure 5-13 Using select() to manage multiple connections

One use for `select()` is for creating a *port monitor*. A port monitor is a program that creates several sockets that each listen on a port.

Adding each socket descriptor to the read set allows our program to listen for incoming connections on each of those ports. When it detects an incoming connection on one the sockets, it calls `accept()`. The port monitor may create a new process or thread for each new connection rather than deal with each socket directly.

6

Client Server Design

Fundamental Choices

- Connection Oriented vs. Connectionless Transport
- Choosing Between Single-Threaded, Multi-Threaded and Multi-Processing Designs
- Blocking vs. Non-Blocking I/O

Recent Developments

- New Features in Winsock 2
- Microsoft's WinInet Library
- The Impact of IPv6

Fundamental Choices

Connection Oriented vs. Connectionless Transport

Connection oriented transport reduces the complexity of program design since the data is guaranteed to be delivered and to be in order. In order to provide this guarantee, the transport layer must perform some complicated bookkeeping. For example, every TCP segment that is sent, has a sequence number in it. These sequence numbers are used to keep track of the order of the incoming packets. A program cannot receive a packet until the immediately preceding packet has been received.

This overhead is acceptable to programmers most of the time however sometimes it is unacceptable. If the amount of data being sent is very small then connectionless transport may be preferable.

One of the problems with connection oriented transport is that the data is sequenced by the transport provider. Most of the time this is useful however, some real time applications may not want to block waiting for a single segment. If the application is sufficiently fault tolerant then it may be able to proceed with incomplete data. An example of such an application might be real time audio and video. Since the data is being presented to humans, a certain amount of missing data may be tolerable.

Blocking vs. Nonblocking I/O

Non blocking I/O calls return immediately. This allows programs to issue I/O commands and then proceed with other tasks. Blocking calls cause the program to wait until the I/O operation has completed.

Most useful servers require the ability to handle multiple connections simultaneously. If the server uses blocking calls on sockets then one slow connection will block other connections. One solution is to spawn off a new process for each new connection. This prevents any one particular connection from holding up other connections. However, the number of processes can grow rapidly which has a serious impact on system performance.

One alternative is to use non-blocking sockets combined with `select()`. `select()` allows us to find out which sockets need attention at a given time. This saves us the overhead of creating new processes every time a new connection is created.

For a long time, the method described above was the standard approach to writing a server. Since threads have become reasonably commonplace it has become feasible to use blocking sockets with each connection being managed by a thread rather than a process. Threads impose much less on the system and scale much better than processes.

Recent Developments

New features in Winsock 2

One of the most important new features is official support for multiple transport protocols. Although Winsock 1.1 was not actually limited to TCP/IP, that was the only protocol that had official support written into the spec. There was no standard way for a vendor to add support for another transport protocol, though a few vendors did do proprietary implementations of other protocols. With Winsock 2, official support for OSI, Novell IPX/SPX and Digital's DECNet exists in the specification, and it's now possible to add support for other protocols in a standard way. More importantly, a program can be written to be transport-independent, so that it works with all of these protocols, without change.

Winsock 2 also adds support for technical initiatives like Quality of Service (QoS) and Multicasting. These technologies will become increasingly important as bandwidth requirements become more regimented and intense. For example, the QoS mechanism would allow a videoconferencing program to reserve a certain amount of bandwidth so that a sudden file transfer, for example, doesn't cause its video to begin breaking up due to a lack of bandwidth.

The multicasting support would allow you to hold a videoconference with multiple people on the same LAN but have each videophone only send a single stream of data. (The current alternative requires every phone to send a stream of data to every other conference participant. This is wasteful and can quickly bog down a common LAN.)

Another important feature of Winsock 2 is complete integration with Win32's unified I/O mechanisms. For example, it is possible to use the Win32 ReadFile() API on a socket instead of recv(). More importantly, Windows NT's overlapped I/O mechanisms can now be used with sockets; Winsock 2 on Windows 95 also implements overlapped I/O with sockets, though it is faked, because Windows 95 does not support overlapped I/O natively.

Winsock 2 also allows for "Layered Service Providers." This enables many interesting features, such as security plug-ins: for example you could add an Secure Sockets Layer (SSL) provider to encrypt your data. This will be completely transparent to your network applications.

Microsoft's WinInet Library

The WinInet library is a collection of MFC classes that provide high level access to internet resources. The WinInet classes are designed for rapid development of internet clients. The basic WinInet class is the `CInternetSession`.

In order to read or write data, you must create a `CInternetFile`. The easiest way to do this is to call `CInternetSession::OpenURL` with the URL of the site you want to access. The function parses the URL and can determine whether to return a `CHttpFile` or `CGopherFile` (FTP is handled differently). `CInternetSession::OpenURL` can also work with local files (returning a `CStdioFile` instead of a `CInternetFile`).

WinInet classes provide developers with a number of useful features that speed up development:

- Buffered I/O
- Default parameters for many functions
- Exception handling for common Internet errors
- Automatic cleanup of open handles and connections

Some of the more common generic WinInet operations as well as FTP and HTTP specific operations are described below:

General Internet URL (FTP, Gopher, or HTTP)

Establish an HTTP connection	Create a CInternetSession Call CInternetSession::GetHttpConnection
Open an HTTP file.	Establish an HTTP connection. Call CHttpConnection::OpenRequest to create a CHttpFile object. Call CHttpFile::AddRequestHeaders. Call CHttpFile::SendRequest.
Read an HTTP file.	Open an HTTP file. Call CInternetFile::Read.
Get information about an HTTP request.	Establish an HTTP connection. Call CHttpConnection::OpenRequest Call CHttpFile::QueryInfo.

FTP Operations

Establish an FTP connection.	Create a CInternetSession Call CInternetSession::GetFtpConnection
Open an FTP file.	Establish an FTP connection. Call CFtpConnection::OpenFile
Read an FTP file.	Open an FTP file with read access. Call CInternetFile::Read.
Write to an FTP file.	Open an FTP file with write access. Call CInternetFile::Write.
Change the client's directory on the server.	Establish an FTP connection. Call CFtpConnection::SetCurrentDirectory.
Retrieve the client's current directory on the server	Establish an FTP connection. Call CFtpConnection::GetCurrentDirectory.

HTTP Operations

Establish a connection.	Create a CInternetSession
Open a URL.	Establish a connection. Call CInternetSession::OpenURL.
Read URL data	Open the URL. Call CInternetFile::Read.
Query the setting of an Internet option	Establish a connection. Call CInternetSession::QueryOption.
Set an Internet option.	Establish a connection. Call CInternetSession::SetOption.

The impact of IPv6

IP Next Generation (IPng) is a new version of the Internet Protocol, designed as a successor to IP version 4. IPng is assigned IP version number 6 and is formally called IPv6.

IPng was recommended by the IPng Area Directors of the Internet Engineering Task Force at the Toronto IETF meeting on July 25, 1994, and documented in RFC 1752, "The Recommendation for the IP Next Generation Protocol". The recommendation was approved by the Internet Engineering Steering Group on November 17, 1994 and made a Proposed Standard.

IPng was designed to take an evolutionary step from IPv4. It was not a design goal to take a radical step away from IPv4. Functions which work in IPv4 were kept in IPng. Functions which didn't work were removed. The changes from IPv4 to IPng fall primarily into the following categories:

Expanded Routing and Addressing Capabilities

IPng increases the IP address size from 32 bits to 128 bits, to support more levels of addressing hierarchy and a much greater number of addressable nodes, and simpler auto-configuration of addresses.

The scalability of multicast routing is improved by adding a "scope" field to multicast addresses.

Anycast addressing

A new type of address called a "anycast address" is defined, to identify sets of nodes where a packet sent to an anycast address is delivered to one of the nodes. The use of anycast addresses in the IPng source route allows nodes to control the path which their traffic flows.

Header Format Simplification

Some IPv4 header fields have been dropped or made optional, to reduce the common-case processing cost of packet handling and to keep the bandwidth cost of the IPng header as low as possible despite the increased size of the addresses. Even though the IPng addresses are four times longer than the IPv4 addresses, the IPng header is only twice the size of the IPv4 header.

Improved Support for Options

Changes in the way IP header options are encoded allows for more efficient forwarding, less stringent limits on the length of options, and greater flexibility for introducing new options in the future.

Quality-of-Service Capabilities

A new capability is added to enable the labeling of packets belonging to particular traffic "flows" for which the sender requests special handling, such as non-default quality of service or "real-time" service.

Authentication and Privacy Capabilities

IPng includes the definition of extensions which provide support for authentication, data integrity, and confidentiality. This is included as a basic element of IPng and will be included in all implementations.

Growth is the basic issue which caused there to be a need for a next generation IP. Currently IPv4 serves what could be called the computer market. The computer market has been the driver of the growth of the Internet. The next phase of growth will probably not be driven by the computer market. Rather we are starting to see some new emerging markets. Some of these are listed below.

Nomadic Computing

The nature of nomadic computing requires an internet protocol to have built in authentication and confidentiality. It also goes without saying that these devices will need to communicate with the current generation of computers. The requirement for low overhead comes from the wireless media. Unlike LAN's which will be very high speed, the wireless media will be several orders of magnitude slower due to constraints on available frequencies, spectrum allocation, error rates, and power consumption.

Device Control

This consists of the control of everyday devices such as lighting equipment, heating and cooling equipment, motors, and other types of equipment which are currently controlled via analog switches and in aggregate consume considerable amounts of electrical power. The size of this market is enormous and requires solutions which are simple, robust, easy to use, and very low cost. The potential payback is that networked control of devices will result in cost savings which are extremely large.

Networked Entertainment

The first signs of this emerging market are the proposals being discussed for 500 channels of television, video on demand, etc. This is clearly a consumer market. The possibility is that every television set will become an Internet host. As the world of digital high definition television approaches, the differences between a computer and a television will diminish. As in the previous market, this market will require an Internet protocol which supports large scale routing and addressing, and auto configuration. This market also requires a protocol suite which imposes the minimum overhead to get the job done. Cost will be the major factor in the selection of an appropriate technology.

Appendix 1

Structure of an IP Datagram

The first five 32 bit words in an IP datagram comprise the packet header. The sixth 32 bit word may be used optionally. Since the header is of variable length, the Internet Header Length (IHL) gives the length of the header in words.

The header contains the following information:

- IP Version
- IHL
- Type of Service
- Total Length
- Identification
- Flags
- Fragmentation Offset
- Time to Live
- Protocol
- Header Checksum
- Source Address
- Destination Address
- Options

The structure of an IP packet is shown in figure 1 below.

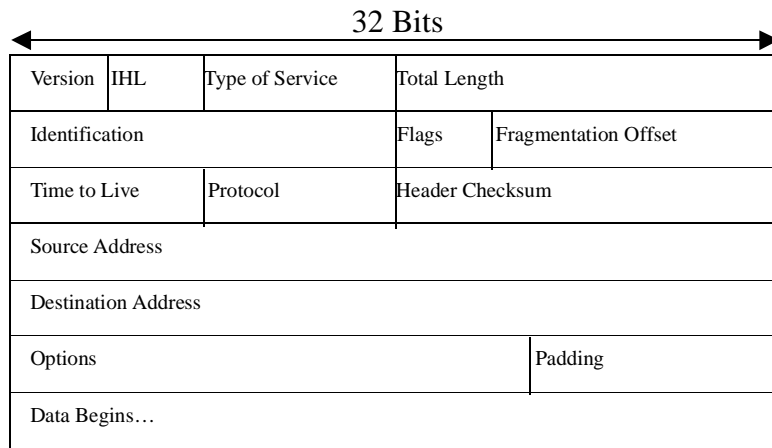


Figure 1 Structure of an IP Packet

Structure of a UDP Datagram

The UDP provider applies the formatting as shown in figure 2. As might be expected, UDP doesn't add much to the existing packet.

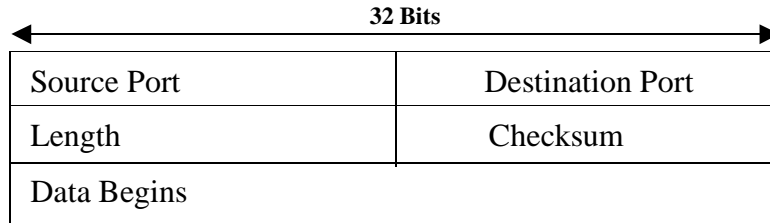


Figure 2 UDP Message Format

Structure of a TCP Segment

TCP has a much higher overhead than UDP. This is because of the stream oriented nature of TCP which requires much more flow control. The structure is correspondingly more complex than UDP and is shown in figure 3 below.

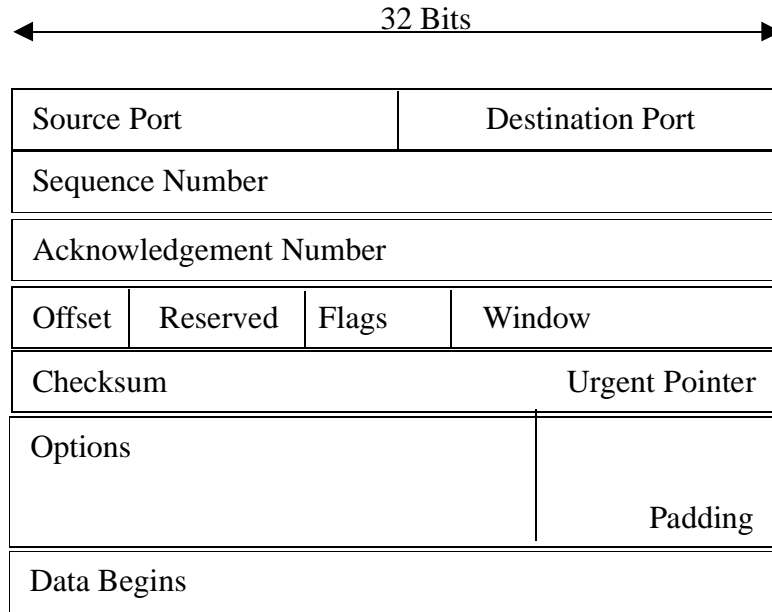


Figure 3 TCP Segment Format

Some of these fields are self-explanatory however, some noteworthy ones are:

Sequence Number

This identifies the sequential position in the data stream of the first byte in the in the segment. For example, if the stream had an initial sequence number (ISN) of 2300, and 2150 bytes have already been transferred, then the value for the sequence number will be 4450.

Acknowledgment Number

This number indicates how much data the receiver has received. For example, if the senders ISN was 3390 and 1100 bytes have been received so far, then the acknowledgement number will be 4490. This serves as acknowledgment of all bytes up to that number. Thus, the acknowledgment number is the number of the next byte that the receiver expects to receive. Note that the TCP/IP standard does not require one acknowledgement for every segment received.

Window

This contains the number of bytes that the receiver can accept. A zero window causes the sender to stop transmitting until a non-zero window is received.