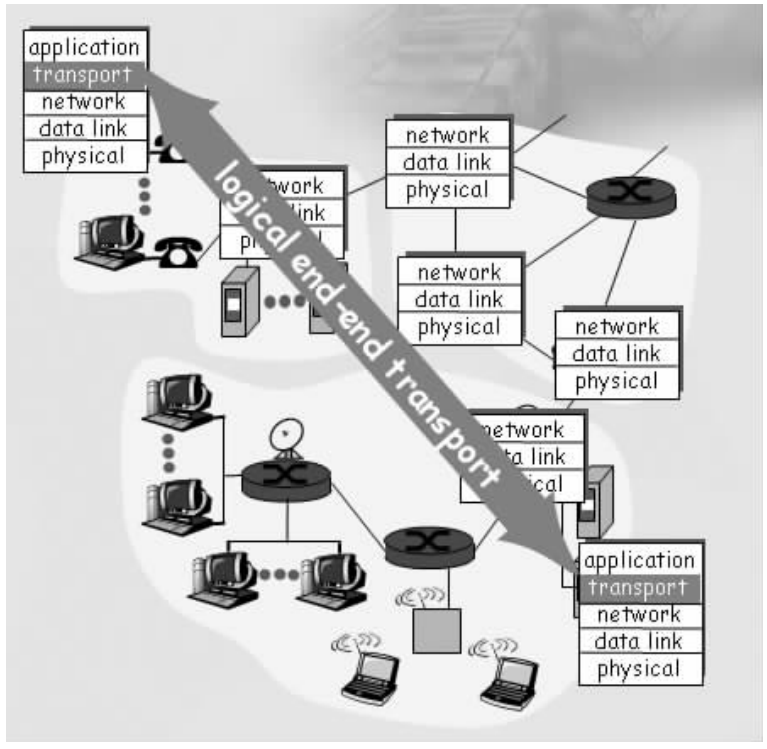
Unit 5　The Transport Layer

Section A　The Transport Service and UDP



I. The transport service

The transport layer is the heart of the whole protocol hierarchy, whose task is to provide reliable, **cost-effective** data transport from the source machine to the destination machine. It provides services to the application layer, which is implemented by the transport **entity** using transport service **primitives.** ①

1. Transport entity

Transport entity is the hardware and/or software within the transport layer that does the work. It can be located in the operating system **kernel** or in a **library package** to the application layer.

2. Transport Service Primitives

To allow users to access the transport service, it must provide some operations to application programs. Each transport service, also called primitive, is done by its own

interface. The simplest transport service primitive is shown in Figure 5.1.

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Figure 5.1 The primitives for a simple transport service

Considering an application with a server and a number of remote clients, these primitives may be used. To start with, the server executes a LISTEN primitive, typically by calling a library procedure to **block** the server until a client comes.② When a client wants to communicate with the server, it executes a CONNECT primitive. The transport entity carries out this primitive by blocking the caller and sending a packet to the server. Now data can be exchanged using the SEND and RECEIVE primitives. When data transmission between the sender and the receiver is done, the connection is must be released to **free up** table space within the two transport entities.

Disconnection has two **variants**: **asymmetric** and **symmetric**. In the asymmetric variant, either transport user can issue a DISCONNECT primitive. In the symmetric variant, each direction is closed separately, independently of the other one. If one side sends a DISCONNECT, it does not send data any more but is willing to accept data from its partner.

Another set of transport primitives is Berkeley **Sockets**, which is used in Berkeley UNIX for TCP (Transport Control Protocol). It involves in eight primitives: SOCKET, BIND, LISTEN, ACCEPT, CONNECT, SEND, RECEIVE, CLOSE. The flow chart of Berkeley Sockets for TCP is shown in Figure 5.2.

II. UDP

There are two protocols available in the transport layer to its applications: UDP and TCP. Now let's study how UDP works.

1. The UDP protocol

UDP (User Datagram Protocol) is **connectionless** and provides a way for applications to send data without having to establish a connection. UDP transmits segments consisting of an 8-byte header followed by the payload. Its header is shown in Figure 5.3.

The *source port* and the *destination port* field are used to **specify** which process on the sending/receiving machine for sending back a reply.

The *length field* includes the 8-byte header and the data.

The *checksum* is optional and stored as 0 if not computed.

Compared with TCP, UDP is a connectionless service and lacks congestion control.

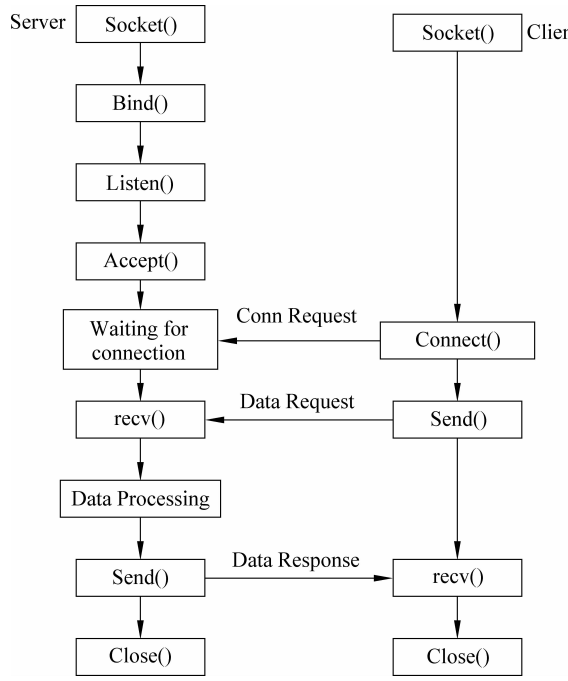


Figure 5.2 Flow Chart of Socket for TCP

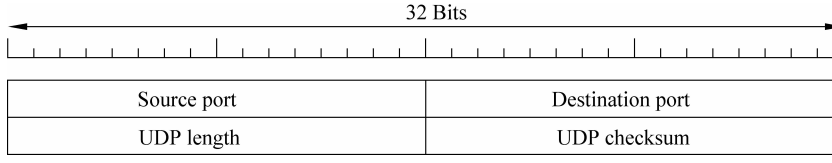


Figure 5.3 The UDP header

But UDP has its own features, such as small segment header **overhead** and **unregulated** send rate. It is also commonly used today with multimedia applications, such as Internet phone, real-time video conferencing, and streaming of stored audio and video.

2. An example of UDP applications

DNS is an example of an application-layer protocol that uses UDP.

When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to a UDP socket. Without performing any handshaking, UDP adds a header fields to the message and passes the resulting segment to the network layer. The network layer **encapsulates** the UDP **segment** into a **datagram** and sends the datagram to a name server. The DNS application at the querying host then waits for a reply to its query. If it doesn't receive a reply, it either tries sending the query to another name-server, or it informs the invoking application that it can't get a reply.

In practice, DNS almost always runs over UDP.

3. UDP client-server programming

Here we present the application of UDP in Java. The reason has two: first, it is more

neatly and cleanly written in Java. Second, client-server programming in Java is becoming increasingly popular. The client/server program using UDP is shown in Figure 5.4.

UDPClient.java

```
import java.io.*;
import java.net.*;
class UDPClient {
public static void main(String args[]) throws Exception
{
BufferedReader inFromUser =
new BufferedReader(new InputStreamReader(System.in));
DatagramSocket clientSocket = new DatagramSocket();
InetAddress IPAddress = InetAddress.getByName("hostname");
byte[] sendData = new byte[1024];
byte[] receiveData = new byte[1024];
String sentence = inFromUser.readLine();
sendData = sentence.getBytes();
DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress,9876);
clientSocket.send(sendPacket);
DatagramPacket receivePacket =
new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
String modifiedSentence =
new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
  }
}
```

UDPServer.java

```
import java.io.*;
import java.net.*;
class UDPServer {
public static void main(String args[]) throws Exception
{
DatagramSocket serverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024];
byte[] sendData = new byte[1024];
while(true)
{
DatagramPacket receivePacket =new DatagramPacket(receiveData, receiveData.length);
serverSocket.receive(receivePacket);
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
String capitalizedSentence = sentence.toUpperCase();
sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket =new DatagramPacket(sendData, sendData.length, IPAddress,
port);
serverSocket.send(sendPacket);
    }
  }
}
```

Figure 5.4 Implementation of UDP programming

Notes

① It provides services to the application layer, which is implemented by the transport entity using transport service primitives. 这句话中的 which 指代前面的 services，由传输实体实现；using 动词的 ing 形式，整个动词短语修饰前面的 entity。

② To start with, the server executes a LISTEN primitive, typically by calling a library procedure to block the server until a client comes. 在这个句子中，主语是 server，谓语是 execute，宾语是 primitive，其他的都是修饰成分，介词 by 接动名词短语修饰主语 server，在动名短语里还包含一个 until 引导的从句修饰其前面的阻塞的服务器。

New Words and Phrases

cost-effective	[ˌkɔsti'fektiv]	*adj.*	有成本效益的；划算的
entity	['entəti]	*n.*	实体；存在；本质
primitive	['primətiv]	*n.*	文艺复兴前的艺术家；原始人
kernel	['kɜ:nl]	*n.*	核心；仁；中心；精髓
block	[blɔk]	*v.*	阻塞
free up			释放
disconnection	[ˌdiskə'nekʃən]	*n.*	断开
variant	['veəriənt]	*n.*	变体
asymmetric	[ˌeisi'metrik]	*adj.*	不对称的
symmetric	[si'metrik]	*adj.*	对称的；匀称的
connectionless	[kə'nekʃnles]	*n.*	无连接

specify	['spesifai]	*v.*	详细说明;指定;阐述
overhead	['əʊvəhed]	*n.*	经常开支;普通用费;总开销
unregulate	[ˌʌn'regjuleitid]	*adj.*	不规范的;紊乱的;不受管理的
encapsulate	[in'kæpsjuleit]	*v.*	装入胶囊;封进内部;压缩;概括
segment	['segmənt]	*n.*	部分;弓形;瓣;段;节
datagram	['detəˌgræm]	*n.*	数据包;数据报

Computer Terminologies

library package	程序包
Sockets	套接字
UDP (User Datagram Protocol)	用户数据报协议

Exercises

Answer the questions.

1. Describe the differences between UDP and TCP.
2. What does the service primitives mean?
3. What does the transport service supply and list them and explain?
4. Tell the differences between asymmetric variant disconnection and symmetric variant disconnection.
5. Which kind of programming language is commonly used in UDP client-server programming.
6. Give some examples of UDP applications.

Extending Your Reading

Case history—Vinton Cerf, Robert Kahn and TCP/IP

In the early 1970's, packet-switched networks began to proliferate, with the ARPAnet—the precursor of the Internet—being just one of many networks. Each of these networks had its own protocol. Two researchers, Vinton Cerf and Robert Kahn recognized the importance of interconnecting these networks and invented a cross-network protocol called TCP/IP, which stands for Transmission Control Protocol/Internet Protocol. The TCP/IP protocol, which is the bread and butter of today's Internet, was devised before PCs and workstations, the web, streaming audio, and chat. Cerf and Kahn saw the need for a networking protocol that, on the one hand, provides broad support for yet-to-be-defined applications and, on the other hand, allows arbitrary hosts and link-layer protocols to interoperate.

参考译文

传输服务和用户数据报协议

传输服务

传输层是整个分层协议的核心，它的任务是为源主机到目标主机提供可信的、成本有效的数据传输。它为应用层提供服务，由传输层通过传输服务原语来实现。

传输实体

传输实体是实现任务的位于传输层的硬件或者软件。它可能位于操作系统内核或者位于应用层的库程序包中。

传输服务原语

为了让用户接入传输服务，它必须向应用程序提供一些操作。也就是，每一个传输服务由它的接口来实现，叫做原语。最简单的传输服务原语如图 5.1 所示。

原语	发送的分组	含义
LISTEN	(无)	阻塞，直到有进程与它建立连接
CONNECT	CONNECTION REQ.	主动尝试建立一个连接
SEND	DATA	发送信息
RECEIVE	(没有)	阻塞，直到一个分组信息到来
DISCONNECT	DISCONNECTION REQ.	释放一个已建立的连接

图 5.1　简单传输服务原语

考虑带一台服务器多台远程客户端的应用，就要用到这些原语。开始，服务器执行 LISTEN 原语，典型地通过呼叫库程序阻塞服务器直到某个客户端到来。当某个客户端想要与服务器通信，它执行 CONNECT 原语。传输层实体通过阻塞调用者以及发送分组给服务器执行此原语。现在，数据可以通过使用 SEND 和 RECEIVE 原语进行交换了。当数据传输在发送者和接收者之间完成，两个传输实体之间的连接必须被释放以释放表空间。

断掉连接有两种方式：非对称和对称。在非对称方式中，任何的传输用户都可以发起一个 DISCONNECT 原语。在对称方式中，每一方都是单独关闭，独立于另外一方。如果一方发送了一个 DISCONNECT，它不再发送任何数据，但是它仍能从另一方接收数据。

另一套传输原语是 Berkely 套接字，它用于 Berkely UNIX 的传输控制协议。它涉及 8 个原语：SOCKET、BIND、LISTEN、ACCEPT、CONNECT、SEND、RECEIVE 和 CLOSE。用于传输控制协议的 Berkeley 套接字流程图如 5.2 所示。

UDP

传输层有两种可用的协议提供应用：UDP 和 TCP。现在学习 UDP 是如何工作的。

UDP 协议

UDP 是无连接的，为应用提供了一种发送数据不需要建立连接的方式。UDP 传输分段，它包含 8 字节的头部，接着是负载。它的头部如图 5.3 所示。

源端口和目标端口字段用来确定发送或者接收机器上哪个进程用来发送答复。

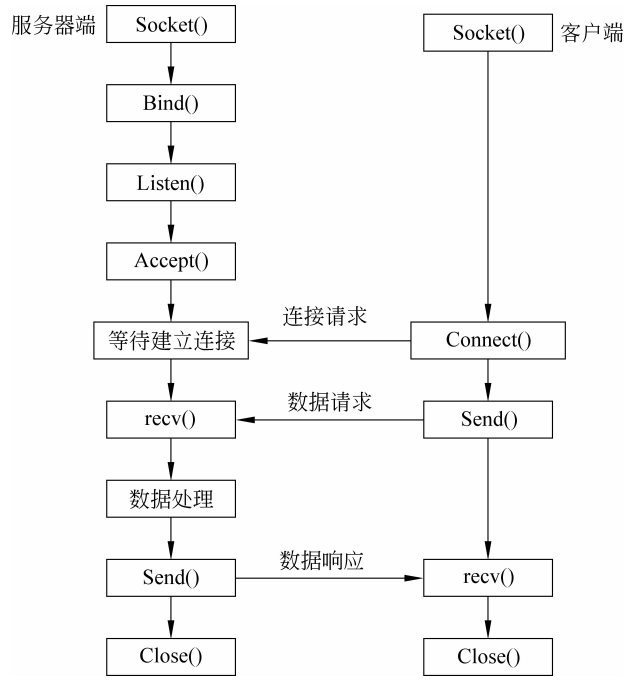


图 5.2 TCP 的 Socket 通信流程图

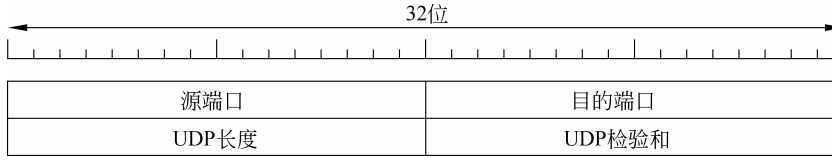


图 5.3 UDP 头部格式

长度字段包括了 8 字节的头部以及数据。

校验和字段是可选项，如果无计算，其值是 0。

与 TCP 相比，UDP 是一个无连接服务，缺少拥塞控制。但是 UDP 有自己的特点，如小的段报头开销，没有规定的发送速率。如今，它经常用于多媒体应用，如网络电话、实时视频。

一个 UDP 应用例子

DNS 是一个应用层协议使用 UDP 的例子。

当主机中的 DNS 应用想要发起一个查询，它构造一个 DNS 查询消息，并把消息发送给 UDP 套接字。不需要任何握手，UDP 对消息增加一个头字段，把结果段发送给网络层。网络层把 UDP 段封装为数据报，并把数据报发给命名服务器。查询端主机上的 DNS 等待查询的回复。如果它没有收到回复，将会尝试向另外的命名服务器发送查询，或者它通知发起程序，它不能得到回复。

实际上，DNS 一直运行在 UDP 之上。

UDP 客户-服务器编程

这里用 Java 语言来描述 UDP 的应用。理由有两点：其一，采用 Java 语言更简洁干净。

其二，Java 客户-服务器编程变得日益流行。UDP 客户服务器端编程如图 5.4 所示。

UDPClient.java

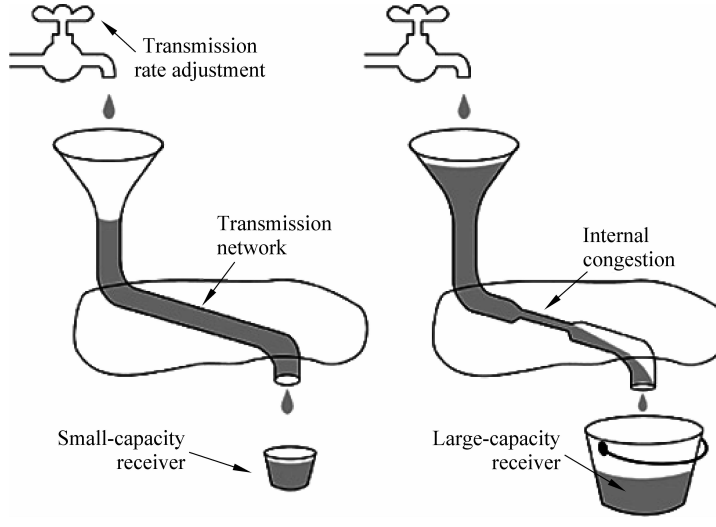
```
import java.io.*;
import java.net.*;
class UDPClient {
public static void main(String args[]) throws Exception
{
BufferedReader inFromUser =
new BufferedReader(new InputStreamReader(System.in));
DatagramSocket clientSocket = new DatagramSocket();
InetAddress IPAddress = InetAddress.getByName("hostname");
byte[] sendData = new byte[1024];
byte[] receiveData = new byte[1024];
String sentence = inFromUser.readLine();
sendData = sentence.getBytes();
DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress,9876);
clientSocket.send(sendPacket);
DatagramPacket receivePacket =
new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
String modifiedSentence =
new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
   }
}
```

UDPServer.java

```
import java.io.*;
import java.net.*;
class UDPServer {
public static void main(String args[]) throws Exception
{
DatagramSocket serverSocket = new DatagramSocket(9876);
byte[] receiveData = new byte[1024];
byte[] sendData = new byte[1024];
while(true)
{
DatagramPacket receivePacket =new DatagramPacket(receiveData, receiveData.length);
serverSocket.receive(receivePacket);
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();
String capitalizedSentence = sentence.toUpperCase();
sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket =new DatagramPacket(sendData, sendData.length, IPAddress,
port);
serverSocket.send(sendPacket);
      }
   }
}
```

图 5.4　UDP 编程实现

Section B　TCP and its Congestion Control



I. TCP

TCP (Transmission Control Protocol) operates in the transport layer, and is designed to provide a **reliable** end-to-end byte **stream** transmission over an unreliable internetwork. The internetworks may differ from each other in topologies, bandwidths, delays, packet sizes, and other parameters. TCP can dynamically adapt to these properties and is **robust**

in the face of many kinds of failures. It is formally defined in RFC 793.

1. Sockets and ports

TCP provides services to the application layer. TCP service is obtained by both the sender and receiver creating end points, called sockets.[①] A socket consists of the IP address of the host and its 16-bit port number. A TCP connection is identified **uniquely** by two ends in communication. That is, TCP connection = {socket1, socket2} = {(IP1: port1), (IP2: port2)}.

A port is the TCP name for a **TSAP** (Transport Service Access Point) and is just used to identify the process of the application layer. Generally, ports can be divided into three categories:

- Well-known ports: 0～1023, reserved for standard services. A few of the better known services are listed in Figure 5.5.
- Registered ports: 1024～49151, reserved for application programs without well-known ports.
- Client ports: 49152～65535, reserved for client processes and used temporarily.

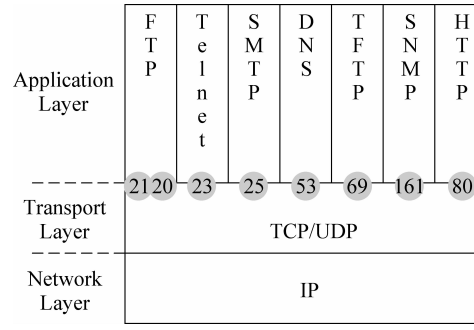


Figure 5.5 Some assigned ports

2. The TCP protocol

One of key features of TCP is that every byte on a TCP connection has its own 32-bit sequence number, which is used for **acknowledgements** and for the window mechanism. TCP exchanges data in the form of segments. Every segment begins with a **fixed-format**, 20-byte header, shown in Figure 5.6.

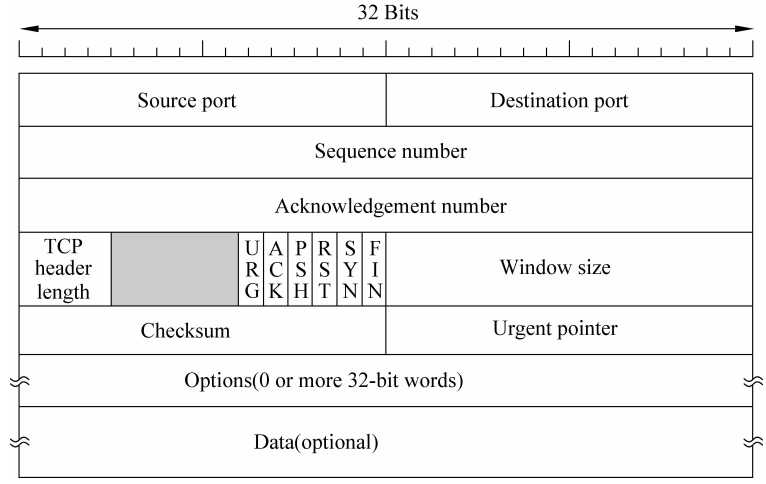


Figure 5.6 The TCP Segment Header

The *Source port* and *Destination port* fields identify the local end points of the

connection.

The *Sequence number* field is the number of the sending data.

The *Acknowledgement number* field specifies the number of the next byte expected, not the last byte correctly received.

The *TCP header length* tells how many 32-bit words are contained in the TCP header.

Next there are a 6-bit field that is not used, and *six* 1-*bit* flags. URG is set to 1 if the Urgent pointer is in use. The *ACK* bit is set to 1 to indicate that the Acknowledgement number is valid. The *PSH* bit indicates PUSHed data. The *RST* bit is used to reset a connection that has become confused. The *FIN* bit is used to **release** a connection. It specifies that the sender has no more data to transmit.

The *Window size* field tells how many bytes may be sent starting at the byte acknowledged. ②

A *Checksum* is also provided for extra reliability.

3. TCP Connection Establishment

Connections are **established** in TCP by means of the **three-way handshake** between the server and the client. Firstly, the server passively waits for an incoming connection by executing the primitives of LISTEN and ACCEPT. Then, the client executes a CONNECT primitive. The CONNECT primitive sends a TCP segment with the SYN bit on and ACK bit off and waits for a response. ③ When the server receives this segment, it can either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The establishment process of the TCP connection in the normal case is shown in Figure 5.7.

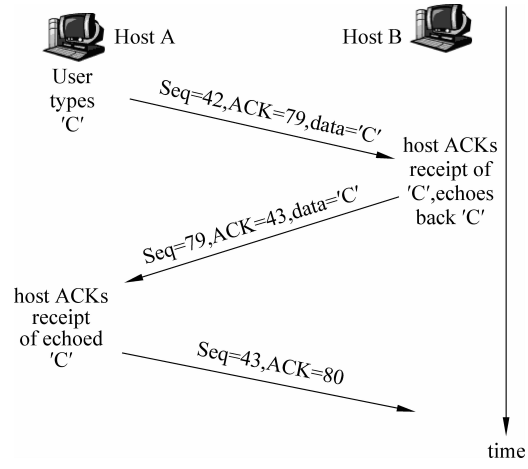


Figure 5.7 Establishment of TCP connection in the normal case

4. TCP Connection Release

To release a connection, either client or server can send a TCP segment with the FIN bit set, which means no more data to transmit. When the FIN is acknowledged, the data transmission in that direction is shut down. However, data may continue to flow