CHAPTER

14

User Datagram Protocol (UDP)

The original TCP/IP protocol suite specifies two protocols for the transport layer: UDP and TCP. We first focus on UDP, the simpler of the two, before discussing TCP in Chapter 15. A new transport-layer protocol, SCTP, has been designed, which we will discuss in Chapter 16.

OBJECTIVE

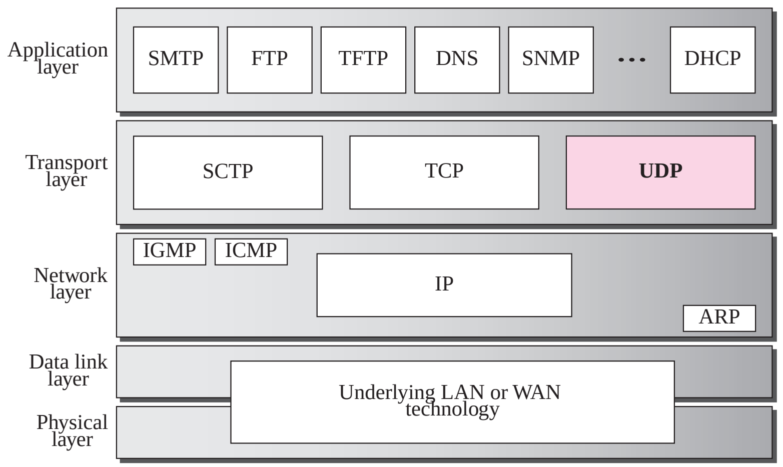
We have several objectives for this chapter:

- ❑ To introduce UDP and show its relationship to other protocols in the TCP/IP protocol suite.
- ❑ To explain the format of a UDP packet, which is called a user datagram, and discuss the use of each field in the header.
- ❑ To discuss the services provided by the UDP such as process-to-process delivery, rudimentary error control, multiplexing/demultiplexing, and queuing.
- ❑ To show how to calculate the optional checksum and why the sender of a UDP packet needs to add a pseudoheader to the packet when calculating the checksum.
- ❑ To discuss how some application programs can benefit from the simplicity of UDP.
- ❑ To briefly discuss the structure of a software package that implements UDP and give the description of control-block, input, and output module.

14.1 INTRODUCTION

Figure 14.1 shows the relationship of the **User Datagram Protocol** (UDP) to the other protocols and layers of the TCP/IP protocol suite: UDP is located between the application layer and the IP layer, and serves as the intermediary between the application programs and the network operations.

Figure 14.1 *Position of UDP in the TCP/IP protocol suite*



As discussed in Chapter 13, a transport layer protocol usually has several responsibilities. One is to create a process-to-process communication; UDP uses port numbers to accomplish this. Another responsibility is to provide control mechanisms at the transport level. UDP does this task at a very minimal level. There is no flow control mechanism and there is no acknowledgment for received packets. UDP, however, does provide error control to some extent. If UDP detects an error in the received packet, it silently drops it.

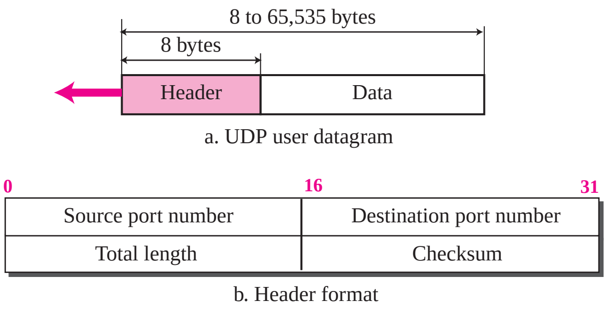
UDP is a **connectionless, unreliable transport protocol.** It does not add anything to the services of IP except for providing process-to-process communication instead of host-to-host communication.

If UDP is so powerless, why would a process want to use it? With the disadvantages come some advantages. UDP is a very simple protocol using a minimum of overhead. If a process wants to send a small message and does not care much about reliability, it can use UDP. Sending a small message using UDP takes much less interaction between the sender and receiver than using TCP.

14.2 USER DATAGRAM

UDP packets, called **user datagrams,** have a fixed-size header of 8 bytes. Figure 14.2 shows the format of a user datagram. The fields are as follows:

Figure 14.2 *User datagram format*



- **Source port number.** This is the port number used by the process running on the source host. It is 16 bits long, which means that the port number can range from 0 to 65,535. If the source host is the client (a client sending a request), the port number, in most cases, is an ephemeral port number requested by the process and chosen by the UDP software running on the source host. If the source host is the server (a server sending a response), the port number, in most cases, is a well-known port number.
- **Destination port number.** This is the port number used by the process running on the destination host. It is also 16 bits long. If the destination host is the server (a client sending a request), the port number, in most cases, is a well-known port number. If the destination host is the client (a server sending a response), the port number, in most cases, is an ephemeral port number. In this case, the server copies the ephemeral port number it has received in the request packet.
- **Length.** This is a 16-bit field that defines the total length of the user datagram, header plus data. The 16 bits can define a total length of 0 to 65,535 bytes. However, the total length needs to be much less because a UDP user datagram is stored in an IP datagram with the total length of 65,535 bytes. The length field in a UDP user datagram is actually not necessary. A user datagram is encapsulated in an IP datagram. There is a field in the IP datagram that defines the total length. There is another field in the IP datagram that defines the length of the header. So if we subtract the value of the second field from the first, we can deduce the length of the UDP datagram that is encapsulated in an IP datagram.

UDP length = IP length – IP header's length

However, the designers of the UDP protocol felt that it was more efficient for the destination UDP to calculate the length of the data from the information provided in the UDP user datagram rather than ask the IP software to supply this information. We should remember that when the IP software delivers the UDP user datagram to the UDP layer, it has already dropped the IP header.

- **Checksum.** This field is used to detect errors over the entire user datagram (header plus data). The checksum is discussed in the next section.

Example 14.1

The following is a dump of a UDP header in hexadecimal format.

```
CB84000D001C001C
```

a. What is the source port number?
b. What is the destination port number?
c. What is the total length of the user datagram?
d. What is the length of the data?
e. Is the packet directed from a client to a server or vice versa?
f. What is the client process?

Solution

a. The source port number is the first four hexadecimal digits ($CB84_{16}$), which means that the source port number is 52100.
b. The destination port number is the second four hexadecimal digits ($000D_{16}$), which means that the destination port number is 13.
c. The third four hexadecimal digits ($001C_{16}$) define the length of the whole UDP packet as 28 bytes.
d. The length of the data is the length of the whole packet minus the length of the header, or 28 – 8 = 20 bytes.
e. Since the destination port number is 13 (well-known port), the packet is from the client to the server.
f. The client process is the Daytime (see Table 14.1).

14.3 UDP SERVICES

We discussed the general services provided by a transport layer protocol in Chapter 13. In this section, we discuss what portions of those general services are provided by UDP.

Process-to-Process Communication

UDP provides process-to-process communication discussed in Chapter 13 using sockets, a combination of IP addresses and port numbers. Several port numbers used by UDP are shown in Table 14.1.

Table 14.1 *Well-known Ports used with UDP*

Port	*Protocol*	*Description*
7	Echo	Echoes a received datagram back to the sender
9	Discard	Discards any datagram that is received
11	Users	Active users
13	Daytime	Returns the date and the time
17	Quote	Returns a quote of the day
19	Chargen	Returns a string of characters
53	Domain	Domain Name Service (DNS)
67	Bootps	Server port to download bootstrap information
68	Bootpc	Client port to download bootstrap information
69	TFTP	Trivial File Transfer Protocol
111	RPC	Remote Procedure Call
123	NTP	Network Time Protocol
161	SNMP	Simple Network Management Protocol
162	SNMP	Simple Network Management Protocol (trap)

Connectionless Services

As mentioned previously, UDP provides a *connectionless service*. This means that each user datagram sent by UDP is an independent datagram. There is no relationship between the different user datagrams even if they are coming from the same source process and going to the same destination program. The user datagrams are not numbered. Also, there is no connection establishment and no connection termination as is the case for TCP. This means that each user datagram can travel on a different path.

One of the ramifications of being connectionless is that the process that uses UDP cannot send a stream of data to UDP and expect UDP to chop them into different related user datagrams. Instead each request must be small enough to fit into one user datagram. Only those processes sending short messages, messages less than 65,507 bytes (65,535 minus 8 bytes for the UDP header and minus 20 bytes for the IP header), can use UDP.

Flow Control

UDP is a very simple protocol. There is no *flow control*, and hence no window mechanism. The receiver may overflow with incoming messages. The lack of flow control means that the process using UDP should provide for this service, if needed.

Error Control

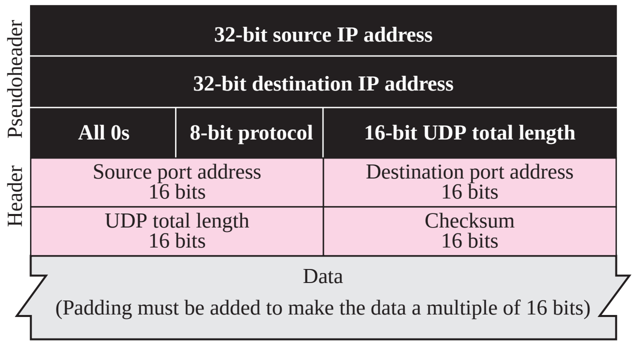
There is no *error control* mechanism in UDP except for the checksum. This means that the sender does not know if a message has been lost or duplicated. When the receiver detects an error through the checksum, the user datagram is silently discarded. The lack of error control means that the process using UDP should provide for this service if needed.

Checksum

We have already talked about the concept of the *checksum* and the way it is calculated for IP in Chapter 7. UDP checksum calculation is different from the one for IP. Here the checksum includes three sections: a pseudoheader, the UDP header, and the data coming from the application layer.

The **pseudoheader** is the part of the header of the IP packet in which the user datagram is to be encapsulated with some fields filled with 0s (see Figure 14.3).

Figure 14.3 *Pseudoheader for checksum calculation*


Pseudoheader
Header
32-bit source IP address
32-bit destination IP address
All 0s
8-bit protocol
16-bit UDP total length
Source port address 16 bits
Destination port address 16 bits
UDP total length 16 bits
Checksum 16 bits
Data
(Padding must be added to make the data a multiple of 16 bits)

If the checksum does not include the pseudoheader, a user datagram may arrive safe and sound. However, if the IP header is corrupted, it may be delivered to the wrong host.

The protocol field is added to ensure that the packet belongs to UDP, and not to TCP. We will see later that if a process can use either UDP or TCP, the destination port number can be the same. The value of the protocol field for UDP is 17. If this value is changed during transmission, the checksum calculation at the receiver will detect it and UDP drops the packet. It is not delivered to the wrong protocol.

Note the similarities between the pseudoheader fields and the last 12 bytes of the IP header.

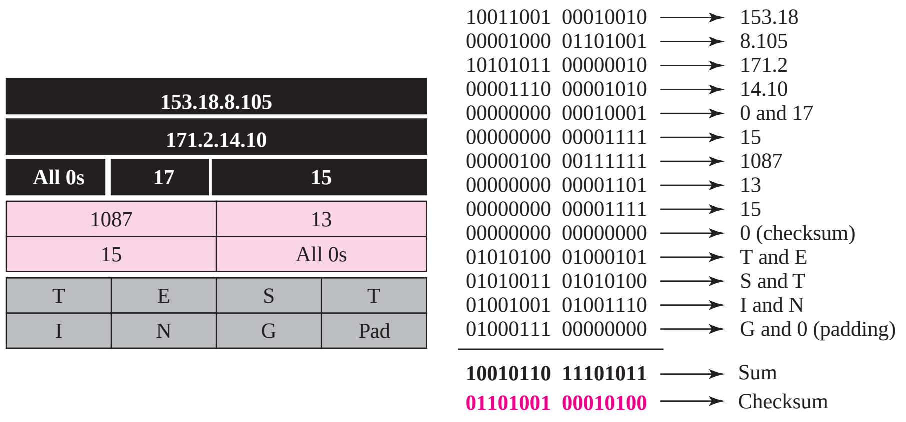
Example 14.2

Figure 14.4 shows the checksum calculation for a very small user datagram with only 7 bytes of data. Because the number of bytes of data is odd, padding is added for checksum calculation. The pseudoheader as well as the padding will be dropped when the user datagram is delivered to IP (see Appendix F).

Optional Inclusion of Checksum

The sender of a UDP packet can choose not to calculate the checksum. In this case, the checksum field is filled with all 0s before being sent. In the situation that the sender decides to calculate the checksum, but it happens that the result is all 0s, the checksum is changed to all 1s before the packet is sent. In other words, the sender complements the sum two times. Note that this does not create confusion because the value of checksum is never all 1s in a normal situation (see the next example).

Figure 14.4 *Checksum calculation of a simple UDP user datagram*



10011001 00010010	153.18
00001000 01101001	8.105
10101011 00000010	171.2
00001110 00001010	14.10
00000000 00010001	0 and 17
00000000 00001111	15
00000100 00111111	1087
00000000 00001101	13
00000000 00001111	15
00000000 00000000	0 (checksum)
01010100 01000101	T and E
01010011 01010100	S and T
01001001 01001110	I and N
01000111 00000000	G and 0 (padding)
10010110 11101011	Sum
01101001 00010100	Checksum

Example 14.3

What value is sent for the checksum in one of the following hypothetical situations?

a. The sender decides not to include the checksum.

b. The sender decides to include the checksum, but the value of the sum is all 1s.

c. The sender decides to include the checksum, but the value of the sum is all 0s.

Solution

a. The value sent for the checksum field is all 0s to show that the checksum is not calculated.

b. When the sender complements the sum, the result is all 0s; the sender complements the result again before sending. The value sent for the checksum is all 1s. The second complement operation is needed to avoid confusion with the case in part a.

c. This situation never happens because it implies that the value of every term included in the calculation of the sum is all 0s, which is impossible; some fields in the pseudoheader have nonzero values (see Appendix D).

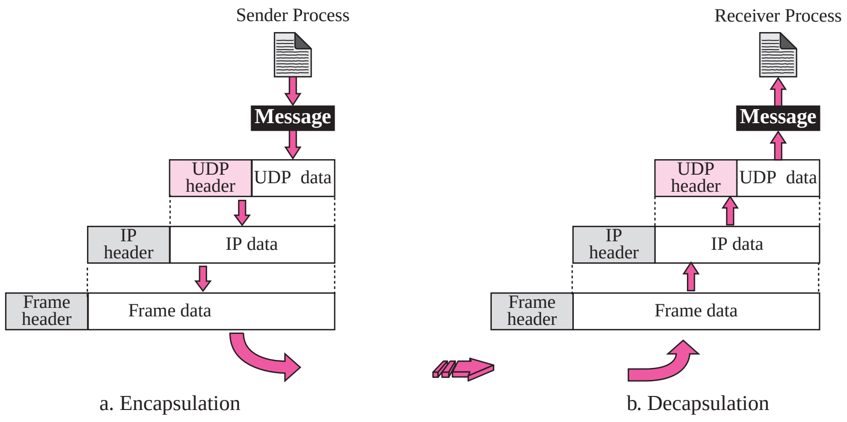
Congestion Control

Since UDP is a connectionless protocol, it does not provide congestion control. UDP assumes that the packets sent are small and sporadic, and cannot create congestion in the network. This assumption may or may not be true today when UDP is used for real-time transfer of audio and video.

Encapsulation and Decapsulation

To send a message from one process to another, the UDP protocol encapsulates and decapsulates messages (see Figure 14.5).

Figure 14.5 *Encapsulation and decapsulation*



Encapsulation

When a process has a message to send through UDP, it passes the message to UDP along with a pair of socket addresses and the length of data. UDP receives the data and adds the UDP header. UDP then passes the user datagram to IP with the socket addresses. IP adds its own header, using the value 17 in the protocol field, indicating that the data has come from the UDP protocol. The IP datagram is then passed to the data link layer. The data link layer receives the IP datagram, adds its own header (and possibly a trailer), and passes it to the physical layer. The physical layer encodes the bits into electrical or optical signals and sends it to the remote machine.

Decapsulation

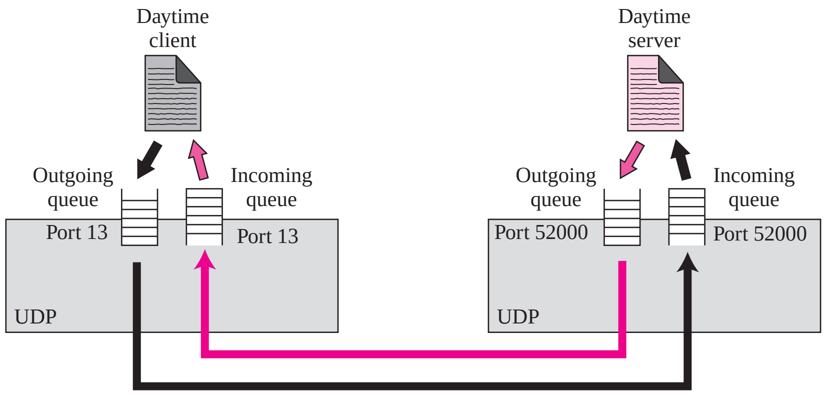
When the message arrives at the destination host, the physical layer decodes the signals into bits and passes it to the data link layer. The data link layer uses the header (and the trailer) to check the data. If there is no error, the header and trailer are dropped and the datagram is passed to IP. The IP software does its own checking. If there is no error, the header is dropped and the user datagram is passed to UDP with the sender and receiver IP addresses. UDP uses the checksum to check the entire user datagram. If there is no error, the header is dropped and the application data along with the sender socket address is passed to the process. The sender socket address is passed to the process in case it needs to respond to the message received.

Queuing

We have talked about ports without discussing the actual implementation of them. In UDP, queues are associated with ports (see Figure 14.6).

At the client site, when a process starts, it requests a port number from the operating system. Some implementations create both an incoming and an outgoing queue associated with each process. Other implementations create only an incoming queue associated with each process.

Figure 14.6 *Queues in UDP*



Note that even if a process wants to communicate with multiple processes, it obtains only one port number and eventually one outgoing and one incoming **queue.** The queues opened by the client are, in most cases, identified by ephemeral port numbers. The queues function as long as the process is running. When the process terminates, the queues are destroyed.

The client process can send messages to the outgoing queue by using the source port number specified in the request. UDP removes the messages one by one, and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system can ask the client process to wait before sending any more messages.

When a message arrives for a client, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a *port unreachable* message to the server. All of the incoming messages for one particular client program, whether coming from the same or a different server, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the server.

At the server site, the mechanism of creating queues is different. In its simplest form, a server asks for incoming and outgoing queues using its well-known port when it starts running. The queues remain open as long as the server is running.

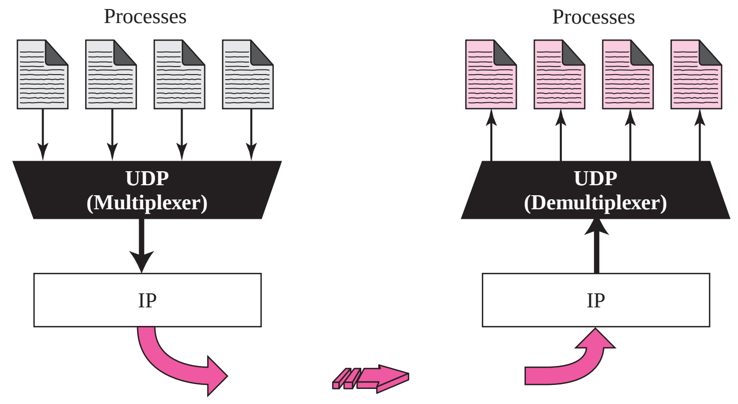
When a message arrives for a server, UDP checks to see if an incoming queue has been created for the port number specified in the destination port number field of the user datagram. If there is such a queue, UDP sends the received user datagram to the end of the queue. If there is no such queue, UDP discards the user datagram and asks the ICMP protocol to send a port unreachable message to the client. All of the incoming messages for one particular server, whether coming from the same or a different client, are sent to the same queue. An incoming queue can overflow. If this happens, UDP drops the user datagram and asks for a port unreachable message to be sent to the client.

When a server wants to respond to a client, it sends messages to the outgoing queue using the source port number specified in the request. UDP removes the messages one by one, and, after adding the UDP header, delivers them to IP. An outgoing queue can overflow. If this happens, the operating system asks the server to wait before sending any more messages.

Multiplexing and Demultiplexing

In a host running a TCP/IP protocol suite, there is only one UDP but possibly several processes that may want to use the services of UDP. To handle this situation, UDP multiplexes and demultiplexes (see Figure 14.7).

Figure 14.7 *Multiplexing and demultiplexing*



Multiplexing

At the sender site, there may be several processes that need to send user datagrams. However, there is only one UDP. This is a many-to-one relationship and requires multiplexing. UDP accepts messages from different processes, differentiated by their assigned port numbers. After adding the header, UDP passes the user datagram to IP.

Demultiplexing

At the receiver site, there is only one UDP. However, we may have many processes that can receive user datagrams. This is a one-to-many relationship and requires demultiplexing. UDP receives user datagrams from IP. After error checking and dropping of the header, UDP delivers each message to the appropriate process based on the port numbers.

Comparison between UDP and Generic Simple Protocol

We can compare UDP with the connectionless simple protocol we discussed in Chapter 13. The only difference is that UDP provides an optional checksum to detect corrupted packets at the receiver site. If the checksum is added to the packet, the receiving UDP can check the packet and discard the packet if it is corrupted. No feedback, however, is sent to the sender.

UDP is an example of the connectionless simple protocol we discussed in Chapter 13 with the exception of an optional checksum added to packets for error detection.

14.4 UDP APPLICATIONS

Although UDP meets almost none of the criteria we mentioned in Chapter 13 for a reliable transport-layer protocol, UDP is preferable for some applications. The reason is that some services may have some side effects that are either unacceptable or not preferable. An application designer needs sometimes to compromise to get the optimum. For example, in our daily life, we all know that a one-day delivery of a package by a carrier is more expensive than a three-day delivery. Although time and cost are both desirable features in delivery of a parcel, they are in conflict with each other. We need to choose the optimum.

In this section, we first discuss some features of UDP that may need to be considered when one designs an application program and then show some typical applications.

UDP Features

We briefly discuss some features of UDP and their advantages and disadvantages.

Connectionless Service

As we mentioned previously, UDP is a connectionless protocol. Each UDP packet is independent from other packets sent by the same application program. This feature can be considered as an advantage or disadvantage depending on the application requirement. It is an advantage if, for example, a client application needs to send a short request to a server and to receive a short response. If the request and response can each fit in one single user datagram, a connectionless service may be preferable. The overhead to establish and close a connection may be significant in this case. In the connection-oriented service, to achieve the above goal, at least 9 packets are exchanged between the client and the server; in connectionless service only two packets are exchanged. The connectionless service provides less delay; the connection-oriented service creates more delay. If delay is an important issue for the application, the connectionless service is preferred.

Example 14.4

A client-server application such as DNS (see Chapter 19) uses the services of UDP because a client needs to send a short request to a server and to receive a quick response from it. The request and response can each fit in one user datagram. Since only one message is exchanged in each direction, the connectionless feature is not an issue; the client or server does not worry that messages are delivered out of order.

Example 14.5

A client-server application such as SMTP (see Chapter 23), which is used in electronic mail, cannot use the services of UDP because a user can send a long e-mail message, which may include multimedia (images, audio, or video). If the application uses UDP and the message does not fit in one single user datagram, the message must be split by the application into different user datagrams. Here the connectionless service may create problems. The user datagrams may arrive and be delivered to the receiver application out of order. The receiver application may not be able to reorder the pieces. This means the connectionless service has a disadvantage for an application program that sends long messages. In SMTP, when one sends a message, one does not expect to receive a response quickly (sometimes no response is required). This means that the extra delay inherent in connection-oriented service is not crucial for SMTP.

Lack of Error Control

UDP does not provide error control; it provides an unreliable service. Most applications expect reliable service from a transport-layer protocol. Although a reliable service is desirable, it may have some side effects that are not acceptable to some applications. When a transport layer provides reliable services, if a part of the message is lost or corrupted, it needs to be resent. This means that the receiving transport layer cannot deliver that part to the application immediately; there is an uneven delay between different parts of the message delivered to the application layer. Some applications by nature do not even notice these uneven delays, but for some they are very crucial.

Example 14.6

Assume we are downloading a very large text file from the Internet. We definitely need to use a transport layer that provides reliable service. We don't want part of the file to be missing or corrupted when we open the file. The delay created between the delivery of the parts are not an overriding concern for us; we wait until the whole file is composed before looking at it. In this case, UDP is not a suitable transport layer.

Example 14.7

Assume we are watching a real-time stream video on our computer. Such a program is considered a long file; it is divided into many small parts and broadcast in real time. The parts of the message are sent one after another. If the transport layer is supposed to resend a corrupted or lost frame, the synchronizing of the whole transmission may be lost. The viewer suddenly sees a blank screen and needs to wait until the second transmission arrives. This is not tolerable. However, if each small part of the screen is sent using one single user datagram, the receiving UDP can easily ignore the corrupted or lost packet and deliver the rest to the application program. That part of the screen is blank for a very short period of the time, which most viewers do not even notice. However, video cannot be viewed out of order, so streaming audio, video, and voice applications that run over UDP must reorder or drop frames that are out of sequence.

Lack of Congestion Control

UDP does not provide congestion control. However, UDP does not create additional traffic in an error-prone network. TCP may resend a packet several times and thus contribute to the creation of congestion or worsen a congested situation. Therefore, in some cases, lack of error control in UDP can be considered an advantage when congestion is a big issue.

Typical Applications

The following shows some typical applications that can benefit more from the services of UDP than from those of TCP.

- UDP is suitable for a process that requires simple request-response communication with little concern for flow and error control. It is not usually used for a process such as FTP that needs to send bulk data (see Chapter 21).
- UDP is suitable for a process with internal flow and error-control mechanisms. For example, the Trivial File Transfer Protocol (TFTP) (see Chapter 21) process includes flow and error control. It can easily use UDP.
- UDP is a suitable transport protocol for multicasting. Multicasting capability is embedded in the UDP software but not in the TCP software.
- UDP is used for management processes such as SNMP (see Chapter 24).
- UDP is used for some route updating protocols such as Routing Information Protocol (RIP) (see Chapter 11).
- UDP is normally used for real-time applications that cannot tolerate uneven delay between sections of a received message.

14.5 UDP PACKAGE

To show how UDP handles the sending and receiving of UDP packets, we present a simple version of the UDP package.

We can say that the UDP package involves five components: a control-block table, input queues, a control-block module, an input module, and an output module. Figure 14.8 shows these five components and their interactions.

Control-Block Table

In our package, UDP has a control-block table to keep track of the open ports. Each entry in this table has a minimum of four fields: the state, which can be FREE or IN-USE, the process ID, the port number, and the corresponding queue number.

Input Queues

Our UDP package uses a set of input queues, one for each process. In this design, we do not use output queues.

Control-Block Module

The control-block module (Table 14.2) is responsible for the management of the control-block table. When a process starts, it asks for a port number from the operating system. The operating system assigns well-known port numbers to servers and ephemeral port numbers to clients. The process passes the process ID and the port number to the control-block module to create an entry in the table for the process. The module

Figure 14.8 *UDP design*

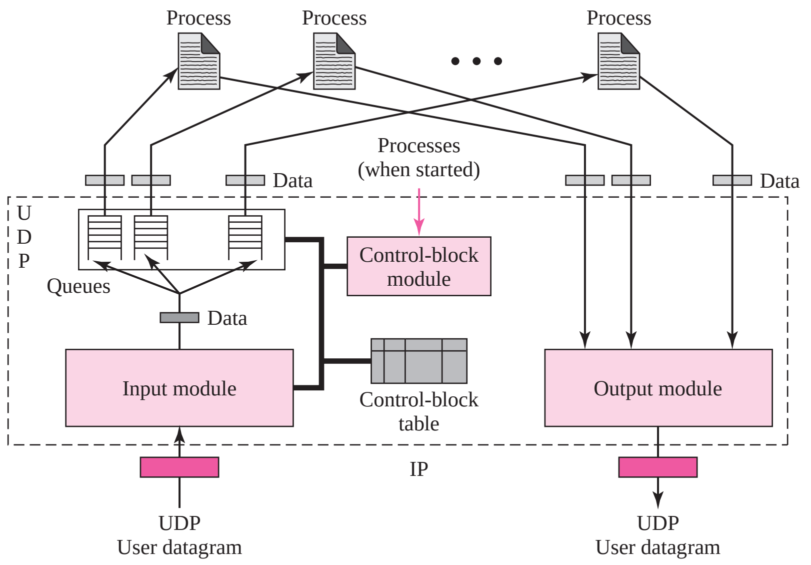


Table 14.2 *Control Block Module*

```
UDP_Control_Block_Module (process ID, port number)
{
    Search the table for a FREE entry.
    if (not found)
        Delete one entry using a predefined strategy.
    Create a new entry with the state IN-USE
    Enter the process ID and the port number.
    Return.
} // End module
```

does not create the queues. The field for queue number has a value of zero. Note that we have not included a strategy to deal with a table that is full.

Input Module

The input module (Table 14.3) receives a user datagram from the IP. It searches the control-block table to find an entry having the same port number as this user datagram. If the entry is found, the module uses the information in the entry to enqueue the data. If the entry is not found, it generates an ICMP message.

Table 14.3 *Input Module*

```
UDP_INPUT_Module (user_datagram)
{
     Look for the entry in the control_block table
     if (found)
     {
          Check to see if a queue is allocated
          If (queue is not allocated)
               allocate a queue
          else
               enqueue the data
     } //end if
     else
     {
          Ask ICMP to send an "unreachable port" message
          Discard the user datagram
     } //end else

     Return.
} // end module
```

Output Module

The output module (Table 14.4) is responsible for creating and sending user datagrams.

Table 14.4 *Output Module*

```
UDP_OUTPUT_MODULE (Data)
{
     Create a user datagram
     Send the user datagram
     Return.
}
```

Examples

In this section we show some examples of how our package responds to input and output. The control-block table at the start of our examples is shown in Table 14.5.

Table 14.5 *The Control-Block Table at the Beginning of Examples*

State	*Process ID*	*Port Number*	*Queue Number*
IN-USE	2,345	52,010	34
IN-USE	3,422	52,011	
FREE			
IN-USE	4,652	52,012	38
FREE			

Example 14.8

The first activity is the arrival of a user datagram with destination port number 52,012. The input module searches for this port number and finds it. Queue number 38 has been assigned to this port, which means that the port has been previously used. The input module sends the data to queue 38. The control-block table does not change.

Example 14.9

After a few seconds, a process starts. It asks the operating system for a port number and is granted port number 52,014. Now the process sends its ID (4,978) and the port number to the control-block module to create an entry in the table. The module takes the first FREE entry and inserts the information received. The module does not allocate a queue at this moment because no user datagrams have arrived for this destination (see Table 14.6).

Table 14.6 *Control-Block Table after Example 14.9*

State	*Process ID*	*Port Number*	*Queue Number*
IN-USE	2,345	52,010	34
IN-USE	3,422	52,011	
IN-USE	4,978	52,014	
IN-USE	4,652	52,012	38
FREE			

Example 14.10

A user datagram now arrives for port 52,011. The input module checks the table and finds that no queue has been allocated for this destination since this is the first time a user datagram has arrived for this destination. The module creates a queue and gives it a number (43). See Table 14.7.

Table 14.7 *Control-Block Table after Example 14.10*

State	*Process ID*	*Port Number*	*Queue Number*
IN-USE	2,345	52,010	34
IN-USE	3,422	52,011	43
IN-USE	4,978	52,014	
IN-USE	4,652	52,012	38
FREE			

Example 14.11

After a few seconds, a user datagram arrives for port 52,222. The input module checks the table and cannot find an entry for this destination. The user datagram is dropped and a request is made to ICMP to send an unreachable port message to the source.

14.6 FURTHER READING

For more details about subjects discussed in this chapter, we recommend the following books and websites. The items enclosed in brackets refer to the reference list at the end of the book.

Books

Several books give information about UDP. In particular, we recommend [Com 06] and [Ste 94].

RFCs

The main RFC related to UDP is RFC 768.

14.7 KEY TERMS

connectionless, unreliable transport protocol	user datagram
pseudoheader	User Datagram Protocol (UDP)
queue	

14.8 SUMMARY

❑ UDP is a transport protocol that creates a process-to-process communication. UDP is a (mostly) unreliable and connectionless protocol that requires little overhead and offers fast delivery. The UDP packet is called a user datagram.

❑ UDP's only attempt at error control is the checksum. Inclusion of a pseudoheader in the checksum calculation allows source and destination IP address errors to be detected. UDP has no flow-control mechanism.

❑ A user datagram is encapsulated in the data field of an IP datagram. Incoming and outgoing queues hold messages going to and from UDP.

❑ UDP uses multiplexing to handle outgoing user datagrams from multiple processes on one host. UDP uses demultiplexing to handle incoming user datagrams that go to different processes on the same host.

❑ A UDP package can involve five components: a control-block table, a control-block module, input queues, an input module, and an output module. The input queues hold incoming user datagrams. The control-block module is responsible for maintenance of entries in the control-block table. The input module creates input queues; the output module sends out user datagrams.