Set 15. Transporting Packets

Skill 15.01: Describe the problem with packets

Skill 15.01 Concepts

The Internet Protocol (IP) describes how to split messages into multiple IP packets and route packets to their destination by hopping from router to router.

IP does not handle all the potential problems that packets might encounter however. Below are some examples,

- What if a computer sends multiple messages to the same destination, how does the receiving computer know which packet belongs to each message?
- Depending on the route each packet takes, the packets can arrive out of order. How does the receiving computer know how to re-assemble the packets?
- Packets can be **corrupted**, which means that for some reason, the received data no longer matches the originally sent data.
- Packets can be **lost** due to problems in the physical layer or in routers' forwarding tables. If even one packet of a message is lost, it may be impossible to put the message back together in a way that makes sense.
- Finally, packets might be **duplicated** due to accidental retransmission of the same packet.

Fortunately, there are higher level protocols in the Internet protocol stack that can deal with these problems. **Transmission Control Protocol (TCP)** is the data transport protocol that's most commonly used on top of IP and it includes strategies for packet ordering, retransmission, and data integrity. **User Datagram Protocol (UDP)** is an alternative protocol that solves fewer problems but offers faster data transport. Internet applications can choose the data transport protocol that makes the most sense for their application.

Skill 15.02: Explain User Datagram Protocol (UDP)

Skill 11.02 Concepts

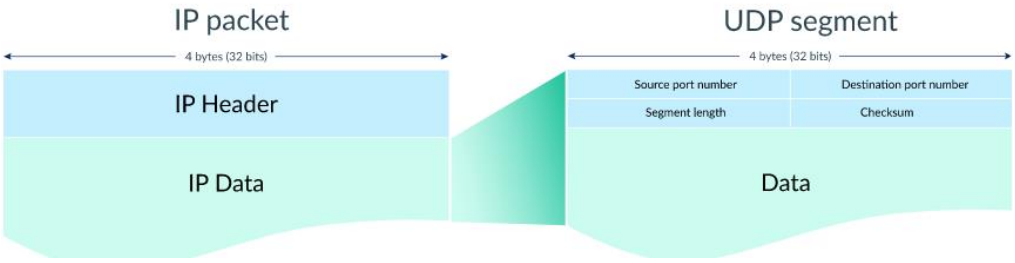
The **User Datagram Protocol (UDP)** is a lightweight data transport protocol that works on top of IP.

UDP provides a mechanism to detect corrupt data in packets, but it does *not* attempt to solve other problems that arise with packets, such as lost or out of order packets. That's why UDP is sometimes known as the ***Unreliable*** **Data Protocol**.

UDP is simple but fast, at least in comparison to other protocols that work over IP. It's often used for time-sensitive applications (such as real-time video streaming) where speed is more important than accuracy.

Packet Format

When sending packets using UDP, the header portion of each IP packet is formatted as a **UDP segment**.



Each UDP segment contains an 8-byte header and variable length data.

Port Numbers

The first four bytes of the UDP header store the port numbers for the source and destination.

A networked device can receive messages on different virtual ports, similar to how an ocean harbor can receive boats on different ports. The different ports help distinguish different types of network traffic.

Here's a listing of some ports in use by UDP on my laptop:

```
$ sudo lsof -i -n -P | grep UDP
launchd            1   IPv4    UDP *:137
launchd            1   IPv4    UDP *:138
syslogd           45   IPv4    UDP *:54465
mDNSResponder    186   IPv4    UDP *:5353
mDNSResponder    186   IPv6    UDP *:5353
mDNSResponder    186   IPv4    UDP *:65327
mDNSResponder    186   IPv6    UDP *:65327
mDNSResponder    186   IPv4    UDP *:55657
mDNSResponder    186   IPv6    UDP *:55657
Google         12306   IPv6    UDP *:5353
```

Each row starts with the name of the process that's using the port and ends with the protocol and port number.

Segment Length

The next two bytes of the UDP header store the length (in bytes) of the segment (including the header).

Two bytes is 16 bits, so the length can be as high as this binary number:

1111111111111111

In decimal, that's ($2^{16} - 1$) or 65,535. Thus, the maximum length of a UDP segment is 65,535 bytes.

Checksum

The final two bytes of the UDP header is the checksum, a field that's used by the sender and receiver to check for data corruption.

Before sending off the segment, the sender:

1. Computes the checksum based on the data in the segment.
2. Stores the computed checksum in the field.

Upon receiving the segment, the recipient:

1. Computes the checksum based on the received segment.
2. Compares the checksums to each other. If the checksums aren't equal, it knows the data was corrupted.

To understand how a checksum can detect corrupted data, let's follow the process to compute a checksum for a very short string of data: "Hola".

First, the sender would encode "Hola" into binary somehow. The following encoding uses the ASCII/UTF-8 encoding:

H	o	l	a
01001000	01101111	01101100	01100001

Next, the sender segments the bytes into 2-byte (16-bit) binary numbers:

0100100001101111
0110110001100001

To compute the checksum, the sender adds up the 16-bit binary numbers:

```
  0100100001101111
+0110110001100001
  1011010011010000
```

The computer can now send a UDP segment with the encoded "Hola" as the data and 1011010011010000 as the checksum.

The entire UDP segment could look like this:

Field	Value
Source port number	00010101 00001001
Destination port number	0001010 100001001
Length	00000000 00000100
Checksum	10110100 11010000
Data	01001000 01101111 01101100 01100001

What if the data got corrupted from "Hola" to "Mola" on the way?

First let's see what the corrupted data would look like in binary.

"Mola" encoded into binary...

M	o	l	a
01001101	01101111	01101100	01100001

...and then segmented into 16-bit numbers:

0100110101101111
0110110001100001

Now let's see what checksum the recipient would compute:

$$\begin{array}{r} 0100110101101111 \\ +0110110001100001 \\ \hline 1011100111010000 \end{array}$$

The recipient can now programmatically compare the checksum they received in the UDP segment with the checksum they just computed:

- Received: 1011010011010000
- Computed: 1011100111010000

Do you see the difference?

When the recipient discovers that the two checksums are different, it knows that the data was corrupted somehow along the way. Unfortunately, the recipient can *not* use the computed checksum to reconstruct the original data, so it will likely just discard the packet entirely.

The actual UDP checksum computation process includes a few more steps than shown here, but this is the general process of how we can use checksums to detect corrupted data.

Skill 15.02 Exercises 1 & 2

Skill 15.03: Explain Transmission Control Protocol (TCP)

Skill 15.03 Concepts

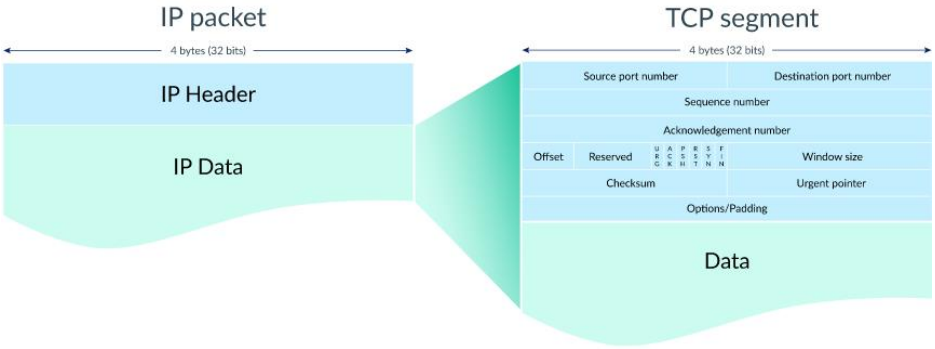
The **Transmission Control Protocol (TCP)** is a transport protocol that is used on top of IP to ensure reliable transmission of packets.

TCP includes mechanisms to solve many of the problems that arise from packet-based messaging, such as lost packets, out of order packets, duplicate packets, and corrupted packets.

Since TCP is the protocol used most commonly on top of IP, the Internet protocol stack is sometimes referred to as **TCP/IP**.

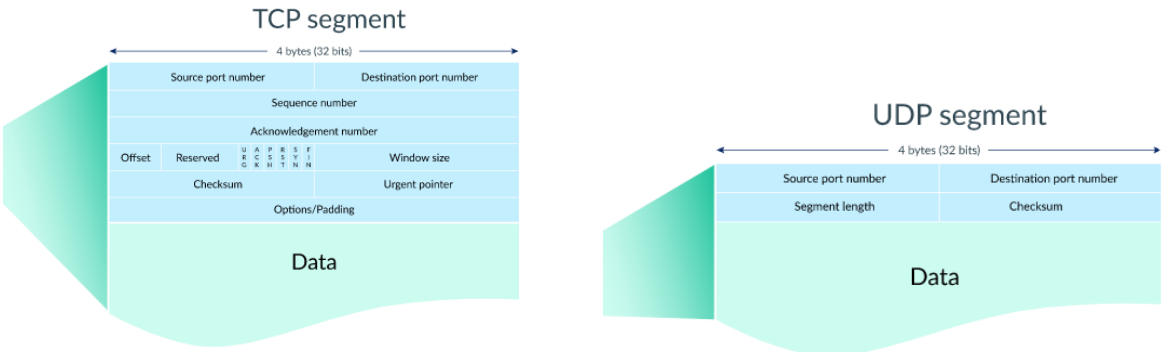
Packet Format

When sending packets using TCP/IP, the data portion of each IP packet is formatted as a **TCP segment**.



Each TCP segment contains a header and data. The TCP header contains many more fields than the UDP header and can range in size from 20 to 60 bytes, depending on the size of the options field.

The TCP header shares some fields with the UDP header: source port number, destination port number, and checksum.

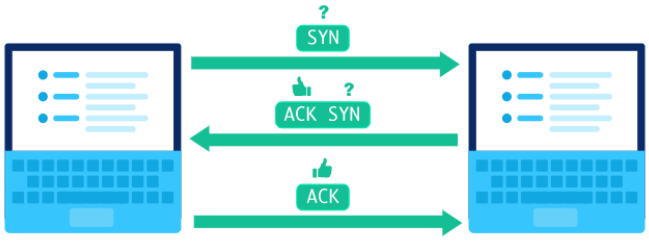


From Start to Finish

Let's step through the process of transmitting a packet with TCP/IP.

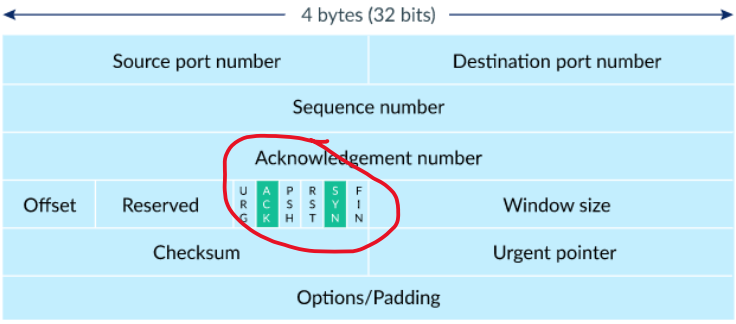
Step 1: Establish connection

When two computers want to send data to each other over TCP, they first need to establish a connection using a **three-way handshake**.



The first computer sends a packet with the SYN bit set to 1 (SYN = "synchronize?"). The second computer sends back a packet with the ACK bit set to 1 (ACK = "acknowledge!") plus the SYN bit set to 1. The first computer replies back with an ACK.

The SYN and ACK bits are both part of the TCP header:



The ACK and SYN bits are highlighted on the fourth row of the header.

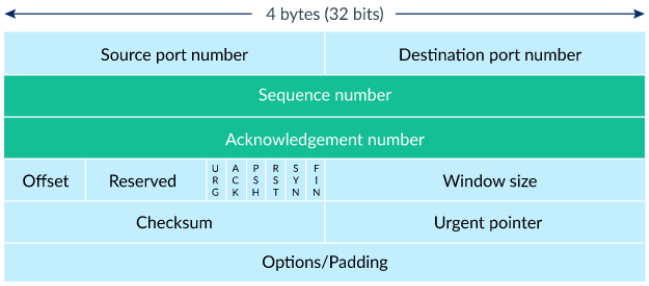
In fact, the three packets involved in the three-way handshake do not typically include any data. Once the computers are done with the handshake, they're ready to receive packets containing actual data.

Step 2: Send packets of data

When a packet of data is sent over TCP, the recipient must always acknowledge what they received.

The first computer sends a packet with data and a sequence number. The second computer acknowledges it by setting the ACK bit and increasing the acknowledgement number by the length of the received data.

The sequence and acknowledgement numbers are part of the TCP header:

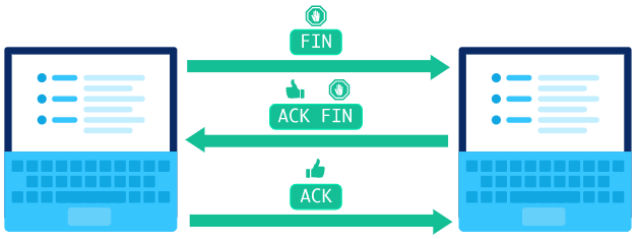


The 32-bit sequence and acknowledgement numbers are highlighted.

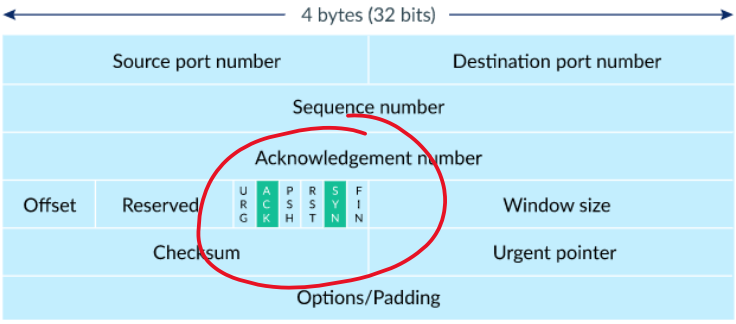
Those two numbers help the computers to keep track of which data was successfully received, which data was lost, and which data was accidentally sent twice.

Step 3: Close the connection

Either computer can close the connection when they no longer want to send or receive data.



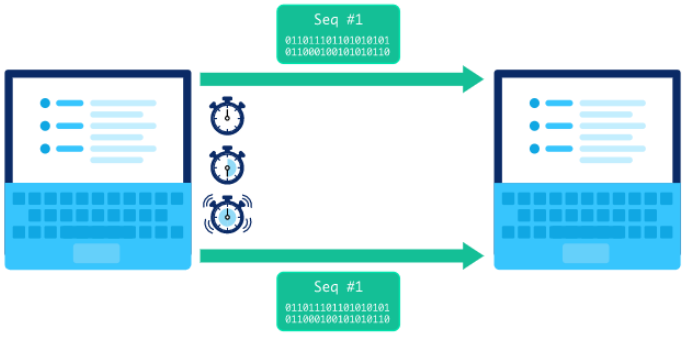
A computer initiates closing the connection by sending a packet with the FIN bit set to 1 (FIN = finish). The other computer replies with an ACK and another FIN. After one more ACK from the initiating computer, the connection is closed



The ACK and SYN bits are highlighted on the fourth row of the header.

Detecting Lost Packets

TCP connections can detect lost packets using a timeout.



After sending off a packet, the sender starts a timer and puts the packet in a retransmission queue. If the timer runs out and the sender has not yet received an ACK from the recipient, it sends the packet again.

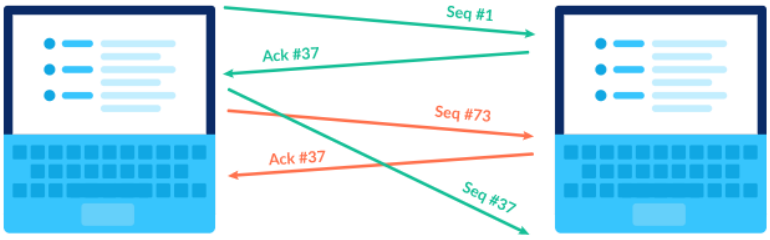
The retransmission may lead to the recipient receiving duplicate packets, if a packet was not actually lost but just very slow to arrive or be acknowledged. If so, the recipient can simply discard duplicate packets. It's better to have the data twice than not at all!

Handling out of order packets

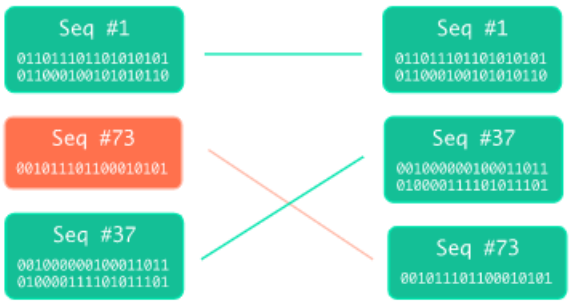
TCP connections can detect out of order packets by using the sequence and acknowledgement numbers.

When the recipient sees a higher sequence number than what they have acknowledged so far, they know that they are missing at least one packet in between. In the situation pictured above, the recipient sees a sequence number of #73 but expected a sequence number of #37. The recipient lets the sender know there's something amiss by sending a packet with an acknowledgement number set to the expected sequence number.

Sometimes the missing packet is simply taking a slower route through the Internet and it arrives soon after.



Other times, the missing packet may actually be a lost packet and the sender must retransmit the packet.
In both situations, the recipient has to deal with out of order packets. Fortunately, the recipient can use the sequence numbers to reassemble the packet data in the correct order.



Skill 15.03 Exercises 1 thru 4