

Network Programming and Design



Unit 6

Network programming



香港公開大學
THE OPEN UNIVERSITY
OF HONG KONG

科技學院 School of Science and Technology

Course team

Developers: Jacky Mak, Consultant

Designer: Ross Vermeer, ETPU

Coordinator: Dr Philip Tsang, OUHK

Member: Dr Steven Choy, OUHK

External Course Assessor

Prof. Cheung Kwok-wai, The Chinese University of Hong Kong

Production

ETPU Publishing Team

Copyright © The Open University of Hong Kong, 2009, 2012, 2013, 2014.

Reprinted 2018.

All rights reserved.

No part of this material may be reproduced in any form by any means without permission in writing from the President, The Open University of Hong Kong. Sale of this material is prohibited

The Open University of Hong Kong
Ho Man Tin, Kowloon
Hong Kong

This course material is printed on environmentally friendly paper.

Contents

Overview	1
An overview of network programming	3
The client/server model	3
Program vs process	5
System calls	8
The basics of sockets	10
What is a socket?	10
Basic socket properties	12
The connection mode of socket communication	14
Communication domains	14
Socket types	15
Connection-oriented mode	17
Connectionless mode	18
Socket programming	20
Data structures	20
Byte ordering functions	23
TCP socket programming (connection-oriented)	25
TCP server programming	26
TCP client programming	29
Lab 1: Implement a connection-oriented (TCP) application	31
UDP socket programming (connectionless)	35
UDP server programming	35
UDP client programming	36
Lab 2: Implement a connectionless (UDP) application	37
Remote procedure calls (RPCs)	41
Challenges for RPC facilities	42
RPC implementation	43
An RPC example using Perl's SOAP::Lite module	43
Summary	47
Suggested answers to self-tests and activities	48
Glossary	53
References	55
Online materials	55

Overview

In this unit we explore an exciting topic: network programming. In the past, students often had mixed reactions to this topic. Some of them found it a bit intimidating at first, since it required an intricate command of technical details and relationships with other technologies. But once they had carried through some actual network programming work, they often found it challenging and rewarding.

This unit is an overview of network programming which will let you gain a grasp on the fundamentals of the subject matter and have a look at some of its latest developments. After the foundation is laid, you will have the ability to pursue the subject further in the future as needs arise, e.g. as required by your job or by your studies of more advanced courses.

All in all, network programming is just a means to make it possible for two processes, which are located in different host systems, to exchange data and interactions with each other (i.e. communicate) over networks. We will begin by addressing some fundamental issues such as the client/server model, UNIX processes, and system calls. We'll confine our discussion to the UNIX environment for the sake of clarity. What you learn in this unit should apply to other common operating environments, so don't worry that we're not covering other OSs.

We will then explore two main modes of socket communications: connection-oriented (over TCP); and connection-less (over UDP). The essential system calls used to implement such communications will be examined in detail. TCP communications are employed by many popular Internet applications nowadays including http (Web browsing), ftp (file transfer), smtp (mails transfers), etc., while UDP communications are often found in network routing and control subsystems and real-time applications.

In addition to this unit's activities and self-tests, two lab exercises — one for each mode of communication — are included to help you master these topics.

The unit ends with an introduction to the Remote Procedure Call (RPC) model. The RPC model uses an application-oriented approach so that programmers can focus on solving the problem that an application is created for, while leaving it to the RPC to make the solution operable in a distributed environment. SOAP (Simple Object Access Protocol), one of the latest manifestations of RPC model, will also be introduced. A very elaborate sample SOAP application is provided so that you can give it a trial on our lab server.

In short, this unit:

- discusses the use of sockets for inter-process communication;
- describes different modes of socket communication;
- describes the process of socket programming with TCP;

- describes the process of socket programming with UDP; and
- explains the operations of Remote Procedure Calls (RPCs).

An overview of network programming

Network programming is one of the key technology components that have brought distributed computing into practical implementations. In essence, it helps any pair of processes establish a linkage between each other, and provides them with a conduit through which they can pass messages back and forth and exchange interactions. Riding upon this transport, applications can make available services such as file sharing, database queries and catalogue searching to any client connected to the network. Many well-known Internet applications such as the Worldwide Web (WWW), electronic mail (email), File Transfer Protocol (FTP), etc. use this model to deliver their services to their clients, which are enormous in size and scattered over the world.

In this topic we will work through some background topics, including the client/server model and UNIX processes. This will put you into a better position to tackle the core of the unit's subject matter, i.e. writing network programs. Remember that throughout this unit we will assume UNIX as our operating system so all of our discussion will be based upon it. But also remember that what you learn using UNIX as our OS should generally apply to other modern operating systems such as Linux, MS Windows, etc.

The client/server model

The client/server model is the predominant software design for distributed systems. It separates its functionality into two parts — clients and servers. A client is a program that uses the services provided by servers. It opens a communication channel to a particular server (*active open*), and then makes requests for services. A server, on the other hand, is a program that keeps its communication channel open to wait for requests (*passive open*). Upon receiving a request, the server will provide the client with the service requested.

Clients and servers are connected by computer networks, and a single server is typically used by multiple clients. This scenario is depicted below.

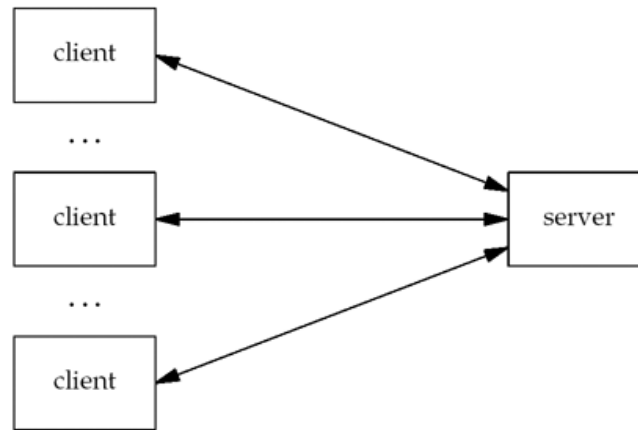


Figure 6.1 A typical client/server based application

For example, Web browsing applications are built on the client/server model. The Web browser is the client while the Web server program is the server. The client sends a request for a particular page to the server. In response, the server retrieves the requested webpage and sends it back to the client.

What functionality a client/server application should provide, and how this functionality is divided between its servers and clients are the issues related to application design (i.e. the application layer), which is not our primary concern in this unit.

What we need to look further into now is the way in which clients and servers pass data back and forth between each other over networks (i.e. the transport layer). This is where network programming or **socket** programming comes into play. In **TCP/IP**, a *socket* is defined as a communication endpoint of a connection. In the client/server model, each connection has two endpoints, one at the client side and one at the server side; data is transmitted through the connection between the client and the server.

The figure below depicts the role of socket programming in the client/server and the TCP/IP models. Note that we will explore sockets and socket programming in depth later on.

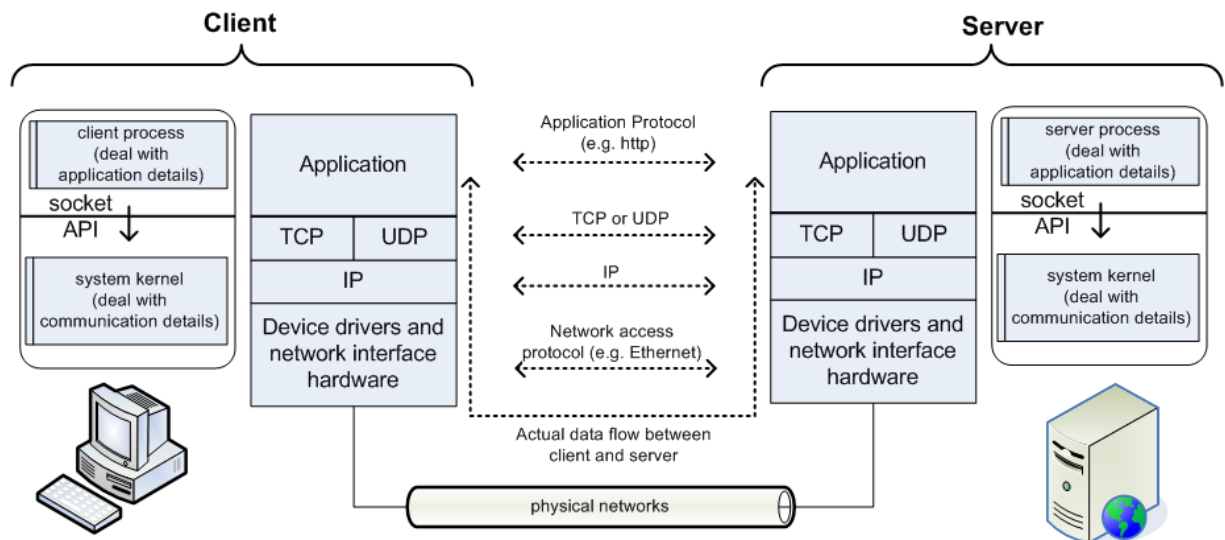


Figure 6.2 The relationship between the socket interface and TCP/IP model

For the client/server model that comprises the Internet, TCP/IP is predominately used as a transport vehicle. So how can we drive TCP/IP to work for network applications within a system? The answer is the socket application programming interfaces (i.e. the socket API). As you can see in Figure 6.2, the socket API sits between the application layers and transport layers within a host system and forms the logical conduit to link up the client and server.

Programmers can then focus on working out the functionality that the application is created for (e.g. serving requested home pages to clients) while leaving it to the UNIX system to deal with the details entailed in low-level network operations. As far as network programming is concerned, programmers should select the appropriate socket system calls, fill them with proper parameters, and get them executed. That is all!

Program vs process

Going back to square one, the purpose of network programming is to enable a pair of processes to communicate. But what is a process? In essence, a process is a program in execution. Programs are *stored in hard disks* as files, and are then loaded into memory to generate processes. A program can be loaded more than one time, spawning multiple instances of processes for execution. Each process has its own address space in the memory allocated by the **kernel** and its own set of variables, even if they are created from the same program. After a process completes its job and exits, it is unloaded from the memory and vanishes. But the program file from which the process originates remains unchanged. It can generate a new instance of the process the next time it is loaded.

You can use the command 'ps' — i.e. process status — to find out what processes are running in your computer system.

```

root> ps
      PID TTY          TIME CMD
  220901 pts/0        0:00.03  -sh  (sh)
  221120 pts/0        0:00.00   vi
  221109 pts/1        0:00.03  -sh  (sh)
  221145 pts/1        0:00.00  ./server.out
  221138 pts/2        0:00.03  -sh  (sh)
root>

```

Figure 6.3 The output of the ‘ps’ (process status) command

As you can see in Figure 6.3, an instance of ‘vi’ (an editor program) and an instance of ‘server.out’ (a self-developed server program using socket API), are running in the system being queried.

Process ID (PID)

In Figure 6.3 the term **PID** stands for process identification. UNIX issues the `fork()` system call to generate a process. Each process so generated bears a unique process identification (i.e. process ID or PID, which is an integer assigned by the kernel) that is subsequently used when referring to the process.

There are some special **processes** you should note. For instance, PID 0 is usually used by the scheduler process, which is often called the *swapper*. No program on disk corresponds to this process, which is part of the kernel and is therefore known as a *system process*. PID 1 is usually used by the *init* process, which is a *user process*, and which is invoked by the kernel at the end of the booting process. This process is responsible for bringing up a UNIX system and the parent or ancestor of all other user processes loaded subsequently into the system. On some virtual memory implementations of UNIX systems, PID 2 is the *pagedaemon*, which is the system process that supports the paging of the virtual memory system.

Parent process ID

Every process has a parent process and corresponding parent process identification (parent PID or PPID). The PPID is the PID of the process’s parent. The parent is the process that created the current process, which is its child. The following program prints the PID and the PPID of a process. A more detailed explanation of the system called `getpid()` and `getppid()` can be found in UNIX command manuals (to access them you can just type ‘man getpid’ in the UNIX shell).

```
#include <stdio.h>
main()
{
    printf("Process ID = %d, Parent process ID = %d\n",
        getpid(), getppid());
}
```

Figure 6.4 PID and PPID information of a process

The output of the above program looks like this:

```
Process ID = 8685, Parent process ID = 8159
```

The process with ID 8685 is the parent or creator of the process with ID 8159.

User and group IDs

The user identification (UID) is a numeric value that identifies a login user to the system. It is defined in the password file */etc/passwd*. This UID is assigned by the system administrator when a login name is assigned, and the user cannot change it. The UID is normally assigned to be unique for each user. The group identification (GID) is a group number that is defined in the file */etc/group*. Each process that runs is associated with the UID and GID of the login account that has put the process in execution.

Real and effective user and group IDs

The UID and GID identify who we really are. On the other hand, the effective user ID (EUID) and effective group ID (EGID) determine the file access permissions, i.e. what resources the process has permission to access. When we execute a program file, the EUID of the process is usually the UID, and the EGID is usually the GID. But the capability exists to set a special flag in the file's mode word that says 'when this file is executed, set the effective user ID of the process to be the owner of the file.' In this way, other users can execute this program as if the owner of the program were doing so.

Please refer to the reading below to learn more about UNIX processes and the different type of IDs related to processes you've been introduced to here.

Reading 6.1

Stevens, W R, and Rago, S A (2005) *Advanced Programming in the UNIX Environment*, 2nd edn, Boston: Addison-Wesley, Sections 8.2 & 4.4.

You should now be ready to tackle the following self-test.

Self-test 6.1

Write a program to identify your process UID, EUID, GID and EGID.

Activity 6.1

Find out how to use ‘ps’ command to display the following attributes of the processes running in your system:

parent process ID, process ID, process group ID, user ID, real user ID, process name

System calls

As you can see in Figure 6.5, the UNIX system kernel provides a system call interface to applications to allow them to communicate with others. The system deals with the intricacies of preparing data into formats conforming to network protocols requirements and passing them down into physical network interfaces. The applications themselves do not need to know much about this groundwork — they just have to invoke the appropriate socket system calls when they need to connect with their peer processes.

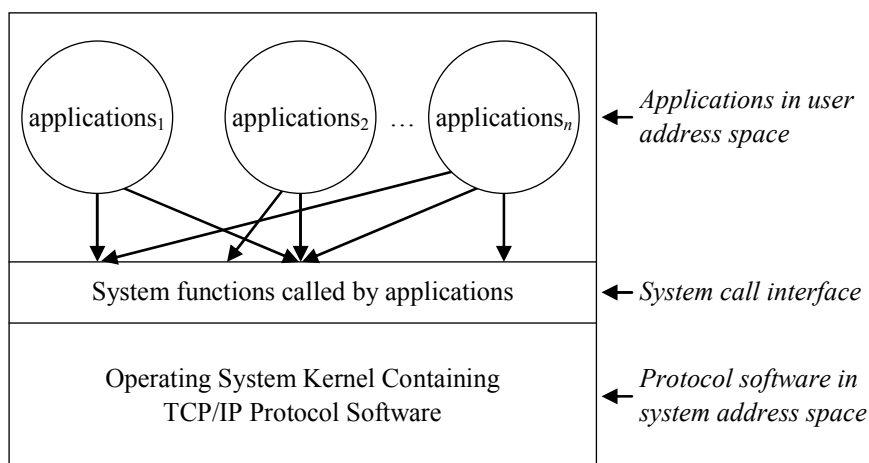


Figure 6.5 Applications interacting with TCP/IP protocol software through the system call interface provide by the kernel

Source: from Internetworking w/ TCP/IP vol. 3:61, Figure 4.1

The socket interface is one of inter-process communication (IPC) methods in UNIX. It enables communications between applications running on *different* systems. There are some other IPC methods in UNIX, but most of them are used mainly for communications within the *same* system. You will find using the system calls of a socket interface not much different from using other function calls. We will explore these calls in detail in coming sections.

The basics of sockets

A socket is one of IPC methods provided by UNIX. In the early 1980s, the Advanced Research Projects Agency (ARPA) funded a group at the University of California at Berkeley to transport TCP/IP software to UNIX, and to make the resulting software available to other sites. As part of the project, the designers created an interface that applications use to communicate with each other over the network. The result became known as the socket interface, or the Berkeley socket interface or BSD (Berkeley Software Distribution) socket interface.

What is a socket?

In TCP/IP, a **socket** is defined as a *communication endpoint* of a connection, which consists of an *IP address* and a **protocol port number**. Each connection has two endpoints, one at the client side and one at the server side, as illustrated in the figure below.

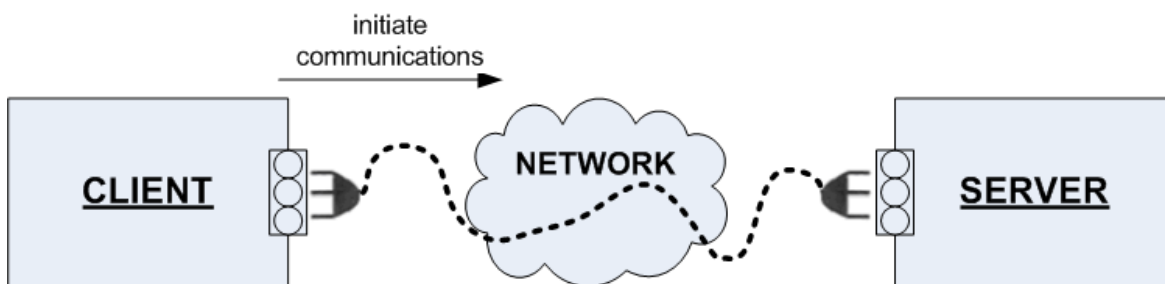


Figure 6.6 A connection made by sockets

Given its wide acceptance by hardware vendors, the BSD socket has ranked as a *de facto* standard. In this unit we therefore focus on the BSD socket. There are different implementations of socket interface in different operating systems (e.g. the Winsock of MS Windows). However, they use concepts and features similar to those for the BSD socket.

Socket descriptors vs file descriptors

To a certain extent, working with sockets is similar to working with files in the UNIX environment since both of them are considered to be input/output (I/O) operations. The following three steps are typical of I/O operations:

1 Create a descriptor:

Before using an I/O device (e.g. a file stored in hard disk), an application needs to obtain a descriptor (which is an integer number) from the system, and use it to refer to the device for access. Each process has its own descriptors tables. In UNIX, sockets and file descriptors share the same descriptor table. That is, if you open a file and it returns a file descriptor with a value of say 9, and then

immediately opens a socket, you will be given a file descriptor with value 10 to reference that socket.

The file descriptor values 0, 1 and 2 are pre-assigned to some specific I/O devices:

- 0 — standard input device (e.g. keyboard)
- 1 — standard output device (e.g. screen)
- 2 — standard error device

For file I/O, the `open()` system call is used for this step to create a *'file descriptor'*, while for socket I/O, the `socket()` is used to create a *'socket descriptor'*.

2 Read or write data:

After creating a descriptor, a process can refer to it to read data from or write data into the associated I/O device. The process can use such system calls as `read()/write()`, `send()/recv()`, etc. The exact choice of these calls depends on the type of the I/O device in use.

Just image a scenario in which a process writes some information to a particular I/O device, while another process then reads from that device. In effect, the I/O device provides a means for the two processes to exchange data, or to communicate with each other. This is a coarse analogy illustrating how sockets can connect two processes through I/O operations.

3 Close the descriptor:

Upon finishing the use of an I/O device, the process should close the descriptor with the `close()` system call. The system will relinquish the resources that have been allocated to support the I/O operations that use the device.

While there are similarities in the I/O operations of files and sockets, there are also some significant differences between them.

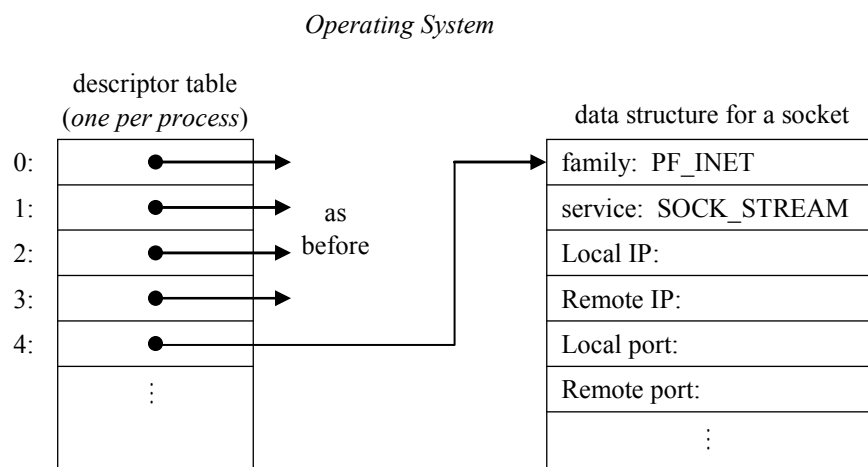
For example, sockets have addresses associated with them, while files do not have. This is a distinguishing feature of sockets. Another type of I/O device — pipes — works like a socket, but pipes are not associated with any address. Also, you cannot randomly access a socket in way as you can with a file (e.g. use the `lseek()` function). Sockets must be in the correct state to perform input or output. The table below briefly compares socket I/Os with file I/Os. Noting these distinctions may be quite helpful to you at the elementary stage.

Table 6.1 Comparison between the I/O operations of files and that of network sockets

File I/O	Network Socket I/O
open a file	open a socket
	name the socket
	associate with another socket
read and write	send and receive between sockets
close the file	close the socket

Basic socket properties

The BSD socket interface provides generalized functions that support network communications. TCP/IP is just one of the communication domains that can make use the socket interface. The programmer has to specify the basic properties of a socket when creating it in order to define the communication characteristics to be supported by the socket. In the following sections, we will examine these properties in detail. The figure below denotes conceptually the data structure that UNIX uses to hold the information related to a created socket.

**Figure 6.7** Conceptual data structure to represent a socket

Source: Comer and Stevens 1996, Fig. 5.2

Although the data structure contains many fields, the system leaves most of them unfilled when a socket is newly created. As you will see, the application that has created the socket needs to make additional system calls to fill in the information in the socket data structure before the socket can be used.

Self-test 6.2

Describe any two key differences between sockets and files I/O operations.

The connection mode of socket communication

Communication domains

The BSD socket interface supports network communication under different communication domains. Socket programming considers TCP/IP version 4 as one domain, while TCP/IP version 6 is another. The address of these two families consists of *an IP address and a port number*. We call this a **socket address**.

Apart from TCP/IP domains, the UNIX domain is another type of domain that can specify the IPC between two processes running on the same UNIX host. Its address consists of UNIX path names.

When creating a socket using the `socket()` system call, we should specify the communication domain in its first argument.

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);

Returns: file (socket) descriptor if OK, 1 on error
```

Figure 6.8 The `socket()` system call, for creating a socket

Figure 6.9 summarizes the domains supported by UNIX. The constants start with `AF_` (for address family) because each domain has its own format for representing an address.

Domains	Descriptions
<code>AF_INET</code>	IPv4 Internet domain
<code>AF_INET6</code>	IPv6 Internet domain
<code>AF_UNIX</code>	UNIX domain
<code>AF_UNSPEC</code>	Unspecified

Figure 6.9 Socket communication domains

In this unit, we focus on TCP/IP version 4 domain (`AF_INET`) since it is being used predominantly on the Internet.

Socket types

Another key attribute of a socket is its type, which further determines the communication characteristics of that socket. As you can see in Figure 6.8, the second argument of the `socket()` is to specify the type of socket. Do you need a reliable and connection-oriented transport service (e.g. TCP), or an unreliable and connectionless service (e.g. UDP) or, some other types of service? You have to select the appropriate type of sockets to be used for the socket communication. In *Unit 3* we covered **TCP** and **UDP** as part of the TCP/IP protocols. Please go back and revise that unit if necessary.

The following table lists the types of socket supported by UNIX.

Table 6.2 The types of 'socket' supported by UNIX

Types	Descriptions
SOCK_STREAM	Sequenced, reliable, bidirectional, connection-oriented byte streams, using TCP as underlying transport.
SOCK_DGRAM	Fixed-length, unreliable, connectionless messages, using UDP as underlying transport.
SOCK_RAW	Datagram interface to IP, bypassing transport layer protocols.
SOCK_SEQPACKET	Fixed-length, sequenced, reliable, connection-oriented messages

This unit, however, concentrates on the following two types of socket since again they are being used predominantly for the communications in the Internet:

- SOCK_STREAM — stream sockets for connection-oriented communication
- SOCK_DGRAM — datagram sockets for connectionless communication.

Below is a brief description on SOCK_RAW and SOCK_SEQPACKET for general reference:

- A SOCK_RAW socket provides a datagram interface directly to the underlying network layer (which means IP in the Internet domain). Applications are responsible for building their own protocol headers when using this interface, because the transport protocols (TCP and UDP, for example) are bypassed. Superuser privileges are required to create a raw socket to prevent malicious applications from creating packets that might bypass established security mechanisms.

- A `SOCK_SEQPACKET` socket is just like a `SOCK_STREAM` socket except that we get a message-based service instead of a byte-stream service. This means that a `SOCK_SEQPACKET` socket maintains a message boundary, while `SOCK_STREAM` does not. SCTP is a newer transport protocol, standardized in the IETF in 2000 (compared with TCP, which was standardized in 1981). It was first designed to meet the needs of the growing IP telephony market; in particular, transporting telephony signaling across the Internet. The requirements it was designed to fulfill are described in RFC 2719. SCTP is a reliable and message-oriented protocol. Since it is a newer transport protocol, it does not have the same ubiquity as TCP or UDP.

We'll now further explore the two main types of sockets used on the Internet: stream sockets (`SOCK_STREAM`) and datagram sockets (`SOCK_DGRAM`).

Stream sockets (`SOCK_STREAM`)

Before exchanging data over stream sockets, you should set up a logical connection with the peer that you want to communicate with first, i.e. before using the service of a **connection-oriented protocol** (i.e. TCP). After the connection is established, you can start to transmit or receive data with your counterpart. Upon finishing, you should close the connection. No further communication can be made over the socket after this.

As you can see, this scenario is much like making a phone call. You have a telephone socket and a telephone number, which is assigned or bound to the socket. Whenever you want to call your friend, you have to dial her phone number and then connect to her socket. The conversation is then transferred along the established connection. The established connection is dedicated to the conversation between you and your friend. After the conversation, you will say 'good bye' to your friend, and hang up the phone.

Connection-oriented mode

Figure 6.10 shows a typical scenario in which stream sockets are used to carry out communications between a client and a server over the connection-oriented TCP service.

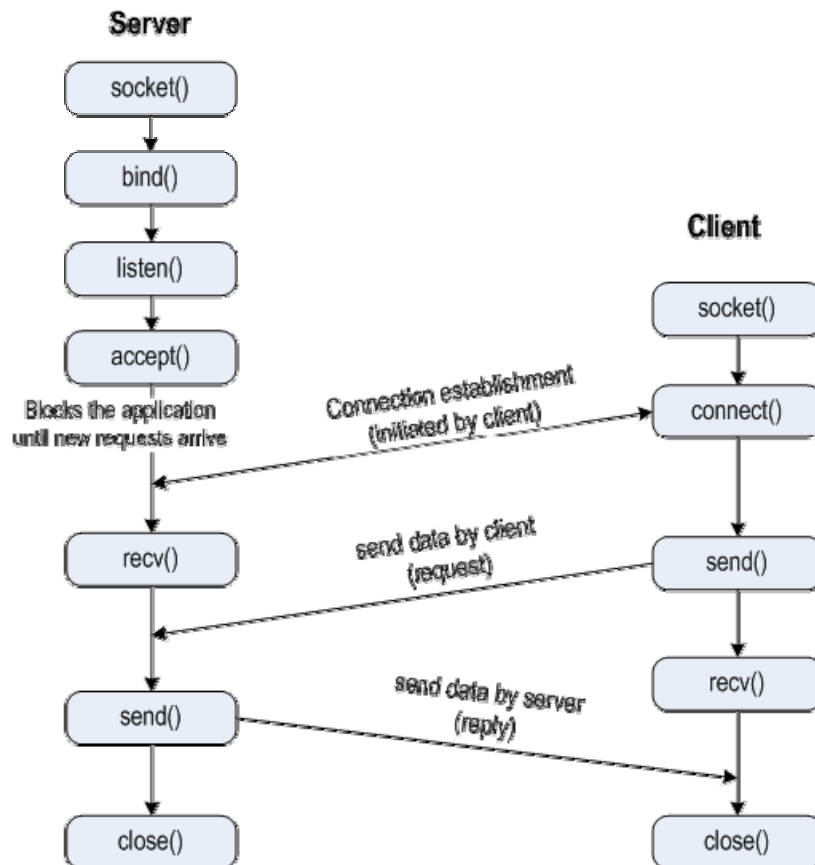


Figure 6.10 Socket functions for a connection-oriented protocol

The boxes represent the **system calls** involved in enabling the communication. First, the server is started, then some time later a client is started which connects to the server. We assume that the client sends a request to the server, the server processes the request, and then sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection. You should try to understand from the figure about the interrelationship of these calls and the chronological order of usage. For instance, the server must have executed the `accept()` first, i.e. before a client can use `connect()` to initiate a connection successfully. Also, `send()` and `recv()` should work as a pair to pass data from one side to another.

Connectionless mode

In contrast to the connection-oriented mode, the connectionless mode, as its name implies, does not require a logical connection to be established before communications can take place between clients and servers. It uses UDP as the underlying transport service. UDP is a connectionless, unreliable, datagram protocol, but there are instances in which it makes sense to use UDP instead of TCP. Some popular applications built using UDP include DNS, NFS, and SNMP, for example.

A datagram is a self-contained message. Sending a datagram is analogous to mailing someone a letter. You can mail many letters, but you can't guarantee their order of delivery, and some might get lost along the way. Each letter contains a recipient's address, making it independent from all the others. Each letter can even go to different recipients.

Datagram sockets (SOCK_DGRAM)

Figure 6.11 shows a typical scenario in which datagram sockets are used to carry out communications between a client and a server over the connectionless UDP service.

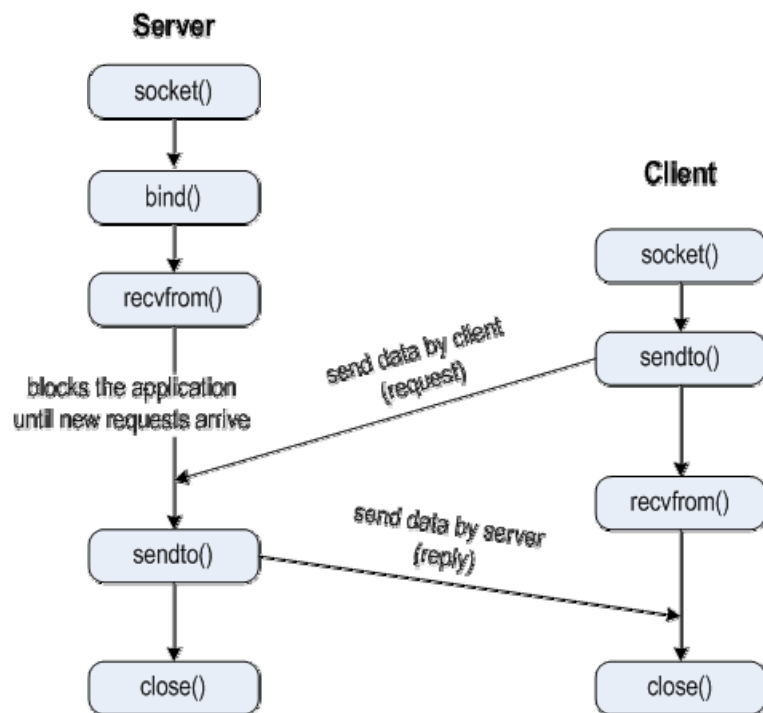


Figure 6.11 Socket functions for connectionless mode of communication between client/server

As you can see, the client does not establish a connection with the server. Instead, the client just sends a datagram to the server using `sendto()` (this is described in the next section), which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just invokes `recvfrom()`, which waits until some client's data arrives.

`recvfrom()` returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

The following reading provides further information on these modes.

Reading 6.2

Stevens, W R, Fenner, B, and Rudoff, A M (2003) *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd edn, Boston: Addison-Wesley, Sections 4.1 and 8.1.

Self-test 6.3

- 1 Describe the steps needed to create a connection-oriented server application using the BSD socket interface.
 - 2 Which two socket types are most commonly used in today's Internet communications, following the client/model model?
 - 3 What is the system call to create a stream socket in the Internet domain?
 - 4 What is the system call to create a datagram socket in the Internet domain?
-

Socket programming

In this unit's previous sections we introduced the big picture of the client/server computing model, and the role of socket programming within the model. We also discussed some key attributes of the socket interface under the two main modes of communication under the Internet domain (i.e. connection-oriented and connectionless).

We are now going to take a step further to see how to use a socket for programming work. In the following sections, you will first learn some data structures to be used in socket programming, and then get down into the details of the essential system calls involved in socket communications.

Data structures

Generic socket address structure

As you learned earlier, the BSD socket interface provides generalized functions that support different kinds of communications. The data structure of a primitive socket is fairly generic; it is defined in the `<sys/socket.h>` header and set out below:

```
struct sockaddr {
    sa_family_t  sa_family;    /* address family: AF_XXX value */
    char         sa_data[14]; /* protocol-specific address */
    ...
};
```

Different communication domains have their own family of addresses with their own specific address structures, e.g. IPv4 addresses, IPv6 addresses, UNIX local addresses, etc. The cast technique is used to solve the problem of how to use a single set of generic socket system calls (e.g. `bind()`, `connect()`, `send()`, `recv()`, etc.) to serve different communication domains, each of which uses its own address structure.

For example, the prototype of the `bind()` functions looks like this:

```
int bind(int, struct sockaddr *, socklen_t);
```

When calling the `bind()` under the Internet domain using IPv4 address, we have to do it like this:

```
struct sockaddr_in serv;    /* IPv4 socket address structure */

bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

'`sockaddr_in`' is an IPv4-specific address structure. We cast (or retrofit) it with the structure '`sockaddr`' when calling the `bind()` using the cast operator `(struct sockaddr *)`.

Socket addresses in the Internet domain

In the Internet domain, an IP address is used to identify a particular *host* on the Internet. Within a particular host, however, there can be many different processes running. How can we identify a particular *process* to communicate with? The answer is the port number, which is like a mailbox number uniquely assigned to each process within a host. It allows multiple addresses on the same host to identify themselves unambiguously.

A *socket address* of the Internet domain consist of two parts — a host IP address and a port number:

```
socket address { Host IP address : Port number }
```

A socket address allows us to uniquely identify a particular process running on the Internet. For instance, 202.40.157.146 : 80 represents the Web server application (identified by port 80) running on the ucourse2.ouhk.edu.hk host (identified by the IP address 202.40.157.146).

The value of a port number ranges from 1 to 65,535. However, 1 — 1023 is the range for the **well-known ports**. Programmers should only use those port numbers beyond this range. The following table lists some well-known ports and their associated applications.

Table 6.3 A list of well-known ports in TCP/IP

Port Numbers	tcp/udp	Services	Explanations
7	tcp/udp	echo	For use in the ‘ping’ command
20	tcp	ftp-data	File transfer protocol (FTP) — data transmission
21	tcp	ftp-control	File transfer protocol (FTP) — control
22	tcp/udp	ssh, scp, sftp	Secure Shell (SSH) —for secure remote logins, file transfers (SCP, SFTP) — for secure file transfer
23	tcp	telnet	Telnet protocol—unencrypted text communications
53	tcp/udp	dns	Domain Name System (DNS) for hostname resolution
80	tcp	http	Hypertext Transfer Protocol (HTTP) for Worldwide Web (WWW) communications

Each socket should bear a socket address. After creating a socket with `socket()`, we can bind the created socket to a particular socket address. The structure `sockaddr_in` defines the socket address in the Internet domain (IPv4), which is shown below:

```
struct sockaddr_in {
    short sin_family;          /* AF_INET */
    u_short sin_port;         /* 16-bit port number */
    /* network byte ordered */
    struct in_addr sin_addr; /* 32-bit netid:hostid */
    /* network byte ordered */
    char sin_zero[8];         /* unused */
};

<netinet/in.h>
struct in_addr {
    u_long s_addr;            /* 32-bit netid:hostid */
    /* network byte ordered */
};
```

Figure 6.12 The address structure `sockaddr_in` for the Internet domain

For example, if you declare `my_addr` with `sockaddr_in`, the `my_addr.sin_family` will store the domain.

Meanwhile, the IP address and port number are saved in `my_addr.sin_addr.s_addr` and `my_addr.sin_port` respectively. A simple initialization of the `my_addr` is given below:

```
my_addr.sin_family = AF_INET; /* host byte order */
my_addr.sin_port = htons(3456); /* short, network byte order */
my_addr.sin_addr.s_addr = inet_addr("192.123.5.10");
bzero(&(my_addr.sin_zero), 8); /* zero the rest of the struct*/
```

Figure 6.13 An example to fill up the structure `sockaddr_in`

The above lines introduce two new system calls, `htons()` and `inet_addr()`. The `htons()` call converts the port number 3456 into a number in the format of network byte order. The use of `inet_addr()` is to change an IP address in the textual format (i.e. a character string) with dot format into a 32-bit number with proper network byte order.

With IPv4, the wildcard address is specified by the constant `INADDR_ANY`, whose value is normally 0. This tells the kernel to choose the IP address of the host for use in communication, just as in the code shown below.

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);    /* wildcard */
```

The byte order function calls are explored further in next section.

Byte ordering functions

Referring to the structure `sockaddr_in`, the type of its member `sin_port` is a 16-bit port number (i.e. '3456' or '0x0D80' in the example), which should conform to the format of the '*network byte order*' to ensure its portability across different systems. The same requirement applies to other integers involved in socket programming. The Internet protocols use big-endian byte ordering for these multi-byte integers. On the other hand, different systems store integers in their own '*host byte order*', as shown in Figure 6.14.

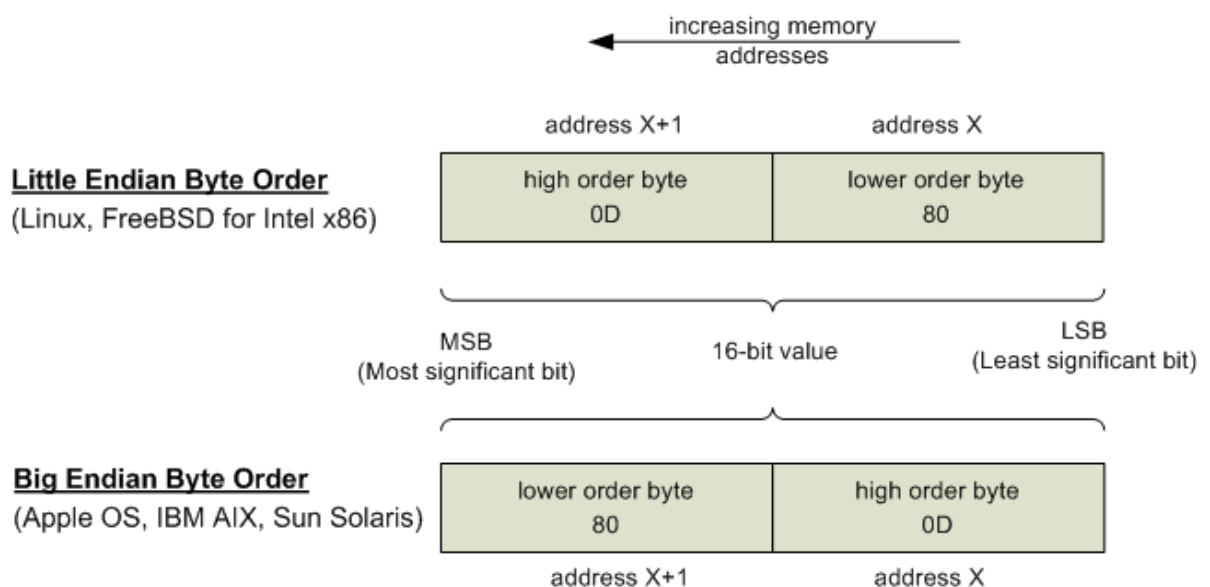


Figure 6.14 Representation of the port number 3456 (or 0x0D80) in different byte orders

The terms *little-endian* and *big-endian* indicate which end of the multi-byte value — i.e. the little end or the big end — is stored at the starting address of the value. To ensure that their code can run properly on different systems, programmers should make use of the appropriate conversion functions to convert the multi-byte integers used in the socket address from host byte order to network byte order when using the socket system calls. Conversely, they should convert the multi-byte integers returned from those calls into the host byte order before performing any further processing on them.

The following are the typical byte ordering routines:

<code>htonl()</code>	converts 32-bit host value into network byte order
<code>htons()</code>	converts 16-bit host value into network byte order
<code>ntohl()</code>	converts 32-bit network byte order value into host order
<code>ntohls()</code>	converts 16-bit network byte order value into host order
<code>inet_aton()</code>	converts an IP address from a dotted-decimal string (e.g., '202.40.157.146') to its 32-bit binary network byte ordered value
<code>inet_addr()</code>	provides functionality similar to <code>inet_aton()</code>
<code>inet_ntoa()</code>	converts an IP address from 32-bit binary network byte ordered value into a dotted-decimal string

Figure 6.15 Byte ordering functions provided by the system

In the names of these functions, `h` stands for host, `n` stands for network, `s` stands for short or 16-bit value, and `l` stands for long or 32-bit value.

In the sections that follow, you will see in the sample code how these calls are used in conjunction with other system calls to achieve the socket communications.

You should now complete the following reading from your textbook, and then undertake the exercises that follow.

Reading 6.3

Stevens, W R, Fenner, B, and Rudoff, A M (2003) *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd edn, Boston: Addison-Wesley, Sections 3.1–3.6.

Self-test 6.4

In Figure 6.13, what does the system call `bzero()` do?

Activity 6.2

Write a program to determine your system's byte ordering.

TCP socket programming (connection-oriented)

You will now learn how to use the elementary socket system calls to write a connection-oriented network application. Figure 6.16 below is a flow chart showing the sequence in using the socket system calls in server operation, with the corresponding pseudo code printed on the right hand side.

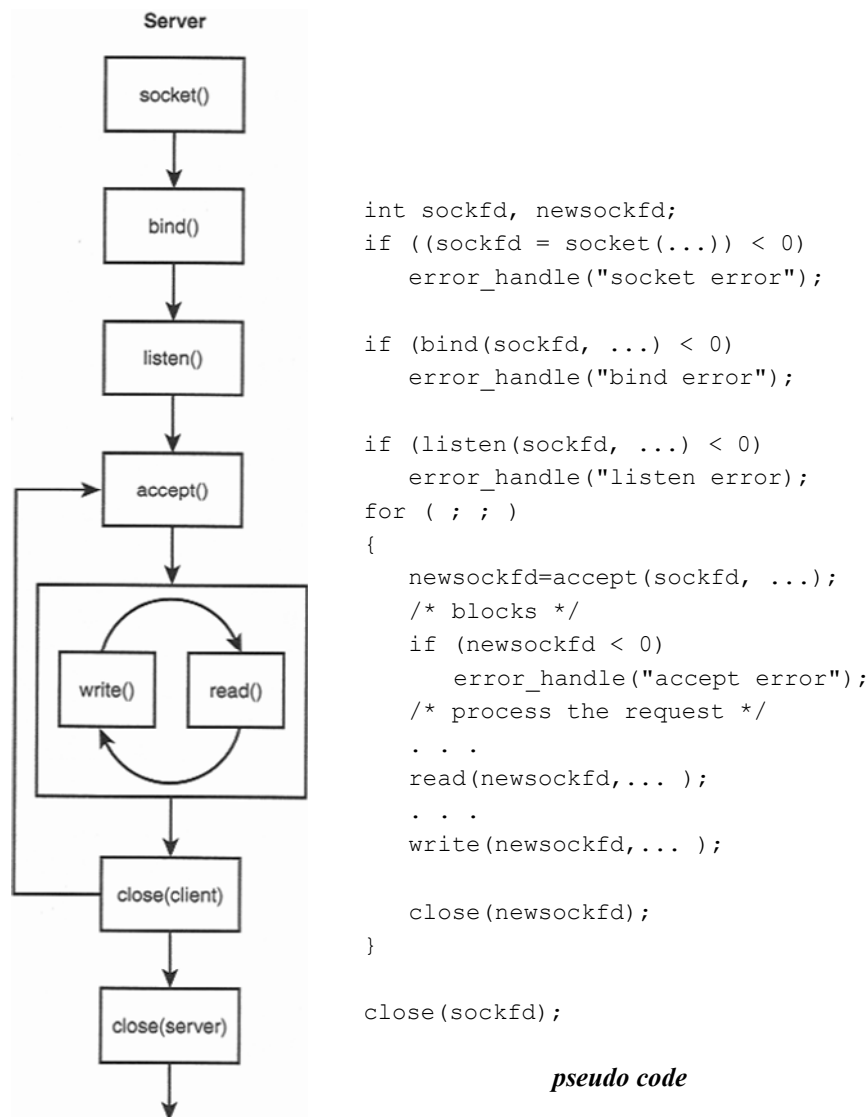


Figure 6.16 The flowchart and pseudo code for servers in connection-oriented mode

TCP server programming

The following are the system calls involved in server-side programming.

socket ()	<pre>#include <sys/types.h> #include <sys/socket.h> int socket(int domain, int type, int protocol);</pre>
------------------	---

`socket ()` creates a socket, much like establishing a connection using a telephone, and returns a socket descriptor. This call requests the system to create a socket in the specified `domain` and `type`. The `domain` should be `AF_INET` (Internet protocols), as far as this unit is concerned, while the `type` is either `SOCK_STREAM` or `SOCK_DGRAM`. Please refer to the earlier section ‘Connection mode of socket communication’ for more about the communication domain and socket type. For protocol, use the value ‘0’ to let the system determine the appropriate protocol.

bind()	<pre>#include <sys/types.h> #include <sys/socket.h> int bind(int sockfd, struct sockaddr *myaddr, int addrlen);</pre>
---------------	---

`bind()` associates the created socket to a particular socket address, whose structure should be confined to the address family that the socket belongs to. For the Internet domain, the address should consist of an IP address and a port number. This is similar to assigning a phone number to a socket.

The first argument is a socket descriptor that was returned by `socket ()`. The second argument is a pointer to a protocol-specific address, and the third argument is the size of this address structure. Please refer to the earlier section ‘Data structures’ for more about these address structures.

Each server application must use `bind ()` to bind themselves to a specific socket address so that the clients can raise requests against it. For instance, a browser client can already connect to the OUHK’s website through its specific address { 202.40.220.3: 80}. Binding to a specific address is important to a server, and it can’t change often. As an analogy, if a shop changes its address often, how can customers patronize it regularly?

listen ()	<pre>#include <sys/socket.h> int listen(int sockfd, int backlog);</pre>
------------------	---

`listen ()` prepares the socket to handle incoming requests from clients for connection. It converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket and allocate data buffers. This is similar to plugging a telephone into a socket.

It's usually executed after both `socket()` and `bind()`, and immediately before `accept()`. The `sockfd` argument is a file descriptor returned by `socket()`. The `backlog` argument specifies the amount of the buffer (or queue) for the server system holding connection requests so that the `accept()` all can process them one by one. Most systems silently limit this number to about 20; you can probably get away with setting it to 5 or 10.

accept()	<pre>#include <sys/types.h> #include <sys/socket.h> int accept(int sockfd, struct sockaddr *peer, int *addrlen);</pre>
-----------------	--

`accept()` gets the server application into standby mode and waits for any incoming requests from clients. It takes the first network connection request from the queue. If the queue is empty, the application goes to sleep (i.e. blocking status) and waits until a connection request arrives.

The `sockfd` argument is a socket descriptor returned by the `socket()`. You must have previously called `listen()` before calling `accept()`, which fills the socket address of the calling client application into the `peer` argument in return. `peer` is a pointer to a data structure `sockaddr`, which we explained in the section 'Data structures'. The `addrlen` argument relates to the size of the buffer (in number of bytes) to be used by the `peer` structure to hold the client socket address. `addrlen` is called a value-result argument: The caller sets its value before the system call, and the system call stores a result in the variable. Often these value-result arguments are integers that the caller sets to the size of a buffer, and the system call changes this value on the return to the actual amount of data stored in the buffer. For this system call, the caller sets `addrlen` to the size of the `sockaddr` structure, whose address is passed as the `peer` argument. On return, `addrlen` contains the actual number of bytes that the system call stores in the `peer` argument.

If the `accept()` is successful, it returns a new socket descriptor to the server application. Subsequently, the application will use it for sending/receiving data with the client application. On the other hand, it will continue to use the old socket descriptor to listen for incoming connection request messages.

read() write()	<pre>#include <sys/types.h> #include <sys/uio.h> #include <unistd.h> int read(int sockfd, char *buff, int size); int write(int sockfd, char *buff, int size);</pre>
---------------------------------	---

Once a request for connection is accepted by the server application, the two parties can start transmitting data between each other using `read()` and `send()`. Alternatively, `recv()` and `write()` can also be used for this type of connection-oriented communication, which will be covered in the next section.

Normally, the argument `sockfd` should be the socket descriptor returned by `accept()` (in blocking mode). `size` is the number of bytes read or written from the file associated with `sockfd` into the buffer pointed to by `buff`. Note that `write()` blocks until data are transferred to the socket buffers.

send()	<code>#include <sys/types.h></code>
recv()	<code>#include <sys/socket.h></code>
	<code>int send(int sockfd, char *buff, int size, int flag);</code>
	<code>int recv(int sockfd, char *buff, int size, int flag);</code>

These system calls are similar to the standard `read()` and `write()` and have one extra argument: `flag`, for specifying additional options to inspect or impose finer control on the communications. For instance, it can be used to glance at the data or exchange out-of-band (urgent) data between the server and client.

If the `flag` argument is set to 0, `send()` and `recv()` behave exactly as `write()` and `read()`. In fact, we will use this setting for the programs used in this unit. If it is not 0, it can be the logical or of the following values:

MSG_OOB Sends or receives out-of-band (urgent) data on sockets that support this notion.

MSG_PEEK (for `recv()` and `recvfrom()` only) This flag lets us look at the data that is available to be read, without having the system discard the data after the `recv()` or `recvfrom()` returns.

MSG_DONTROUTE This flag tells the kernel that the destination is on a locally attached network so it should not perform a lookup of the routing table; e.g. the destination is on the other end of a point-to-point link, or is on a shared network. Only diagnostic or routing programs use it.

close()	<code>int close(int fd);</code>
----------------	---------------------------------

`close()` terminates the connection upon completion of the communication. It accepts both file and socket descriptors, and closes the connection. This will prevent any more reads and writes to the socket. Anyone attempting to read or write the socket on the remote end will receive an error.

TCP client programming

Figure 6.17 below is a flow chart setting out the sequence for using socket system calls in a client operation.

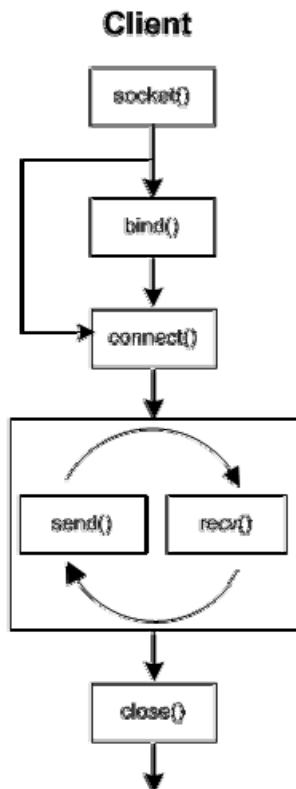


Figure 6.17 The flowchart for client side operation in connection-oriented mode

The client application should begin with creating a socket using the `socket()`. Normally, a client does not use `bind()` to bind an IP address to its socket. The kernel chooses the source IP address, based on the outgoing interface that is used, and an ephemeral port when the socket is connected. After that, the client can make connection request to the server using `connect()`.

Once connected, data exchange can take place using `read()` and `write()`, similar to the situation that occurs at the server side. Upon finishing all transmissions, we use `close()` to close the connection. `connect()` is explained below, while you can refer to the last section of this topic for the information about other related system calls.

connect()	<pre>#include <sys/types.h> #include <sys/socket.h> int connect(int sockfd, struct sockaddr *servaddr, int addrlen);</pre>
------------------	--

`connect()` is used by the client process to make a request for connection to the server with the server's address specified in the request.

`sockfd` is a socket descriptor returned by the `socket()`. You can specify the remote server's address in the second and third argument, which is a pointer to a socket address `servaddr` and a pointer to its size, respectively. These are types of value-result arguments. In connection-oriented applications the client program typically executes `connect()`, which results in a connection between the local and remote systems.

Reading 6.4

Stevens, W R, Fenner, B, and Rudoff, A M (2003) *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd edn, Boston: Addison-Wesley, Sections 4.1–4.6 and 4.9.

Self-test 6.5

- 1 Why is `bind()` optional for client applications, but mandatory for server applications?
 - 2 What is the difference between the system calls `bind()` and `connect()`?
-

Activity 6.3

- 1 Write a snippet of program code for a server application that can accept requests for connection from remote clients via its server port 7890. Your application has to reserve the port number 7890, and the maximum number of the pending processes in the listening queue is 5.
- 2 Write a snippet of program code for a client application that can create a socket file descriptor and use it to connect to '202.40.157.146' on port '22'.

(*Hint:* In order to convert an IP address into a 32-bit integer with network byte order, the `inet_addr()` call should be used. This routine can interpret character strings representing numbers expressed in the Internet dot-decimal '.' notation, returning numbers suitable for use as Internet addresses.)

Lab 1: Implement a connection-oriented (TCP) application

Let's now work through a lab in which we will implement an application that functions as a simple client/server time machine.

Step 1

Login to the `labsupport.no-ip.org` server. Contact your tutor if you don't have your account information.

Step 2

Use an editor (e.g. `vi`, `pico`, etc.) to create the server program (e.g. `server.c`) at the server based on the program code provided below. The server creates a socket and binds it to port 5999 in the code. However, *you should change the port number to another value in your implementation*. Otherwise you will probably run into problems if any other student has already bound port 5999 in their program. Remember that a port number can be bound to a single application at any one time. After the binding, the server will stop and wait (`accept()`) for the client request. Once a client request arrives, the server process will keep on going and create a new socket, `client_sockfd`, to handle the communication with the client process. The server socket `sockfd` will then be free to accept other new incoming requests.

```
/* TCP time server with the port number 5999 */

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

#define SIZE    1024
#define TIME_PORT 5999
/* the port users will be connecting to, change it to another value
greater than 1024 */

#define BACKLOG 5 /* how many pending connections queue will hold */

int main(int argc, char *argv[])
{
    char buf[SIZE];
    int sockfd, client_sockfd;

    int nread, len;
    struct sockaddr_in serv_addr; /* my address information */
    struct sockaddr_in client_addr; /* connector address information */
    time_t t;
```

```
/* create endpoint */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror(NULL);
    exit(2);
}

/* bind address */
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
serv_addr.sin_port = htons(TIME_PORT); /* short, network byte order */
bzero(&(serv_addr.sin_zero), 8); /* zero the rest of the struct */
if (bind(sockfd, (struct sockaddr*) &serv_addr, sizeof(serv_addr)) < 0)
{
    perror(NULL);
    exit(3);
}
/* specify queue */
listen(sockfd, BACKLOG);

/* wait and accept connection */
for (;;)
{
    len = sizeof(client_addr);
    client_sockfd = accept(sockfd, (struct sockaddr*) &client_addr, &len);
    if (client_sockfd == -1)
    {
        perror(NULL);
        continue;
    }

    /* perform server specific tasks */
    time(&t); sprintf(buf, "%s", asctime(localtime(&t)));
    len = strlen(buf) + 1;

    write(client_sockfd, buf, len);
    close(client_sockfd);
}
}
```

Step 3

Compile the program (server.c) by executing the following command:

```
gcc -o server.o server.c
```

Debug the program if any error messages appear.

Step 4

Execute the compiled program server.o by:

```
./server.o
```

Step 5

The client process connects to the server host to request the return of the timestamp from the server. Similarly, login to the server `ucourse2.ouhk.edu.hk` and create the client program (e.g. `client.c`) using the program code provided below.

The client should have created a socket and connected to the server's socket, named `serv_addr`. After the setup of the connection, data transmission can start, and the client can then read the timestamp from the server. From the client's point of view, the server IP address and the corresponding port number are needed to set up the connection. In this example, the port number of the timeserver is defined at 5999. Make sure that the port number used in the server address structure of the client program is the same as that used in the server program. Otherwise, they won't be able to connect to each other.

```
/* TCP client that finds the time from a server */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#define SIZE 1024
#define TIME_PORT 5999
/* the port users will be connecting to the server, make sure it is the
same as the port used by the server.c*/

int main(int argc, char *argv[])
{
    char buf[SIZE];
    int sockfd;
    int nread;
    struct sockaddr_in serv_addr; /* my address information */

    /* check if correct usage */
    if (argc != 2)
    {
        fprintf(stderr, "usage: %s IPaddrs\n", argv[0]);
        exit(1);
    }

    /* create endpoint */
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror(NULL);
        exit(2);
    }

    /* no need to bind a socket */
    /* connect to a server */
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(argv[1]);
    serv_addr.sin_port = htons(TIME_PORT); /* short, network byte order */
}
```

```

bzero(&(serv_addr.sin_zero), 8); /* zero the rest of the struct */
if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr))<0)
{
    perror(NULL);
    exit(3);
}

/* perform client specific tasks */
nread = read(sockfd, buf, SIZE);
write(2, buf, nread);
close(sockfd);
exit(0);
}

```

Step 6

Compile the program (client.c) by executing the following command:

```
gcc -o client.o client.c
```

Debug the program if any error messages appear.

Step 7

You have to input the server's IP as the first argument when running the client (client.o). Make sure the server program is running on the labsupport server (210.17.156.179), then at command prompt, run `./client 210.17.156.179`.

The result of this lab, including the server and client programs, is shown in Figure 6.18 below.

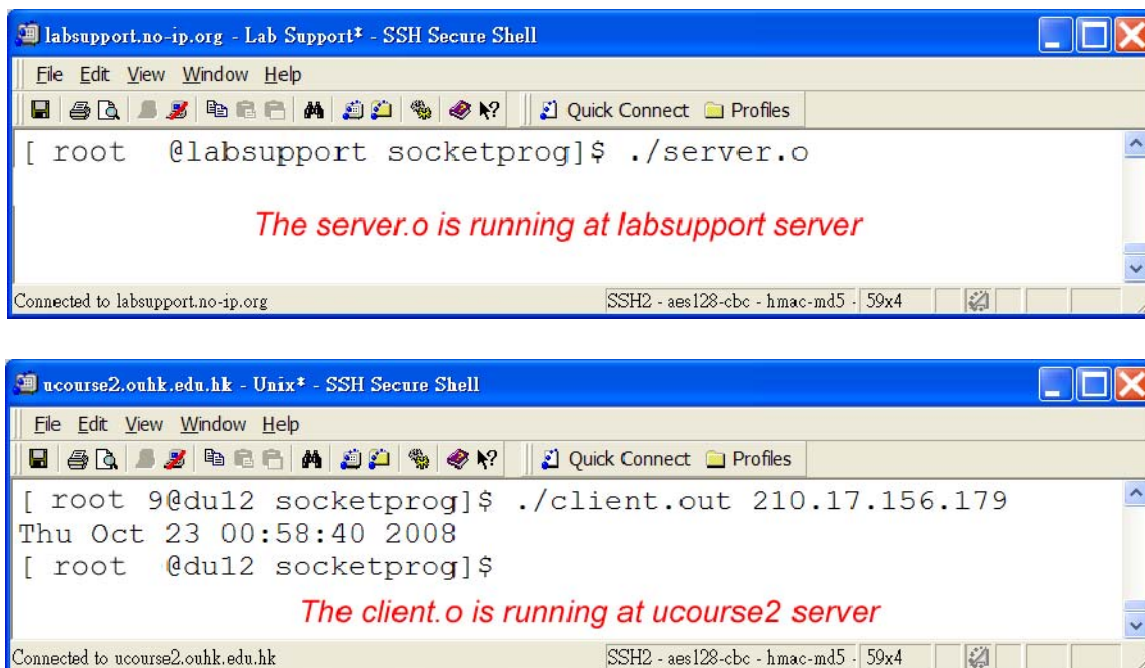


Figure 6.18 The result of Lab 1

UDP socket programming (connectionless)

It is now time to introduce you to UDP socket programming. UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream service provided by TCP that we covered in the last topic. There are instances, however, when it makes sense to use UDP instead of TCP. For real-time applications, an efficient protocol is more useful than a reliable one because it's better to have lost packets discarded from a voice stream than to have them retransmitted, since there is no way to insert the retransmitted packets back into a part of the stream that has already been played.

UDP server programming

The figure below shows the system calls involved in connectionless socket programs.

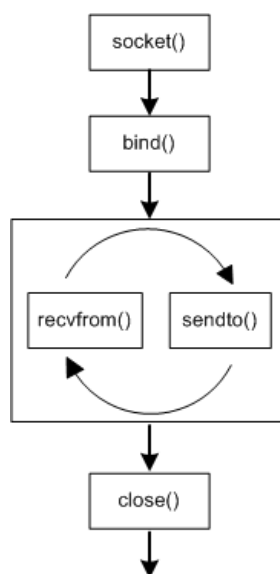


Figure 6.19 The flowchart for server operation in connectionless mode

As you can see, the connectionless server program creates and assigns a name to a socket using `socket()` and `bind()`, respectively. However, since no connection is established between the client and server, it is possible to skip `listen()`, `accept()` and `connect()`. After binding a socket, the server process waits for the arrival of data from the client through `recvfrom()`. When client data is received, the server process may send data back using `sendto()`. `close()` is used to prevent any further data transmission from a socket.

UDP client programming

A similar mechanism is involved in the connectionless client program, which is shown in the figure below.

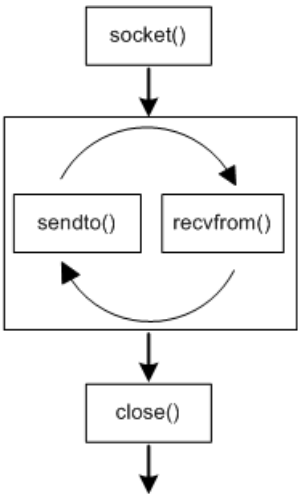


Figure 6.20 The flowchart for client operation in connectionless mode

As you can see, the client just sends a datagram to the server using the `sendto()` function, which requires the address of the destination (the server) as an argument and receives data from the server using `recvfrom()`.

sendto() recvfrom()	<pre>#include <sys/types.h> #include <sys/socket.h> #int sendto(int sockfd, char *buff, int size, int flag, #struct sockaddr *to, int addrlen); int recvfrom(int sockfd, char *buff, int size, int flag, struct sockaddr *from, int *addrlen);</pre>
--------------------------------------	--

The `sockfd` argument is a file descriptor in a socket file. The `buff` argument contains the data you want to send or receive, and `addrlen` is the number of bytes in the data. The `to` argument in the `sendto()` system call contains the destination address of the message (datagram) sent to the network. You must use `sendto()` if a datagram transport provider is being used and you want to send datagrams to several different sockets. Because you specify the destination address on every message, the `sendto` routine lets you vary the destination on every datagram.

The `recvfrom()` routine can be used on both the connection and connectionless oriented modes. The `from` argument in the `recvfrom()` system call contains the source address of the message received from the network. After the `recvfrom()` routine fills the `from` argument with the address of the socket that sent the data, it resets the `addrlen` argument to contain the number of bytes in the address.

Reading 6.5

Stevens, W R, Fenner, B, and Rudoff, A M (2003) *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd edn, Boston: Addison-Wesley, Section 8.2.

Activity 6.4

- 1 Can we use the `connect()` system call at a client to carry out connectionless (UDP) mode of communication?
 - 2 Apart from using `close()`, we can also end a connection using the system call `shutdown()`. Please find out from the Internet how to use `shutdown()`.
-

Self-test 6.6

- 1 What is the difference between the `sendto()` and `send()` system calls?
 - 2 What is the difference between the `recvfrom()` and `recv()` system calls?
-

Lab 2: Implement a connectionless (UDP) application

To summarize the system calls for the connectionless communication described above, a simple program example is described in this section. This application requires the user in the client process to input the user's name. The user's name will then be sent to the server process and displayed to the terminal. *The server IP address is 210.17.156.179, which is the labsupport.no-ip.org server.* In order to execute the process, you have to run the server process first. The server process then waits for the client connection and data. After executing the server process, you can start to run the client process.

Step 1

Similar to the last lab, login to the server labsupport.no-ip.org first.

Step 2

Use an editor (e.g. vi, pico, etc.) to create the server program (echo_server.c) based on the program code provided below:

```
/* Server Process - echo_server.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define SERVER_PORT 6999          /* a number > 5000 */
main()
{
    int sockid, nread, addrlen;
    struct sockaddr_in my_addr, client_addr;
    char msg[56];
    printf("Server: creating socket\n");
    if ( (sockid = socket (AF_INET, SOCK_DGRAM, 0)) < 0 )
    {
        printf("Server: socket error:%d\n",errno);
        exit(0);
    }
    printf("Server: binding my local socket\n");
    bzero((char *) &my_addr, sizeof(my_addr));
    my_addr.sin_family = AF_INET;
    my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    my_addr.sin_port = htons(SERVER_PORT);
    if ( (bind(sockid, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0) )
    {
        printf("Server: bind fail: %d\n",errno);
        exit(0);
    }
    printf("Server: starting blocking message read\n");
    addrlen = sizeof(client_addr);
    nread = recvfrom(sockid, msg, 56, 0, (struct sockaddr *) &client_addr,
        &addrlen);

    if (nread > 0) printf("Welcome. Your name is: %.54s\n", msg);
    close(sockid);
}
```

Step 3

Compile the program (echo_server.c) by executing the following command:

```
gcc -o echo_server.o echo_server.c
```

Debug the program if any error messages appear.

Step 4

Execute the compiled program server.o by:

```
./echo_server.o
```

Step 5

Login to the server ucourse.ouhk.edu.hk as before, and then create the client program (echo_cilent.c) using the program code below:

```
/* Client Process*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define SERVER_PORT_ID 6999
#define SERV_HOST_ADDR "210.17.156.179" /*labsupport.no-ip.org host*/
main()
{
    int sockid, retcode, length;
    struct sockaddr_in my_addr, server_addr;
    char yourname[55];
    printf("Client: creating socket\n");
    if ( (sockid = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        printf("Client: socket failed: %d\n", errno);
        exit(0);
    }
    printf("Client: creating addr structure for server\n");
    bzero((char *) &server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    server_addr.sin_port = htons(SERVER_PORT_ID);
    printf("Client: initializing message and sending\n");
    printf("What is your name ? ");
    gets(yourname);
    for (length=0; length < 56 && yourname[length]!='\0' ; length++);
    if(length > 55 )
    {
        printf("Name buffer overflow\n");
        exit(0);
    }
    retcode = sendto(sockid, yourname, length, 0, (struct sockaddr *)
    &server_addr, sizeof(server_addr));
    if (retcode <= -1)
    {
        printf("client: sendto failed: %d\n", errno);
        exit(0);
    }
    /* close socket */
    close(sockid);
}
```

Step 6

Compile the program (echo_client.c) by executing the following command:

```
gcc -o echo_client.o echo_client.c
```

Debug the program if any error messages appear.

Step 7

Run the client program using the command below:

```
./echo_client.o
```

The result of this lab, including the server and client programs, is shown below.

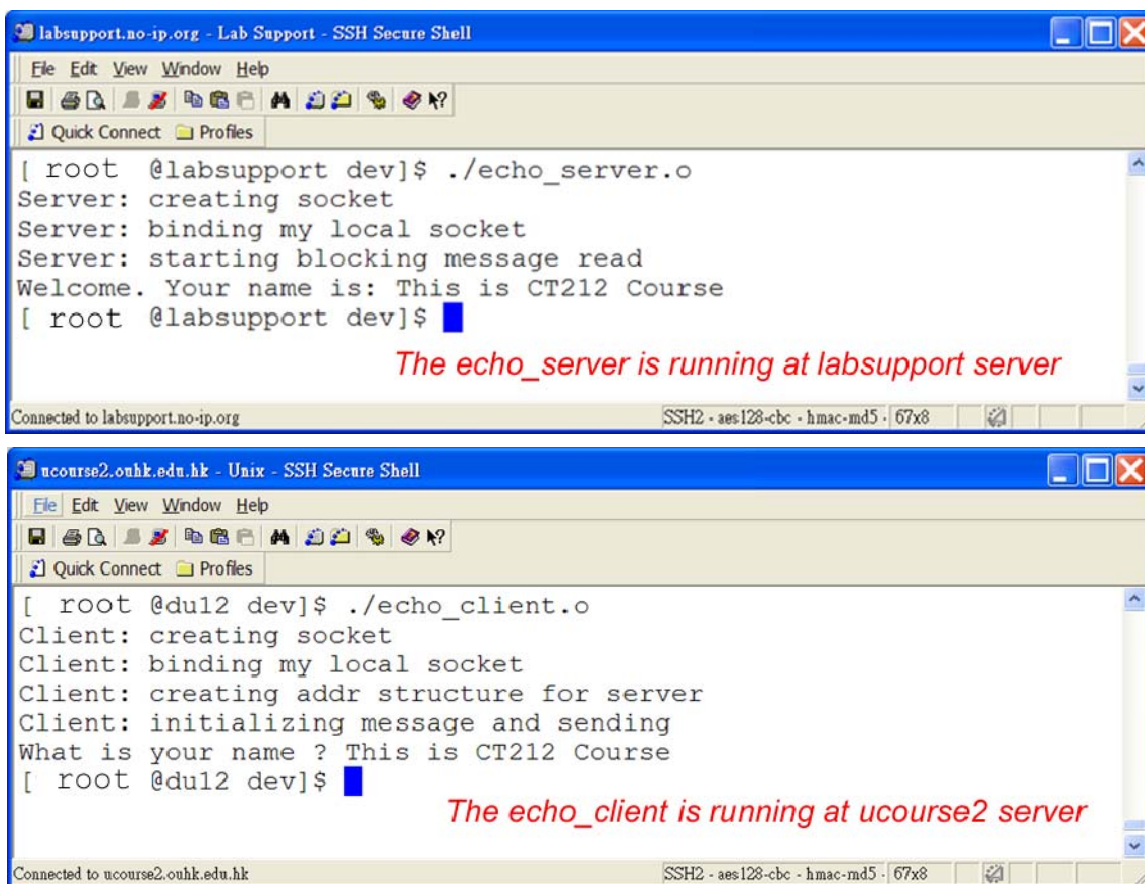


Figure 6.21 The result of the Lab 2

Remote procedure calls (RPCs)

A remote procedure call (RPC) facility can make it possible to implement client-server applications without needing to explicitly issue requests for communication services from within the application program. The idea behind this is that procedure calls are a well understood mechanism for transferring control and data from one procedure to another in a conventional computing application. All of these procedures reside in a single system, so it is therefore useful to extend the procedure call mechanism from a set of procedures in a single-computer environment to a set of procedures in a distributed, client-server environment. As a result, application programs can make use of the procedures hosted in remote systems, in addition to any local ones.

The remote procedure call model uses the application-oriented approach, which emphasizes the problem to be solved (instead of the communication needed). Using the remote procedure call model, a programmer first designs a conventional program to solve the problem, and then divides the program into pieces that run on two or more computers. Programmers should follow good design principles that make their code modular and maintainable. Also, they should focus on solving the problem that their application is aimed at addressing, while leaving it to the RPC to make the solution operable in distributed environments.

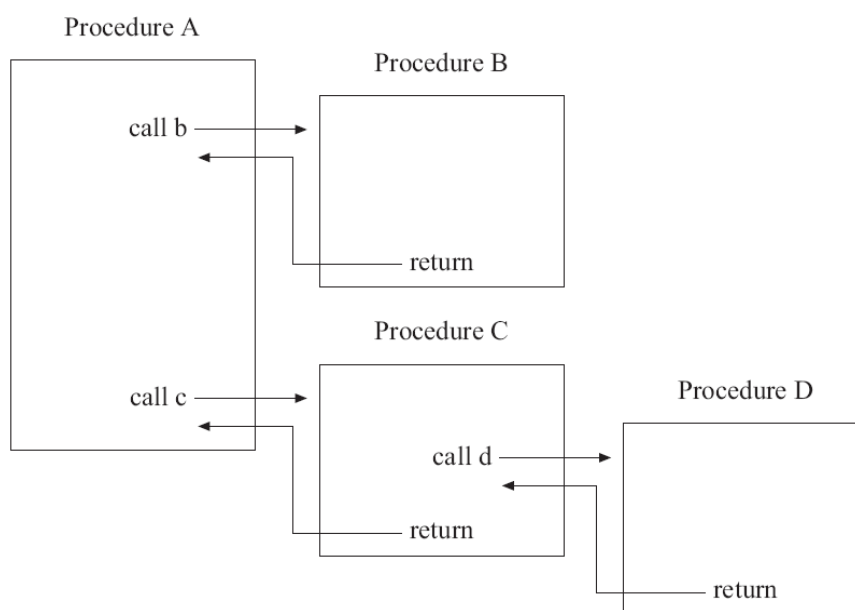


Figure 6.22 The basic concept behind a procedure call mechanism

In Figure 6.22, procedure A makes a procedure call (also known as a function call), possibly referencing some arguments, which passes control to procedure B. While procedure B executes, procedure A waits. When procedure B finishes its processing, it returns to the caller, which causes control to be passed to the statement immediately after the statement in procedure A that invokes procedure B. In most programming languages and operating system environments, function or procedure

calls can be nested to any desired level, as procedure A in Figure 6.22 invokes procedure C, which in turn invokes procedure D.

Challenges for RPC facilities

An RPC facility allows the procedure call mechanism to work in a client/server computing environment where the calling procedure (the client) and the called procedure (the server) reside on different host computers. A good RPC facility can keep itself as transparent as possible so that an application can use remote procedures as if they are using local ones. However, none of today's RPC facilities has achieved complete transparency. The challenges lie in the following three major areas:

- 1 Locating the called procedure — The RPC facility must provide a means for locating the called procedure in the network, and the calling procedure must provide the information required to do this. A directory service such as facility can be used to provide the means for locating remote procedures.
- 2 Passing arguments and results — With procedure call mechanisms implemented in traditional programming language environments, communication between the two procedures is based on a shared address space. Argument values are sometimes passed by reference, which means the calling procedure passes the called procedure pointers to the argument values, rather than the values themselves. When the two procedures reside on different host computers, however, there is no common address space. Therefore, an RPC facility must be able to handle the passing of arguments and results in both directions without the calling and called procedures having the benefit of a shared address space. There may also be differences in data formats from one type of host computer to another, including differences in integer octet ordering, in the number of bits used to represent binary integer values, and in floating point data representation. Such differences must be resolved by the RPC facility, and data format conversions take place when required.
- 3 Binding the called procedure to the calling procedure — In a conventional procedure call mechanism, many techniques can be used for binding. With early binding, a linking mechanism is used to construct a single program module containing both the calling and the called procedure. Both the calling and the called procedures are then loaded into storage together. With late binding, however, the procedure call mechanism implemented by the operating system may allow the called procedure to be dynamically loaded into computer storage at the time the procedure is actually invoked. With a remote procedure call facility, binding is even more complex because it involves finding the host computer on which the desired procedure resides and loading the procedure into memory as and when required.

RPC implementation

There are various RPC implementations. **SOAP** (Simple Object Access Protocol) is one that allow us to make calls to procedures that exist on a remote server. SOAP is one of latest manifestations of RPC, providing a key technology component for **Web services** implementation. It traces its ancestry back to the Sun RPC system, which is a common facility available in UNIX systems.

As you learned in last section, passing arguments and results between heterogeneous systems is challenging. The binary representation of values varies, depending on the operating system, hardware, and programming languages that have created the value. While the Sun RPC system provides a way to encode values in the standard binary format, representing data was still not straightforward. SOAP solves this problem by representing values in XML (Extensible Markup Language), which has gained wide acceptance by different platforms as a standard data specification language.

In this unit we use SOAP simply as a tool to elaborate on some of the concepts related to RPC. In the next section you will see a simple example of an RPC operation that is built upon SOAP. We will leave it to the next unit to go through SOAP more thoroughly.

An RPC example using Perl's SOAP::Lite module

There are many systems that can provide remote procedure call facilities in the TCP/IP environment. In this unit, we use the SOAP::Lite module implemented in the Perl environment as an example of the RPC facility. It is depicted in the following figure:

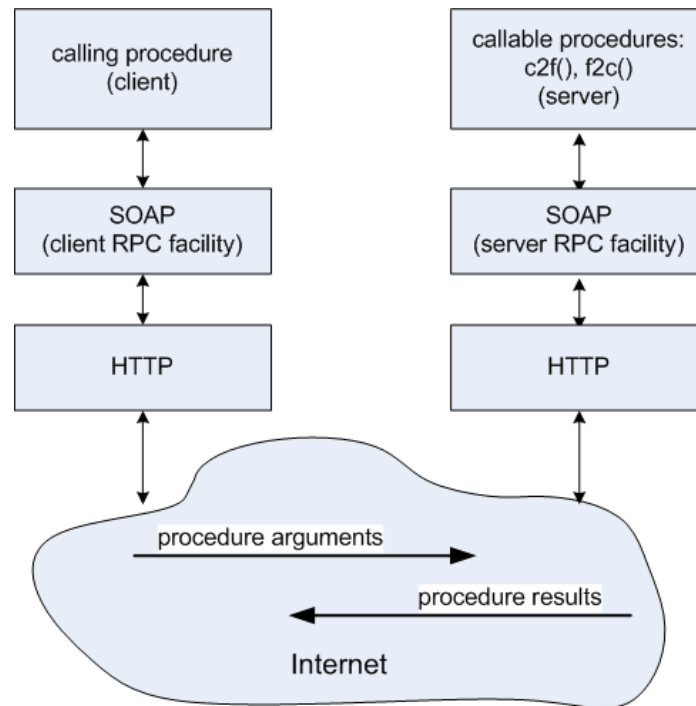


Figure 6.23 The functional model of a RPC facility

Callable procedure

At the server (labsupport.no-ip.org for this example), we use the SOAP::Lite module to make available two remotely callable procedures, namely `f2c()` and `c2f()`. These two functions, written in a Perl script file — `temper.cgi`, shown in the figure below — can translate from Fahrenheit to Celsius and vice versa.

```
#!/usr/bin/perl -w

use SOAP::Transport::HTTP;

SOAP::Transport::HTTP::CGI
    -> dispatch_to('Temperatures')
    -> handle;

package Temperatures;

sub f2c {
    my ($class, $f) = @_;
    return 5 / 9 * ($f - 32);
}

sub c2f {
    my ($class, $c) = @_;
    return 32 + $c * 9 / 5;
}
```

Figure 6.24 The source code of two remotely callable procedures: `f2c()` and `c2f()`

Calling procedure

At the client, we construct the calling procedure (as a Perl script `temp.pl`) to call the remote procedure `c2f()`. In this example, we also use `labsupport.no-ip.org` to run the client. As a matter of fact, any host that has `SOAP::Lite` installed can also run the client.

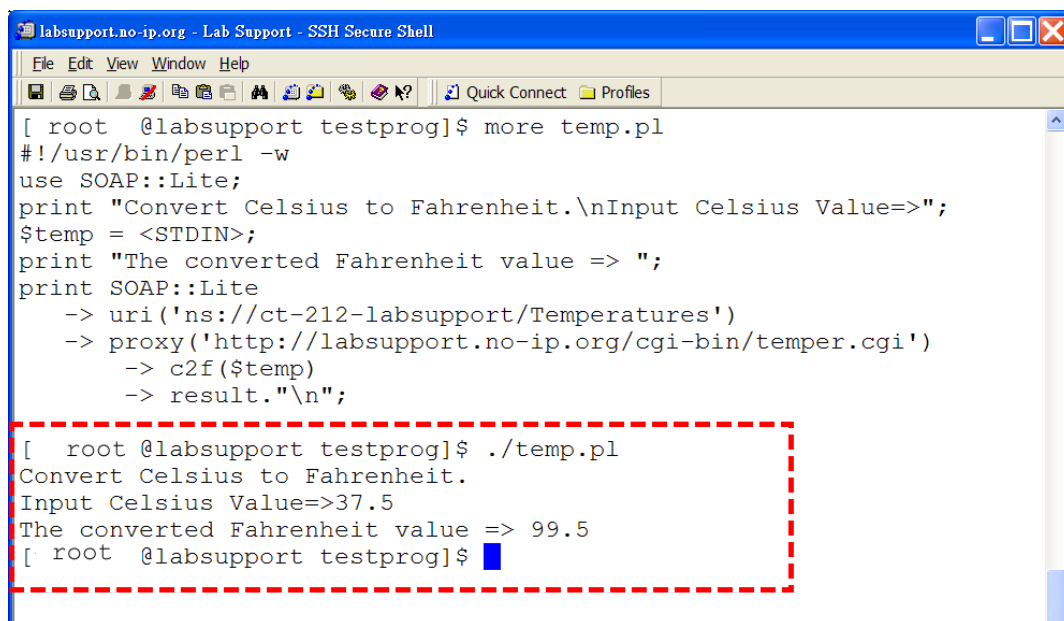
```
#!/usr/bin/perl -w
use SOAP::Lite;
print "Convert Celsius to Fahrenheit.\nInput Celsius Value=>";
$temp = <STDIN>;
print "The converted Fahrenheit value => ";
print SOAP::Lite
  -> uri('ns://ct-212-labsupport/Temperatures')
  -> proxy('http://labsupport.no-ip.org/cgi-bin/temper.cgi')
    -> c2f($temp)
    -> result."\n";
```

Figure 6.25 The source code of calling procedure at the client

The two methods — `proxy()` and `uri()` of `SOAP::Lite` — used above are explained below:

- `proxy()` is simply the address of the server to contact that provides the methods. You can use `http:`, `mailto:`, even `ftp:` URLs.
- `uri()` specifies the ‘namespace’, which is a convention of Web services. Each server can offer different services through the one `proxy()` URL. Each service has a unique URI-like *identifier*, which you specify through the `uri()` method.

Below is the result of the implementation of these programs.



```
labsupport.no-ip.org - Lab Support - SSH Secure Shell
File Edit View Window Help
[ root @labsupport testprog]$ more temp.pl
#!/usr/bin/perl -w
use SOAP::Lite;
print "Convert Celsius to Fahrenheit.\nInput Celsius Value=>";
$temp = <STDIN>;
print "The converted Fahrenheit value => ";
print SOAP::Lite
  -> uri('ns://ct-212-labsupport/Temperatures')
  -> proxy('http://labsupport.no-ip.org/cgi-bin/temper.cgi')
    -> c2f($temp)
    -> result."\n";

[ root @labsupport testprog]$ ./temp.pl
Convert Celsius to Fahrenheit.
Input Celsius Value=>37.5
The converted Fahrenheit value => 99.5
[ root @labsupport testprog]$
```

Figure 6.26 The results of the RPC programs

Do you now find RPC much simpler than you had expected? Work through the reading below which provides an introduction to SOAP (Simple Object Access Protocol) as a means to implement RPC over the Internet and be sure to try out the exercises that follow to consolidate your learning.

Reading 6.6 (online)

Kulchenko, P (2001) 'Quick Start with SOAP', *perl.com*:

<http://www.perl.com/pub/2001/01/soap.html>

Self-test 6.7

You have been introduced to the **DCOM** (Distributed Component Object Model) technology from Microsoft. How does it relate to RPC? And how does it compare to Web services technology?

Activity 6.5

Modify the code provided in Figure 6.25 so that the client can use the remote procedure to convert a value from Fahrenheit to Celsius.

Summary

In this unit you studied a number of fundamental concepts and issues surrounding network programming. This foundational study will help you move ahead with ease to tackle more advanced programming topics in coming units.

To start off, you reviewed some topics related to network programming, including the client/server model, UNIX processes, and interprocess communication (IPC). You saw how the main purpose of network programming is to enable two processes to communicate with each other over networks. These basic topics provided you with a good foundation for the rest of the unit's subject matter.

You then worked through the connection-oriented mode of communication over TCP. TCP is a reliable, connectional, and error-free protocol. A connection must be established before communication can take place. Another mode of communication is connectionless communication over UDP. UDP is unreliable, connectionless and non-error-free protocol. If you have not already done so, you should take time to finish the two purpose-built lab exercises, which helped you grasp these topics more effectively.

Finally you were introduced to the Remote Procedure Call (RPC). The RPC model uses the application-oriented approach so that programmers can focus on solving the problem that an application is created to address without bothering too much with networking details. In addition to studying the theories and concepts behind RPC, you had the chance to try out a sample program code based on the SOAP (Simple Object Access Protocol).

In next unit you will study more advanced network programming topics, including Web application development, Web 2.0, and network game design.

Suggested answers to self-tests and activities

Self-test 6.1

At a UNIX host (e.g. ucourse2.ouhk.edu.hk), run the command:

```
ps -A -o ppid,pid,pgid,user,ruser,comm
```

At a Linux host (e.g. labsupport.no-ip.org), run the command:

```
ps -eo ppid,pid,pgid,user,euser,comm
```

Self-test 6.2

Any of the following key differences is correct:

- Sockets have addresses associated with them, but files don't.
- Sockets only allows sequential data access, while files support both sequential and random access.
- Apart from `read()` and `write()`, sockets have some other system calls for data access e.g. `send()`, `recv()`, `sendto()`, `recvfrom()`, while files don't have them.

Self-test 6.3

- 1 To create a server application using the BSD interface, you can follow these steps:
 - Use `socket()` to create a new socket.
 - Use `bind()` to affix the server application to a particular socket address. This step identifies the server so that the client knows where to go.
 - Use `listen()` to get the system ready to handle new connection requests.
 - Use `accept()` to wait for and accept client connection requests.
 - Use `send()` and `recv()` to exchange data with the connected client.
 - Use `close()` to de-establish the connection and complete the transmission.

- 2 The following are the two most popular types of sockets used nowadays:
 - Stream socket — `SOCK_STREAM`
 - Datagram socket — `SOCK_DGRAM`
- 3 The following call can create a *stream* socket in the Internet domain:

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

- 4 The following call can create a *datagram* socket in the Internet domain:

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

Self-test 6.4

`bzero()` sets the specified number of bytes to 0 in the destination. We often use this function to initialize a socket address structure to 0. Its prototype is shown below:

```
#include <strings.h>
void bzero(void *dest, size_t nbytes);
```

Self-test 6.5

- 1 Servers should bind themselves to a specific port when they start. If a server does not do this, the kernel chooses an ephemeral port for the socket when `listen()` is called. It is rare for a TCP server to let the kernel choose an ephemeral port, since servers should make themselves available for clients contacting it via a fixed port. On the other hand, it is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port since they don't need to be contacted by others.
- 2 The `connect()` system call is used to specify the remote side of an address connection. The `bind()` system call only allows specification of a local address.

Self-test 6.6

- 1 The `sendto()` system call is the same as the `send()` system call, except that the calling process also specifies the address of the recipient's socket name.
- 2 The `recvfrom()` system call is the same as the `recv()` system call, except that the calling process also specifies the address of the sender's socket name.

Self-test 6.7

COM (Component Object Model) was based on language independence, interoperability, a strong focus on *reusable components*, and extensibility. DCOM (Distributed Component Object Model) took a step further to overcome the limitations of data and interface specifications. It set out to make remote *objects* as accessible as they were local. *While RPC dealt only in procedures, DCOM was designed to exist in a world of objects and method calls.* Unfortunately, DCOM has never really taken off outside the Microsoft world.

Web services escape being tied down to particular hardware or languages by using the Extensible Markup Language (XML) to represent data. There are XML parsers available for everything from embedded systems to supercomputers, and almost every conceivable programming language. Furthermore, to get away from the complexity of object brokers, sockets, and all kinds of connectivity hassles, Web services are built on top of the Hypertext Transfer Protocol (HTTP). HTTP is also ubiquitous, with Web servers available for almost every platform. These are some of the reasons why Web services can flourish.

Activity 6.1

```
#include <stdio.h>
main()
{
printf("UID=%d, EUID=%d, GID=%d and EGID=%d\n",
getuid(),
geteuid(), getgid(), getegid());
}
The result would be as follows:
[root@dul2 dev]$ ./act6.1.o
UID=2090, EUID=2090, GID=1002 and EGID=1002
[root@dul2 dev]$
```

Activity 6.2

The program below can show a system's byte order.

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

int main(void)
{
    uint32_t i;
    unsigned char *cp;

    i = 0x04030201;
    cp = (unsigned char *)&i;
    if (*cp == 1)
        printf("little-endian\n");
    else if (*cp == 4)
        printf("big-endian\n");
    else
        printf("who knows?\n");
    exit(0);
}
```

Activity 6.3

1

```

#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#define MYPORT 7890 /* the port users will be connecting to */
#define BACKLOG 5 /* how many pending connections queue will hold */
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 their_addr; /* connector's address information */
    int sin_size;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    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 */
    bzero(&(my_addr.sin_zero), 8); /* zero the rest of the struct */

    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    listen(sockfd, BACKLOG);
    sin_size = sizeof(struct sockaddr_in);
    new_fd = accept(sockfd, &their_addr, &sin_size);
    .
    .
    .

```

2

```

#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define DEST_IP "202.40.157.146" /* ucourse2 server */
#define DEST_PORT 22
main()
{
    int sockfd;
    struct sockaddr_in dest_addr; /* will hold the destination addr */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    dest_addr.sin_family = AF_INET; /* host byte order */
    dest_addr.sin_port = htons(DEST_PORT); /*short, network byte order */
    dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
    bzero(&(dest_addr.sin_zero), 8); /* zero the rest of the struct */

    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
    .
    .
    .

```

Once you define the socket descriptor and the destination socket information, you can connect to the corresponding host.

Activity 6.4

- 1 In fact, we can call `connect()` for a connectionless (UDP) socket at a client. But this does not result in anything like a connection-oriented (TCP) communication: there is no connection established at all. Instead, the kernel just checks for any immediate errors (e.g., an obviously unreachable destination), records the IP address and port number of the server (from the socket address structure passed to `connect()`), and returns immediately to the calling process. Also, `send()` and `recv()` are used for exchanging data subsequently, but not `sendto()` and `recvfrom()`, since there is no need to specify the server's address structure again.
- 2 If you want more control over how the socket closes, you can use the `shutdown()` function; it allows you to cut off communication in a certain direction, or both ways (just like `close()` does.)

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

sockfd is the socket file descriptor you want to shut down, and *how* is one of the following:

- 0 — Further receives are disallowed
- 1 — Further sends are disallowed
- 2 — Further sends and receives are disallowed (such as `close()`)

Activity 6.5

```
#!/usr/bin/perl -w
use SOAP::Lite;
print "Convert Fahrenheit to Celsius.\nInput
Fahrenheit value=>";
$temp = <STDIN>;
print "The converted Celsius value => ";
print SOAP::Lite
    -> uri('ns://ct-212-labsupport/Temperatures')
    -> proxy('http://labsupport.no-ip.org/cgi-
bin/temper.cgi')
    -> f2c($temp)
    -> result."\n";
```


Glossary

connection-oriented protocol — A protocol that requires that a channel of communication to be established.

DCOM — Distributed Component Object Model (DCOM) is a proprietary Microsoft technology for communication among software components distributed across networked computers. DCOM provides the communication substrate under Microsoft's COM+ application server infrastructure.

kernel — A Unix kernel — the core or key components of the operating system — consists of many kernel subsystems such as process management, memory management, file management, device management and network management.

PID — The process identification (PID) is used by the UNIX kernel to identify each task operating in the system.

process — A process is an instance of a program that is being executed by a computer system that has the ability to run several programs concurrently (e.g. UNIX).

protocol — In computing, a protocol is a convention or standard that controls or enables the connection, communication, and data transfer between two computing endpoints.

RPC — A remote procedure call (RPC) facility can make it possible to implement client-server applications without needing to explicitly issue requests for communication services from within the application program.

SOAP — SOAP is a protocol for exchanging XML-based messages over computer networks, normally using http. SOAP forms the foundation layer of the Web services protocol stack providing a basic messaging framework upon which abstract layers can be built. The Remote Procedure Call (RPC) is the most common model for SOAP implementations

socket — A socket is an endpoint in network communications. A pair of sockets is required for connection-oriented communications.

socket address — A socket address is a combination of the host IP address and port number, associated with a particular process. In socket programming, communication is made between a pair of socket addresses — one on the server, the other on the client.

system calls — System calls provide the interface between a process and the kernel. Most operations interacting with the system require permissions not available to a user level process, i.e. any I/O operations or any form of communication with other processes (including socket communication) requires the use of system calls. User processes request the kernel to carry such operations on their half by invoking appropriate system calls.

Transmission Control Protocol over Internet Protocol (TCP/IP) —

TCP/IP was developed by the Defense Advanced Research Project Agency (DARPA) for internetworking, encompassing both network layer and transport layer protocols. Whereas TCP and IP specify two protocols at specific layers, TCP/IP is often used to refer to the entire US Department of Defense (DoD) protocol suite based on these, including Telnet, FTP and UDP.

TCP — The Transmission Control Protocol (TCP) is one of the core protocols of the Internet Protocol Suite. TCP is connection-oriented. It provides reliable, ordered delivery of a stream of bytes from one program on one computer to another program on another computer. Common applications of TCP include web, email and file transfer.

UDP — The User Datagram Protocol (UDP) is one of the core protocols of the Internet Protocol Suite. UDP is connectionless. It does not guarantee reliability or ordering in the way that TCP does. Datagrams may arrive out of order, appear duplicated, or go missing without notice. With less overhead, UDP is faster and more efficient. Time-sensitive applications often use UDP because dropped packets are preferable to delayed packets.

well-known ports — They refers to those TCP/IP port numbers in the range 0–1023.

Web services — A Web service (also Web Service) is defined by the W3C as ‘a software system designed to support interoperable machine-to-machine interaction over a network.’ Web services are frequently just Web APIs that can be accessed over a network, such as the Internet, and that can be executed on a remote system hosting the requested services.

References

Comer, D E, Stevens, D L (1996) *Internetworking with TCP/IP Vol. III — Client-server Programming and Applications*, Prentice Hall.

Online materials

‘Beej’s Guide to Network Programming Using Internet Sockets’:

<http://beej.us/guide/bgnet/>

‘Quick Start with SOAP’:

<http://www.perl.com/pub/2001/01/soap.html>

‘Unix Sockets’:

<http://www.cis.temple.edu/~ingargio/old/cis307f95/readings/unix4.html>