Sessions are important because in current network infrastructure schedules access to resources via Sessions. One session can span across several connections (in space or time). Session may last for a long time Cookie-based services, using many different connections (FTP, Torrent, VoIP).

For a long time there was a question of how many messages is needed to reliably establish session. Main task is to synchronize two hosts taking into consideration lost, connection delay, duplicated messages, nodes crash (losing connection state). We need to avoid that data from one session gets delivered in another session.

**Two way handshakes**: We have unidirectional Connection Setup and data and ACKs that have sequence numbers. Connection request is sent with sequence number, connection acknowledge sent back with same number. Then the number is increased for the sending data, and acknowledges just returns the same numbers that are received. This goes on as long as the session. Duplicate requests blocked because server knows that it already have a connection with that sequence number, and acknowledge for connection with a different sequence number is ignored. Problem appears if connection establishment is delayed too much and additional request is sent. Suppose that this request takes long time to get to the server. In the meantime first acknowledge arrives, and communication is established. After the communication is over, late additional request arrives and gets treated as completely new request.

Setting a **maximum packet life time**, addresses the issue, but requires a Timer and Memory on the server side. It is also not the best solution because if timer is set for too long, ignoring ids might influence on performance too much.

Other approach is to introduce 3 way handshake with unique ids on both sides of the communication. It is implemented as a double 2 way handshake, one for each side. One node sends request, other responds with acknowledge and request of its own. This goes on as long as the session. In theory this solves all the problems, but in practice it is hard to have unique sequence numbers.

Solution has been found as combination of last two approaches. It has timer and unique number. Timer relaxes requirement of eternally unique sequence number, and unique sequence number relaxes the need for having too long timers. Things can still go wrong but chance of that is very low. This solution is used today in TCP.

Resources in the network need to be shared. We have limited bandwidth but many users, so we use sharing to make them think they are all serviced at the same time by doing resource allocation in smart way. We have three aspects of scheduling: Method of access to a resource (Opportunistic, Token-based and Synchronized), Level of access (packet, flow and service type) and Distribution Metric (Load and Fairness). Scheduler needs to have means to measure its performance. Load – how much network is used?

// Synchronized – everybody gets equal amount of time even if they don’t use it completely

// Token-based- token gives access but excludes the others. What if token gets lost?

// Opportunistic-constant race for resources. We have some back of timer. At the long run users really do synchronize. Problem might be that someone abuses the link.

There are two metric to describe flow. **Throughput**: how much information gets transmitted and **delay**: how long to be delivered. **Power** of the network is combination of these two. Upper boundary is defined by link capacity and lower by queue length. Queue is what basically connects throughput and delay. Larger the queue grater is the throughput but delay is also grater. Increasing queue size increases performance of the network up to a point, beyond which the system thrashes. That is the point of **congestion** where scheduler cannot pass enough packet in time and they stay there so delay increases exponentially. Controlling the situation is unfortunately not that simple and we have two parallel directions for the solution: Improve scheduling so the boundary could be higher, and to control flow behavior near the boundary conditions at the protocol level.

**Fairness** is based on throughput and that is not really fair, because it is decided in one moment and not through time. If some flow lasts as long as three other flows and link is shared equally, based on throughput that is not really fair. Another way to view fairness is by type of service (ftp vs. voice) and finally in one session there can be many connections each competing separately where one user gets many times more throughput.

Optimization of scheduler is based on combining goals like fairness, efficiency, response time, turnaround and throughput. Some of these goals cannot be satisfied simultaneously!

In **preemptive scheduling**, a task can be suspended and the resource can be granted to another task opposed to old run-to-completion policy where access to the resource was maintained until the task is finished. In the network we have something in between. Scheduler can’t really suspend the flow, it can only suspend packet boundaries.

There are three types of schedule depending on relation of the follows. First is **Precedence process model** where we have ordered dependency between processes, so they need to be synchronized. Second is **Communication process model** where flows depend on each other but without specific order, so the processes can communicate asynchronously. Third one, **disjoint process model**, is used in IP where nothing about the processes is known to the scheduler.

For better use of disjoint process model, we have some **scheduling strategies**. Single queue, FIFO schedule (many packets of different flows get in one queue) with tail-drop policy (when queue is full last that got in is kicked out). Single queue, Shortest Job First: queue is ordered by length of the job (size of the packet). This way we get smaller average waiting time in the queue. Single queue, Earliest Deadline First: queue is ordered by job deadline, and tries to avoid timeouts. Here we have a problem because of different lengths of jobs. In practice it does work because of packet size boundaries. Multiple queues, Round-robin (fair) schedule: Each flow gets its own queue and the scheduler takes one packet from each queue in turns. Multiple queues, Priority (weighted fair) schedule: Each class of traffic (various premiums for customers) or each type of service gets its own queue. Each queue has priority (i.e. number of packet in turn). Problem is that with different sizes of packets fairness disappears. Flow that is supposed to get double throughput has smaller packets and doesn’t use it. For true fairness we need to decide on bit level and not from number of packets that can be different sizes. Deficit Round-Robin: every queue is given a credit of how many bits can it send. Size of the packet reflects that. Only the packet size is subtracted if queue is scheduled. If there is not enough credit for the current packet size queue needs to wait another turn and accumulate credit.

**TCP**: application sends byte stream to TCP that he packs them in segment. Segment size depends on current size of the TCP window. Over IP message is sent, and acknowledge needs to get back after some timeout. If for some packet it doesn’t, we retry sending just that packet, not the whole window.

**Congestion control**: Congestion window is used by the source to limit the number of packets in transit. We start communication with a cold start where we exponentially (2x) increase congestion window after each acknowledge, until a Congestion threshold is found (packet loss). Then we continue with Additive Increase and Multiplicative Decrease where we increase Congestion window by 1, every time an entire batch of Congestion window worth messages has been acknowledged and every time a timeout occurs, we divide Congestion window by 2. When a packet arrives at destination, it responds with acknowledge. If the packet is received out of order acknowledge is a duplicate of the last acknowledge. When the sender sees 3 duplicate acknowledges, it retransmits the assumed lost packet. After fast retransmit the sender halves the Congestion window, and starts recovery using additive increase. With this last technique we avoid long waiting period for sender to realize that packet was not delivered.

TCP has problems with congestion control if packet loss is not result of congestion. It can be because of unreliable links in wireless or taking a different path, or it is unreliable in general. That is the reason why wireless networks cannot use congestion control.

**Congestion avoidance** tries to avoid congestion completely, and not to react on it. It tries to pre-empt congestion and avoid it. We have 3 such mechanisms:

**DECbit:** Every router along a path monitors its queues usage. If the queue usage exceeds an average over some time-interval when a packet arrives, it flags the DECbit in and in the packet downstream. When the destination sees the DECbit flagged, it copies it to the ACKs sent back to the source. If more than 50% of packets in the window were flagged DECbit source divides its Congestion window by 2.

**RED:** Router that detects likely congestion drops a packet with a certain probability. This puts the TCP flow in “fast-retransmit with fast-recovery” mod and reduces the flow rate. Drop probability is calculated based on average queue length so it becomes more aggressive based on load.

**Traffic shaping** (Token Bucket Policing): It is usually performed at the ingress routers in the network. Router has predefined rate of traffic with some number of tokens. Each packet needs to have tokens to be transmitted. If tokens are not available packet is queued in the buffer. This technique aims UDP because it doesn’t understand congestion control and it can congest the network.