# 50005 - Networks and Communications - Lecture 4

Oliver Killane

14/02/22

Lecture Recording

Lecture recording is available here

# The Transport Layer

Provides connection (**TCP**) and connection-less (**UDP**) services to allow communication between **end-systems/hosts**.

- Connection decision is made at this level.
- Only runs on **end hosts**, not on **routers** or **switches**.
- Requires the lower layers in order to operate (Network, Data-link and Physical).
- Protocols in this layer work on the assumption that the lower levels are working, however must consider that **IP** is best effort, and gives no guarantees on data integrity or order of delivery for packets.

Other layer-4/Transport Layer protocols include:

**QUIC** A **UDP** based transport layer designed by google employees to replace **TCP** using multiple multiplexed connections using **UDP**, focused on improving **HTTP** performance. The wikipedia article goes into more detail.

**UDP-Lite** A **UDP** like connectionless protocol that allows potentially damaged data payloads to be propagated to the application layer, and hence allows the application layer to discern data integrity and act accordingly (Wikipedia article).

**DCCP** The **Datagram Congestion Control Protocol** is a message-oriented protocol that uses reliable connection setup, close and has explicit congestion notification (Wikipedia article).

**SCTP** The **Stream Control Transmission Protocol** is a message-oriented protocol based on **UDP** (Wikipedia article).

**RSVP** The **Resource Eservation Protocol** is used to reserve network resources to ensure quality of service (Wikipedia article).

## Terminology

| **Layer** | **No** | **Data Name** |
|---|---|---|
| Application | 5 | Data |
| Transport | 4 | TCP Segments (created by segmentation) of UDP Datagrams |
| Network/Internet | 3 | IP Datagrams (a.k.a. packets) (created by fragmentation) |
| Data Link | 2 | Frames |
| Physical | 1 | Bits |

## Port Numbers

**Definition: Ports**

Used to connect applications together/ separate different application's connections.

The transport layer uses port numbers to differentiate between many different network communications. Each application on a host uses a unique port number.

Port numbers are cross-platform, meaning on many different devices, computer architectures and OSes they are the same for the same types of applications (e.g HTTP, IMAP).

| Ports | | | Use |
|---|---|---|---|
| 0 | $\to$ | 1023 | (well known/reserved for certain protocols, e.g HTTP $\to$ 80, SMTP $\to$ 25, SSH $\to$ 22) |
| 1024 | $\to$ | 49151 | (for any user application to use or register) |
| 49152 | $\to$ | 65535 | (dynamic/ephemeral/private) and are used by clients temporarily) |

# TCP

**Definition: Tranmission Control Protocol (TCP)**

A connection-oriented transport layer protocol.

- Data is split into **segments**.
- Reliable data transfer (integrity of data and (possibly) ordered delivery)
- Not secure (other mechanisms need to be used to ensure security)
- Can offer stream connections (ordered delivery, only accept segments in order, e.g received 4, waiting for 5, but received $6, 7$, ignore 6 and 7 until 5 is received.)
- Congestion Control (avoids destructive congestion on the network)
- Requires A handshake to start the connection.
- **Full-Duplex** so both sides can send and receive at the same time.

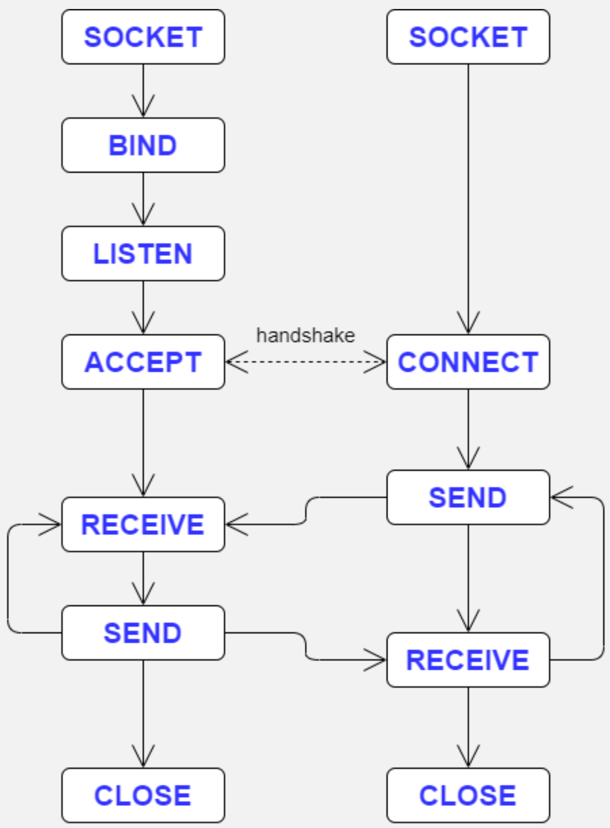
To identify a socket connection we use the **IP Address**, port number and protocol (**TCP/UDP**).

**Definition: Berkely Socket Interface**

An interface adpoted by all **UNIX** systems and windows.

| | |
|---|---|
| **1. SOCKET** | Create a new communication endpoint. |
| **2. BIND** | Attach a local address to the socket. The client and server both bind a transport level address and name to the locally created socket. |
| **3. LISTEN** | Prepare for / Annouce ability to accept, $n$ connections. The kernel now waits for connections from clients. |
| **4. ACCEPT** | Block until some remove client wants to establish a connection, hence the server can now wait, receive a request and choose to accept or deny a connection. |
| **5. CONNECT** | Attempt to establish a connection. When a client connects it must provide the full transport-level address to locate the socket. |
| **6. SEND** | Send data over the connection. |
| **7. RECEIVE** | Receive data over a connection. |
| **8. CLOSE** | Release the connection, communication ends when the socket is closed. |

A connection-oriented example:




In a connection-less scenario, **LISTEN**, **ACCEPT** and **CONNECT** are not required.

Example: Simple Java Web Client

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;

public class exampleTCPClient {
    // connect to localhost (this machine) on port 2251
    static final int port = 2251;
    static final String ip = "127.0.0.1";

    public static void main(String[] args) throws UnknownHostException, IOException {
        // Create a socket and implicitly connect.
        Socket socket = new Socket(ip, port);

        // create reader and writer for sending and receiving
        BufferedReader receive = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter send = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true);

        // send a message from console input
        send.println(System.console().readLine());

        // print a received message from the socket to the console
        System.out.println(receive.readLine());

        // close the socket
        socket.close();
    }
}
```

Example: Simple Java Web Server

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class exampleTCPServer {
    static final int port = 2251;

    public static void main(String[] args) throws IOException {
        // Bind and set socket to listen.
        ServerSocket serverSocket = new ServerSocket(port);

        // status message
        System.out.println("Started listening on port " + port);

        while (true) {
            // accept a new connection and create socket to handle connection.
            Socket socket = serverSocket.accept();

            // create reader and writer for sending and receiving
            BufferedReader receive = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter send = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true);

            // send a message from console input
            send.println(System.console().readLine());

            // print a received message from the socket to the console
            System.out.println(receive.readLine());

            // close the socket
            socket.close();
        }
    }
}
```

To handle many clients, a thread must be created per client, rather than the basic forever loop as above.

## Segments

Definition: TCP Segment

A wrapper for **TCP** data, transmitted within the Network Layer protocol (e.g **IPv4** or **IPv6**)

Definition: Maximum Segment Size (MSS)

The maximum amount of application data transmitted in a single segment (header size is not included).

Usually related to the **MTU** of the connection to avoid network level fragmentation (splitting segments in the network layer into multiple packets).

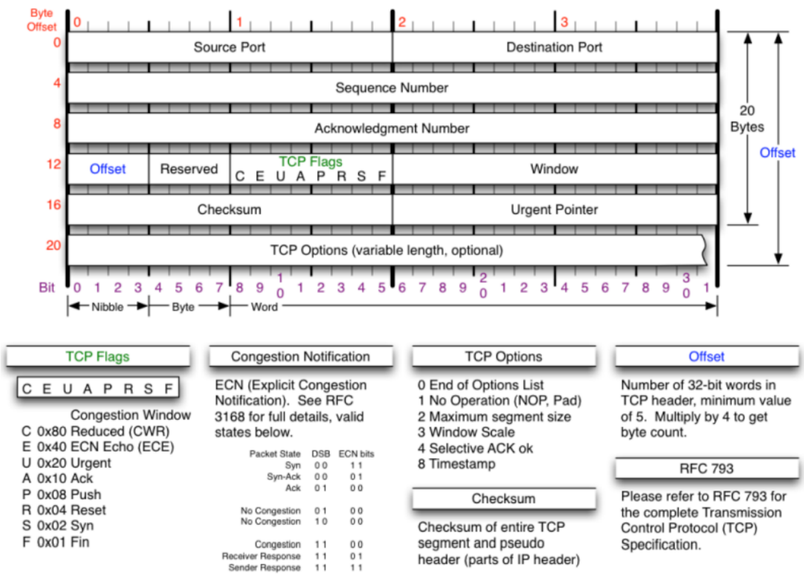
Definition: Maximum Transmission Unit (MTU)

The largest link layer frame available to the sender. Consider it as the largest unit of data that can be transmitted through all links to the receiver without requiring it to be split.

**Path MTU Discovery** determines the largest frame that can be sent on all links from the sender to receiver.

## TCP Header

From the NMap book:




Points of note:

- Source and destination ports are 16 bit identifiers.
- Sequence number and Acknowledgement Number (32 bits) is used for reliable data transfer (identifies a segment, so any segments missing in the sequence can be detected)
- Receive window (16 bits), the amount of data that can be sent before an acknowledgement is received (if the receiver cannot process data as fast as it arrives, it will ask to reduce the TCP window), more here.
- Header length determines the size of the **TCP** header in 32 bit words.

- The optional/variable length field is used to negotiate protocol parameters such as window scale, or **maximum segment size**.

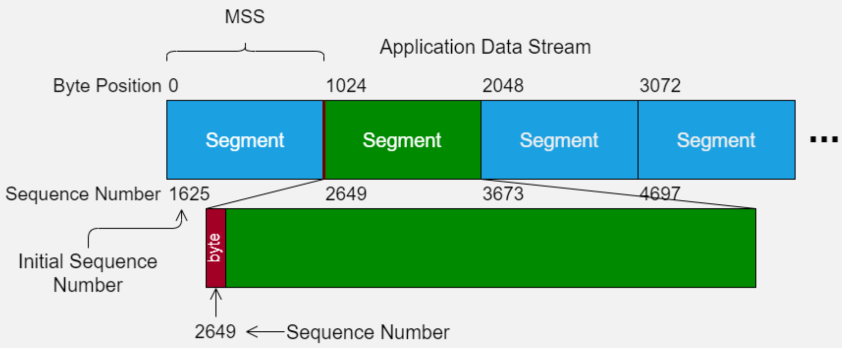
There are several header Fields:

| Field | Bits | Description |
|---|---|---|
| **URG** | 1 | Signals the data as urgent, location of urgent data marked by urgent data pointer field. Note that some software will ignore it. |
| **ACK** | 1 | Signals that the acknowledgement number is a valid acknowledgement. |
| **PSH** | 1 | Push flag, asks the receiver to push data to the application immediately. |
| **RST** | 1 | Resets the connection, often used to shutdown a connection when some unexpected error occurs. |
| **SYN** | 1 | Synchronisation flag, used as part of the handshake. |
| **FIN** | 1 | Signals connection to finish/shutdown. |
| Checksum | 16 | Used for error detection. |

**Definition: Sequence Number**

Each byte in the data stream has a sequence number (byte, **not** segment).

The sequence number in a **TCP segment** indicates the position of the first byte carried by that segment.



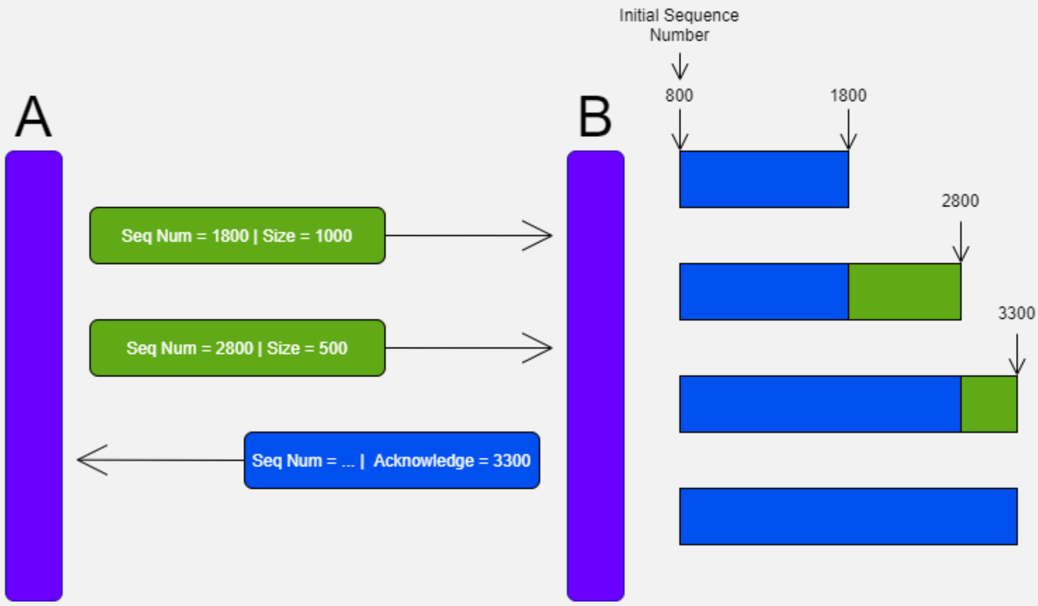

When the **TCP** connection is setup, a random **Initial Sequence Number** is decided upon to avoid any leftover segments being received by mistake.

Hence when creating a new connections, even with the same data the sequence numbers will be different.

**Definition: Acknowledgement Number**

An **acknowledgedment number** represents the end of the data received, or the first sequence number of the data waiting to be received.

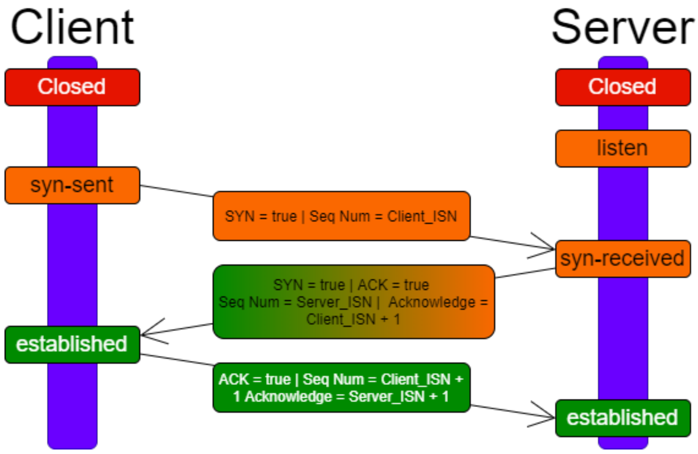
- TCP acknowledgements can be cumulative (recieve segments 1, 2, 3, ackenowledge (wait for) 4).
- Typically acknowledge every other packet.




- As **TCP** is full duplex, multiple streams/sequences can be received, and acknowledged at the same time.

## 3-Way Handshake

1. Client sends a **TCP segment** with the **SYN** flag set to **true**, and the **initial sequence number**.
2. Server responds with another **SYN** **TCP segment** which also has **ACK** as **true** and the first unseen client Sequence number.
3. Client responds with an **ACK** with first unseen server **sequence number**, and a new **sequence number**.




Connection termination is similar, but uses **FIN**.

# UDP

**Definition: User Datagram Protocol (UDP)**

A connection-less transport layer protocol.

- Data is split into **datagrams** (think like telegrams).
- **Datagrams** cannot be larger than $65,507$ bytes ($20B$ IP Header $+8B$ UDP Header $+$ $65,507B = 65,535B$ which is the maximum IP packet size).
- In practice smaller $500B \rightarrow 1KB$ datagrams are used to increase the proportion of packets that are intact (any small error only effects a small datagram, does not invalidate a large datagram).
- Application identification is provided (multiplexing/demultiplexing).
- Integrity of data is checked by a **CRC**-type checksum.

**UDP** is very simple:

- no flow Control
- no error Control
- no retransmissions

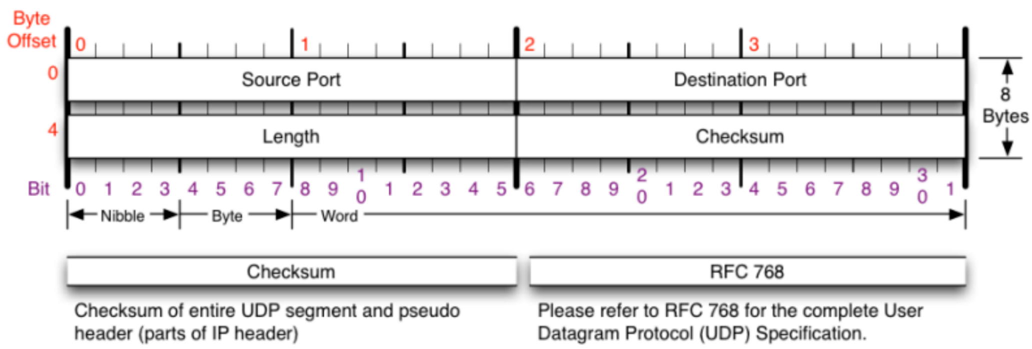
Why use **UDP**?

- Finer level of application layer control over when and what data is sent (e.g real-time such as skype).
- No connection needs to be established (faster than TCP).
- No connection state needs to be stored.
- Very small packet overhead (only a small bit of the packet is not payload).

It is also very useful in **client-server** interactions.

A client can send a short message to a server, and get a quick response, on failure can time out or try again. The resulting code is simple and fewer messages are needed (no connection setup/teardown).

## UDP Header



Example: Simple Java Web Client

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class exampleUDPClient {
    // connect to localhost (this machine) on port 2251
    static final int port = 2251;
    static final String ip = "127.0.0.1";

    public static void main(String[] args) throws IOException {
        // Create a buffer to store data to send & receive, place contents from
        // the console.
        byte buffer[] = System.console().readLine().getBytes();

        // Create a new packet sourced from the buffer to the target ip and port.
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, InetAddress.getByName(ip), port);

        // Create a datagram socket to send & receive packets
        DatagramSocket socket = new DatagramSocket();

        // send the packet, the ip and port and inside the packet.
        socket.send(packet);

        // reallocate the buffer to take the response packet
        buffer = new byte[256];

        // take the response from the socket and store in buffer.
        packet = new DatagramPacket(buffer, buffer.length);
        socket.receive(packet);

        // print the received data to the console.
        System.out.println(new String(packet.getData()));

        // Socket no longer used, so close.
        socket.close();
    }
}
```

Example: Simple Java Web Server

```java
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class exampleUDPServer {
    static final int port = 2251;

    public static void main(String[] args) throws IOException {
        // Create a socket to receive datagrams on.
        DatagramSocket socket = new DatagramSocket(port);

        // Server runs in forever loop to deal with packets.
        while (true) {
            // Allocate a buffer to store packet data
            byte buffer[] = new byte[256];

            // Create a packet to use buffer.
            DatagramPacket packet = new DatagramPacket(buffer, buffer.
                ↪ length);

            // Receive data, write to buffer.
            socket.receive(packet);

            // Print the data received to the console.
            System.out.println(new String(packet.getData(), 0, packet.
                ↪ getLength()));

            // Take response from standard input.
            buffer = System.console().readLine().getBytes();

            // Get the address of the sender from the received packet.
            InetAddress clientAddress = packet.getAddress();
            int clientPort = packet.getPort();

            // Create a new packet from the buffer.
            packet = new DatagramPacket(buffer, buffer.length,
                ↪ clientAddress, clientPort);

            // Send the response.
            socket.send(packet);
        }
    }
}
```

# TCP vs UDP

- **(UDP) - A PvP Game sending short bursts of data to players**
  Data transmission is time critical, if a message is lost, we can simply recover our own way (e.g list of messages, retry).

  We get to control the implementation. While we may mimic TCP in some error recovery, we can decide what features we want and don't for the best experience.

- **(TCP) - An online card game**
  Speed is not a concern, **TCP** is just fine.
- **(TCP) - Movie player application**
  We want it to be fast, however we do not want to drop frames.
  - Pre-Buffer the video, and constantly use connection to get next few seconds as the movie plays.
  - TCP manages errors to reduce dropped frames.

## Data Transfer

**Lecture Recording**

Lecture recording is available here

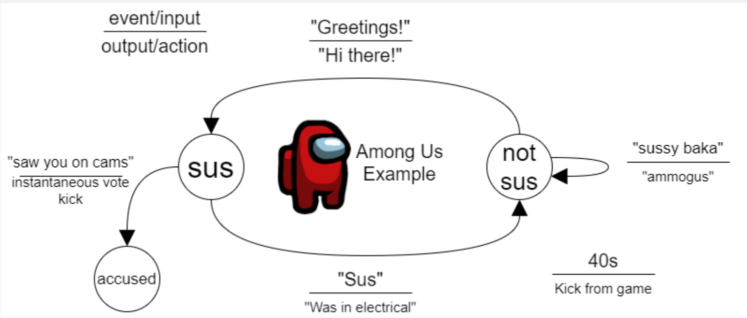
**Definition: (FSM) Finite State Machine**

A mathematical abstraction used among other uses, to describe network protocols.

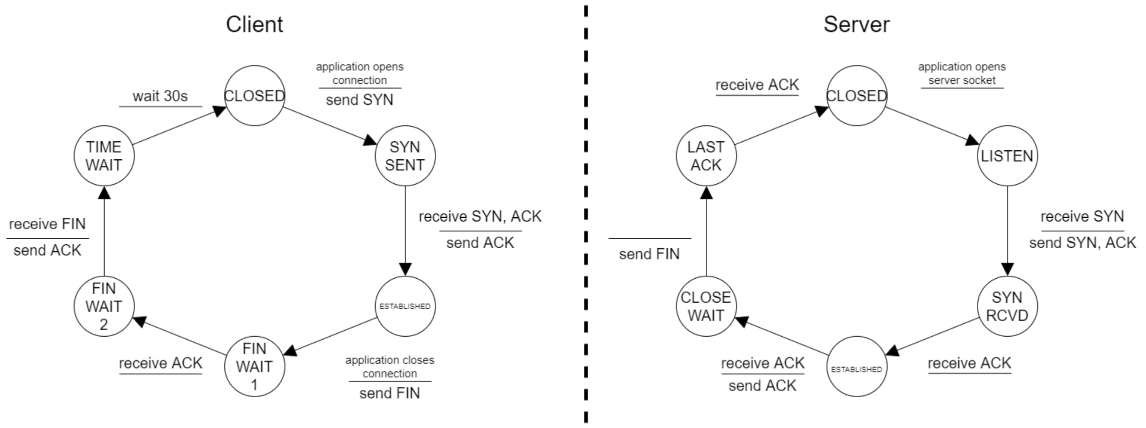
| | |
|---|---|
| FSM | Finite State Machine |
| FSA | Finite State Automata |
| DFA | Deterministic finite-state automaton |
| NFA | None-deterministic finite state automaton |

We can describe transitions between states for a protocol by the event and action.

**Example: Among Us Basic Group Meeting**



## TCP FSM




**View TCP states on your device**

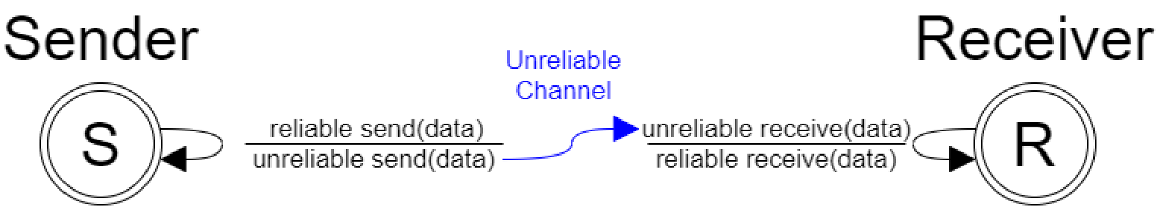
On windows can use the **netstat -a** commend, tcpview or currPorts.

On linux there is htop and iptraf.

## Data Transfer FSM

**TCP** provides mechanisms to ensure reliability in data transfer. While **IP** in the **Network Layer** is a best-effort protocol and is unreliable, by going through **TCP** we can create a reliable connection.

we can generalise this as:




## Error Detection

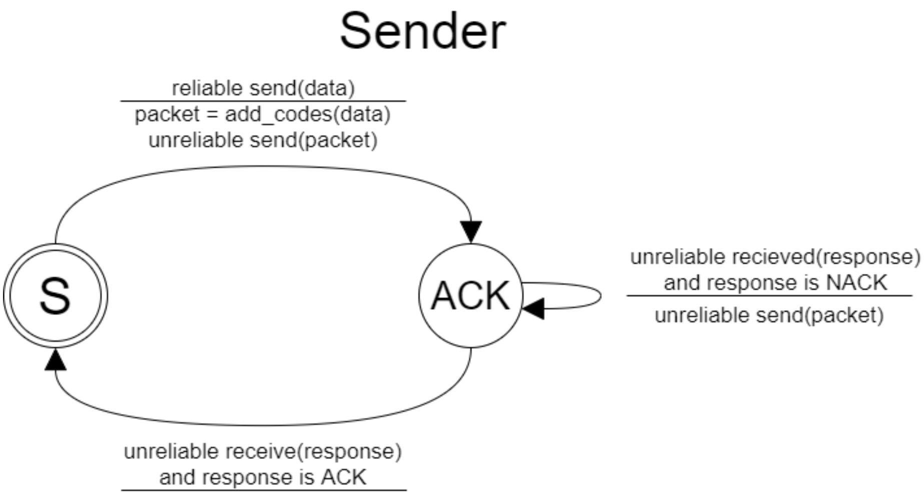
Bits may be flipped in transmission (due to noise/interference and imperfect physical hardware).

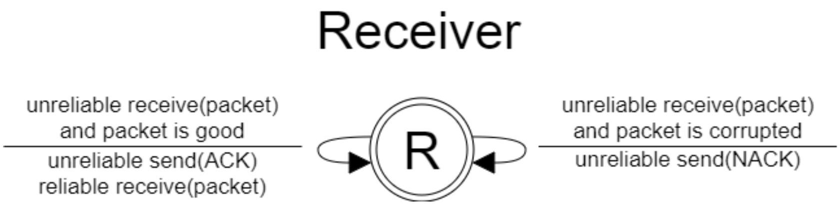
| | |
|---|---|
| **Error Detection** | Receiver must be able to check if packet is corrupted. |
| **Receiver Feedback** | Receiver must be able to tell sender the packet sent was corrupted. |

> **Definition: Parity Bit**
>
> A very simple type of error dectection code, where a single bit is set based on the parity of the rest of the packet. Typically this is the **XOR** of all the bits.
>
> $$\begin{aligned} 1001 &\rightarrow 1001\ 0 \\ 1101 &\rightarrow 1101\ 1 \end{aligned}$$

**Stop and Wait with Error detection**

We can express the data transfer **FSM** to include error detection. In this setup the protocol is synchronous, meaning for each segment the sender must receive back an acknowledgement before the next segment is sent.




The main issue with this approach, is that **ACK**s and **NACK**s can also get corrupted. If we use the same scheme to reliably transfer the **ACK**s and **NACK**s we end up with potentially no termination (sender and receiver stuck in a loop of replying with **NACK**s at each other, that the other does not receive uncorrupted on a noisy channel).

**Assume NACK and retransmit**

We can add a sequence number to each packet, so that packets can be retransmitted, and the receiver knows which packets are retransmissions.

If we use **stop and wait** we only need 1 bit for the sequence number, as 0 is original, 1 is retransmission.

### Use Sequence Numbers instead of NACKs

We can instead use sequence numbers. If the packet is not acknowledged then an **ACK** is not sent. **ACK** is sent for the last good packet, hence the receiver can use a lack of **ACK**s to determine that it must retransmit some data.

Note that with TCP the **ACK** contains the start of the packet to be sent (or resent) next/ the end byte sequence number of the data received.

### Out of Order Sequence Numbers

Rather than use stop and wait, a sender may send many packets.

- **Delayed ACK**
  The receiver only accepts in order, so ignores any packets sent out of order. After receiving a packet in order, it waits some time (e.g 500$ms$) before sending the **ACK**, allowing for more in order-packets to be received in this time (resetting the wait).

  1. Received 0, start wait
  2. Wait interrupted, received 1, start wait again
  3. Wait interrupted, received 2, start wait again
  4. Received 7, ignored
  5. Received 6, ignored
  6. Wait from (3.) over, send **ACK** (have recieved up to 2, please send me 3)

     ...
- **Cumulative ACK**
  Received an in-order segment with the expected number, waiting on the next segment.

  Immediately send a cumulative **ACK**.
- **Duplicate ACK**
  Send an **ACK** for the next segment. Then an out-of-order segment arrived with a higher than expected sequence number, there is a gap.

  Immediately send another (duplicate) **ACK**.
- **Immediate ACK**
  A segment received partially or completely fills a gap in the received data.

  Immediately send an **ACK** for the lower end of the gap (to fill).

### Timeouts

We can set a timeout for receiving **ACK**s. When the sender does not receive an acknowledgment within the time, it assumes the packet was not received and can try again (retransmit).

- If the timeout is too long, when packets are lost retransmission will have to wait a long time and hence slow down the connection.
- If the timeout is too short, packets may be needlessly retransmitted.

### TCP & Checksums

Used by **TCP**. A checksum is calculated from the payload data.

- When received, if the checksum does not match the recalculated checksum, then corruption has occurred.
- Retransmission allows for error recovery.
- **ACK**s and **NACK**s are also protected by error detection code.
- Corrupted **ACK**s are used as **NACK**s
- Sequence numbers allow the receiver to ignore duplicate segments.

## Network Simulation

We can use network simulation to check, test and optimise parameters for protocols and network designs.

- **Cisco Packet Tracer**
  Packet tracer is a lightweight network simulator with a user friendly GUI.
- **GNS3 Network Emulator**
  GNS3 is network simulation tool available for free.
- **Opnet Modeler**
  Opnet (now riverbed) modeller is a commercial (paid for) network simulation tool.

## Detecting Congestion

**Lecture Recording**

Lecture recording is available here

**Definition: Congestion**

So far we have roughly described the **TCP Reno** protocol. However there are many other variants to deal with congestion control.

Routers have a limit to how many packets they can route. Packets are held in a queue.

If too many packets are sent to one of the routers between a sender and reciever, its queue will overflow, resulting in some segments being dropped.

Hence the server assumes the network is congested when it detects segment loss from:

- timeouts (no **ACK** received)
- multiple **ACK**s (or equivalent acknowledgements) can be considered a **NACK**

There are many different congestion control algorithms:

| Algorithm | Affects |
|---|---|
| TFRC | Sender, Receiver |
| RED | Router |
| CLAMP | Router, Receiver |
| XCP | Sender, Router, Receiver |
| VCP | Sender, Router, Receiver |
| MaxNet | Sender, Router, Receiver |
| JetMax | Sender, Router, Receiver |
| ECN | Sender, Router, Receiver |
| Vegas | Sender |
| High Speed | Sender |
| BIC | Sender |
| CUBIC | Sender |
| H-TCP | Sender |
| FAST | Sender |
| Compound TCP | Sender |
| Westwood | Sender |
| Jersey | Sender |
| BBR | Sender |

- **Linux**
  Usually CUBIC, but can be found at `/proc/sys/net` ↪ `/ipv4/tcp_congestion_control`.
- **Windows**
  Can be found at `netsh interface tcp>sh gl`, if nothing then it is using the windows default.
- **Custom**
  It is possible to force any socket to use any variant
- **Characteristics**
  Most have variable characteristics combined. For example:

| | |
|---|---|
| Tahoe | Slow start, AIMD, Fast Retransmit |
| Reno | Fast Recovery |
| Vegas | Congestion Avoidance |

**Definition: TCP Vegas**

A popular **TCP** implementation.

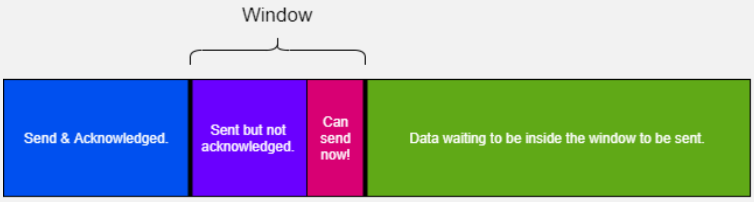
- Attempts to detect congestion before losses occur.
- Predicts packet loss using **RTT** (round trip time)
- Larger **RTT** $\Rightarrow$ greater congestion

**Definition: TCP CUBIC**

Used by linux as the standard.

In order to avoid advantaging smaller **RTT**s (as can happen with **TCP Reno**), grows **window** as a function of time rather than **RTT**

Definition: Congestion Window




The **congestion window** is the number of bytes that can be sent before blocking to wait for acknowledgements.

Both the sender and receiver can define the window size, the size used is the minimum of both.

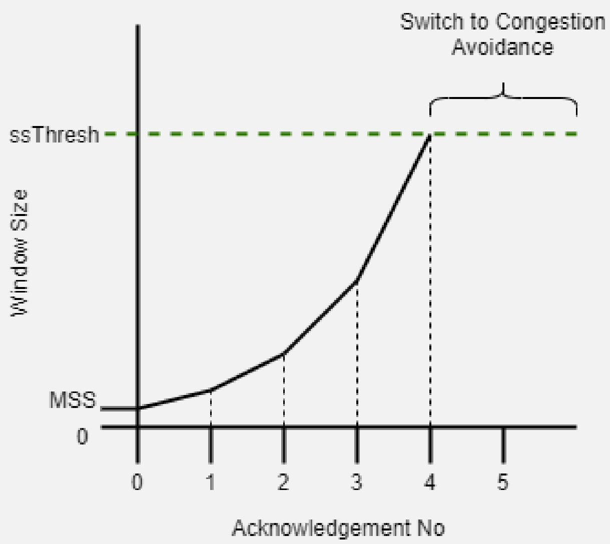
$$W = min(\text{Congestion Window}, \text{Receiver Window})$$

Hence with a given $RTT$ and window size $W$:

$$\text{maximum rate} \ \ \lambda \approx \frac{W}{RTT}$$

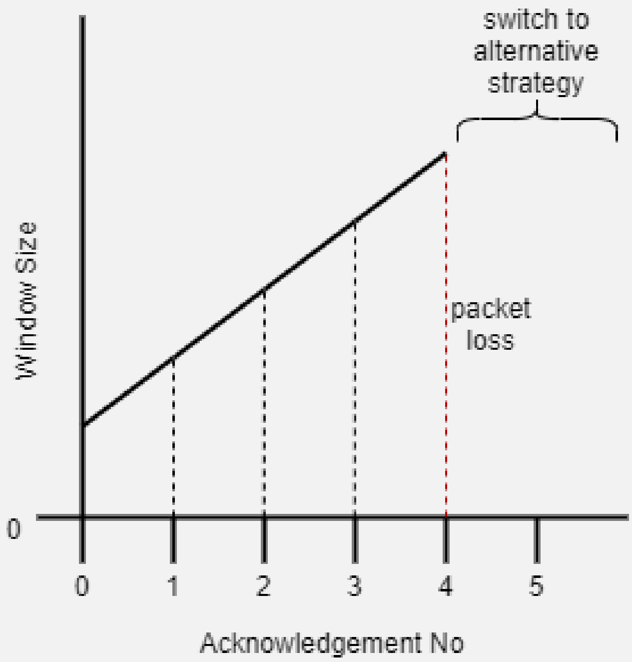
## Congestion Methods

Definition: Slow Start




1. Set initial window size to **MSS** (maximum segment size) (quite small for high-speed networks).
2. For every good acknowledgement, increase the window size by the size of data acknowledged (meaning window size is roughly doubled every **RTT**).
3. Continue this exponential increase until window size reaches the **ssthresh** (segment size threshold).
4. The use **Congestion Avoidance**.

Definition: Congestion Avoidance



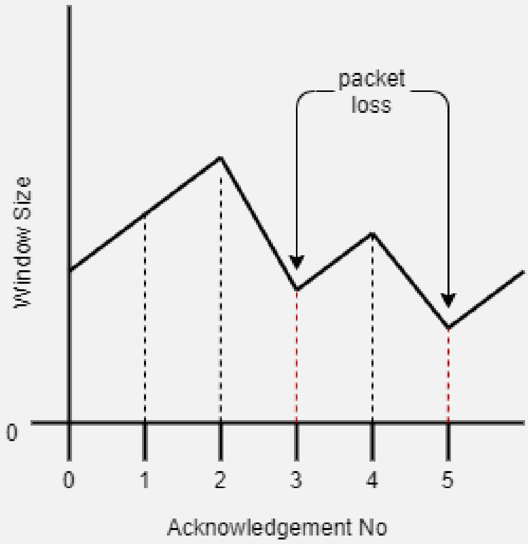

The window size is increased roughly linearly ($\approx 1$ $MSS$ per $RTT$).

For each good acknowledgement:

$$W = W + \frac{MSS^2}{W}$$

When congestion is detected (packet loss) switch to a different strategy.

**Definition: (AIMD) Additive Increase / Multiplicative Decrease**




- For every good acknoledgement: $W = W + \frac{MSS^2}{W}$
- For every packet loss event: $W = \frac{W}{2}$

**Definition: timeout**

We need to detect packet loss, when no **ACK** is send back.

- **timeout interval** $T$ must be larger than the $RTT$ otherwise we will retransmit data unnecessarily.
- $T$ cannot be too large, otherwise it will be slow to retransmit.
- **TCP** continuously estimates the $RTT$.
- **TCP** sets $T$ using the **smoothed RTT** ($SRTT$) and the RTT Variation $RTTVAR$ (exact computation can be found in section 2.2 & 2.3 here.)

$$T = SRTT + 4 \times RTTVAR$$

**Definition: Fast Retransmission**

Three duplicate **ACK**s are interpreted as a **NACK**. The number 3 is agreed upon in section 3 here as a tradeoff between fast retransmission and unnecessary premature retransmission.

- timeout suggests congestion
- 3 duplicate **ACK**s suggests the network can still transmit,

**Definition: Fast Recovery**

Given the current window size is $\overline{W}$:

If **timeout** occurs:

1. $W = MSS$
2. Run slow start until $W = \frac{\overline{W}}{2} = ssthresh$.
3. Switch to collision avoidance.

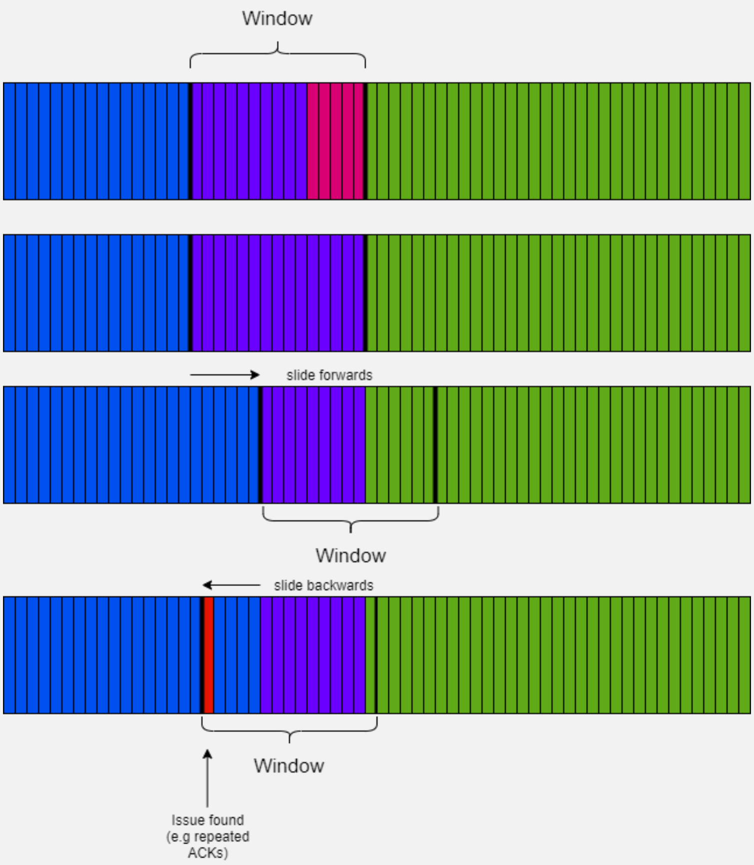
If it is 3 duplicate **ACK**s (a **NACK**) then run **Fast Recovery**:

1. $W = \frac{\overline{W}}{2}$
2. Switch to collision avoidance.

Fast recovery is *fast* as the window size is not reset all the way back to $MSS$, so can ramp up the window size more quickly.

## Window Strategies

**Definition: Sliding Window - Go Back $N$**




Sender transmits multiple segments without waiting for acknowledgement.

- The sender can have up to W bytes of unacknowledged segments in its pipeline.
- Sender's state is a queue of acknoledgements.
- When we receive some acknoledgements, we can move the slide the window along.

**Definition: Sliding Window - Selective Repeat**

Sender only re-transmits those segments it suspects were dropped or corrupted.

- Sender keeps a list/vector of acknowledgements.
- Receiver keeps a list/vector of acknowledged segments.
- When segments received out of order, they are kept to be added into the data once the missing/gap segments arrive.
- Sender keeps a timer for each segment it is waiting for acknowledgement of, resending only when the timer expires.
- Sender slides the window when the lowest pending segment is acknowledged.

**Definition: Flow Control**

Flow control attempts to prevent the receiver from being overwhelmed/overflowing (rate of sending is too high for it to cope).

- The receiver sends the **RecieverWindow** size along with acknowledgements.
- This typically is the size of buffer left to fill.
- When a buffer is full and the receiver can take no more, it sends an acknowledgement with **RecieverWindow** set to 0, and repeats a 1-byte ping to the sender to indicate it is not down or deadlocked, but rather just processing.

# Wireless TCP

**TCP** was designed before the popularisation of wireless networks.

**Wired Network**

When packets are lost, this indicated congestion.

- Reduce packets sent.
- Use congestion avoidance and recovery strategies.

**Wireless Network**

When packets are lost, this is most likely a channel reliability issue.

- Resend packets as much as possible.
- Gives best chance of one getting received correctly.

We can fix these conflicting requirements in two ways:

- **Split TCP Connections** If we use separate connections for wired and wireless we can distinguish between the two and hence use different algorithms for congestion avoidance.
- **Use Base Station** Have the wired base station do some retransmissions without informing the wireless source.

  Here the base station tries to improve wireless IP reliability using **TCP**.

# Network Usage

$$\text{Utilisation Factor} = \frac{\text{network use}}{\text{maximum theoretical usage}}$$

When we have the $RTT$, packet size $L$ and transmission rate $R$, we can also use the time on the connection used out of the possible time length:

$$d_{trans} = \frac{L}{R}$$

$$\text{Utilisation Factor} = \frac{d_{trans}}{RTT + d_{trans}}$$