Networks and Distributed Systems

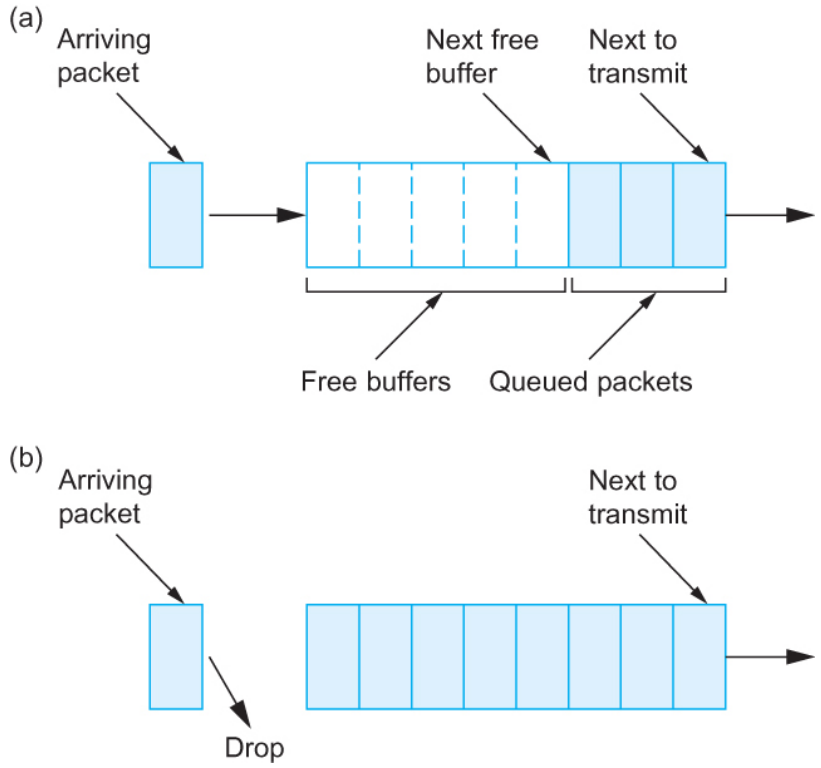
Lecture 17 – Congestion Control and Resource Allocation

Queuing Disciplines

First-IN-First-Out (FIFO)

- The first packet that arrives at a router is the first packet to be transmitted
- Given that the amount of buffer space at each router is finite
  - if a packet arrives and the queue (buffer space) is full, then the router discards that packet
- This is done without regard to which flow the packet belongs to or how important the packet is.
- This is sometimes called *tail drop, since packets that arrive at* the tail end of the FIFO are dropped

Queuing Disciplines



(a) FIFO queuing; (b) tail drop at a FIFO queue.

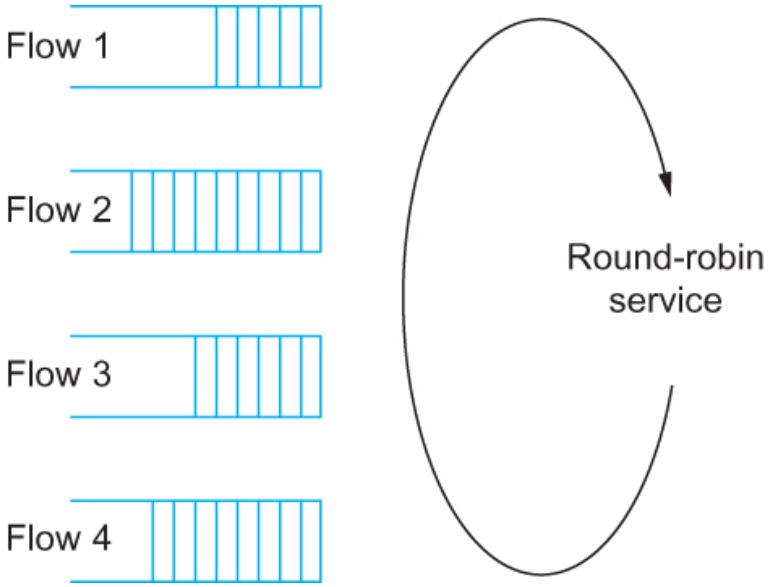
Priority Queuing

- Mark each packet with a priority
- The mark could be carried, for example, in the IP header.
- The routers :
  - Implement multiple FIFO queues: one for each priority class.
  - Transmits packets out of the highest-priority queue if that queue is nonempty before moving on to the next priority queue.
- Within each priority, packets are still managed in a FIFO manner.

Fair Queuing

- The main problem with FIFO queuing is that it does not separate packets according to the flow to which they belong.
- Fair queuing (FQ) is an algorithm that has been proposed to address this problem.
- FQ maintains a separate queue for each flow currently being handled by the router. The router then services these queues in a sort of round-robin manner.

Fair Queuing



Round-robin service of four flows at a router

Fair Queuing

- The main complication with Fair Queuing is that the packets being processed at a router are not necessarily the same length.
- To truly allocate the bandwidth of the outgoing link in a fair manner, it is necessary to take packet length into consideration.
- Example:
  - Two flows, one with 1000-byte packets and the other with 500-byte packets
  - A simple round-robin servicing from each flow's queue will give the first flow 2/3 of the link's bandwidth and the second flow only 1/3 of its bandwidth.

Fair Queuing

- What we really want is bit-by-bit round-robin
  - Transmits a bit from flow 1, then a bit from flow 2, & so on.
- Clearly, it is not feasible to interleave the bits from different packets.
- FQ mechanism therefore simulates this behavior
  - Determine when a packet would finish being transmitted if it were being sent using bit-by-bit round-robin
  - Use this finishing time to sequence the packets for transmission.

Fair Queuing

- To understand the algorithm for approximating bit-by-bit round robin, consider the behavior of a single flow

- For this flow, let
  - P_i : denote the length of packet i
  - S_i: time when the router starts to transmit packet i
  - F_i: time when router finishes transmitting packet i
  - Clearly, $F_i = S_i + P_i$

Fair Queuing

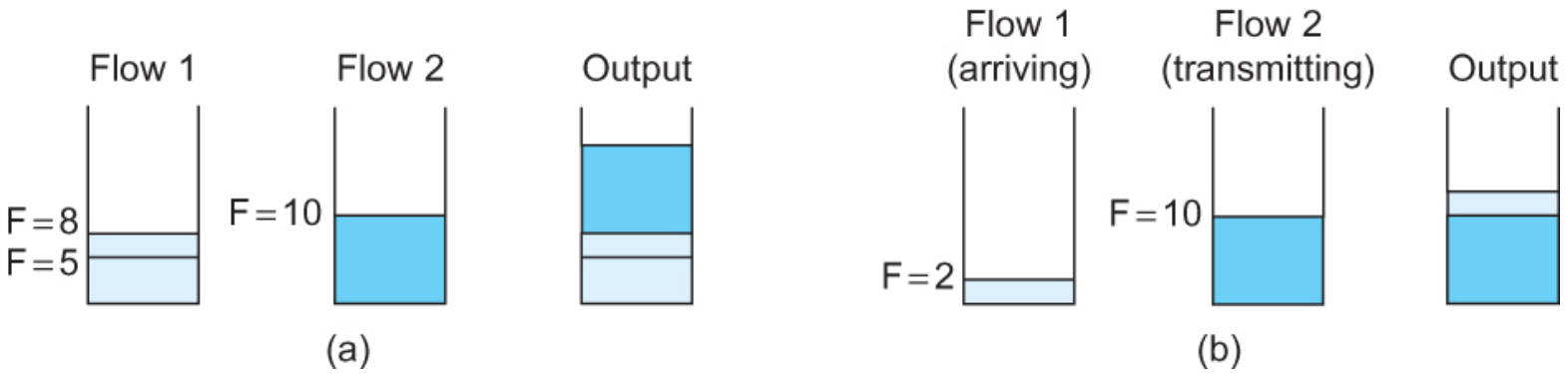
- When do we start transmitting packet *i*?
  - Depends on whether packet i arrived before or after the router finishes transmitting packet *i*-1 for the flow

- Let A_i denote the time that packet *i* arrives at the router
- Then $S_i = \max(F_{i-1}, A_i)$
- $F_i = \max(F_{i-1}, A_i) + P_i$

Fair Queuing

- Now for every flow, we calculate F_i for each packet that arrives using our formula
- We then treat all the F_i as timestamps
- Next packet to transmit is always the packet that has the lowest timestamp
  - The packet that should finish transmission before all others

Fair Queuing



Example of fair queuing in action: (a) packets with earlier finishing times are sent first; (b) sending of a packet already in progress is completed

TCP Congestion Control

- TCP congestion control was introduced into the Internet in the late 1980s by Van Jacobson,

- Immediately preceding this time, the Internet was suffering from congestion collapse—
  - hosts would send their packets into the Internet as fast as the advertised window would allow
  - congestion would occur at some router (causing packets to be dropped)
  - hosts would time out and retransmit their packets, resulting in even more congestion

TCP Congestion Control

- Each source to determine how much capacity is available in the network, so that it knows how many packets it can safely have in transit.
  - Once a given source has this many packets in transit, it uses the arrival of an ACK as a signal that one of its packets has left the network
  - It is therefore safe to insert a new packet into the network without adding to the level of congestion.
  - By using ACKs to pace the transmission of packets, TCP is said to be *self-clocking.*

Additive Increase Multiplicative Decrease

- TCP maintains a new state variable for each connection, called *CongestionWindow*
  - Used by the source to limit how much data it is allowed to have in transit at a given time.
- The congestion window is congestion control's counterpart to flow control's advertised window.
- TCP is modified such that the maximum number of bytes of unacknowledged data allowed is now the minimum of the congestion window and the advertised window

Additive Increase Multiplicative Decrease

- TCP's effective window is revised as follows:
  - MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
  - EffectiveWindow = MaxWindow − (LastByteSent − LastByteAcked).
- That is, MaxWindow replaces AdvertisedWindow in the calculation of EffectiveWindow.
- Thus, a TCP source is allowed to send no faster than the slowest component—the network or the destination host—can accommodate.

Additive Increase Multiplicative Decrease

- The problem, of course, is how TCP comes to learn an appropriate value for CongestionWindow.
- Unlike the AdvertisedWindow, which is sent by the receiving side of the connection, there is no one to send a suitable CongestionWindow to the sending side of TCP.
  - The answer is that the TCP source sets the CongestionWindow based on the level of congestion it perceives to exist in the network.
  - This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down. Taken together, the mechanism is commonly called *additive increase/multiplicative decrease (AIMD)*

Additive Increase Multiplicative Decrease

- The key question, then, is how does the source determine that the network is congested and that it should decrease the congestion window?
  - Based on the observation: packet drops.
  - TCP interprets timeouts as a sign of congestion and reduces the rate at which it is transmitting.
  - Specifically, each time a timeout occurs, the source sets CongestionWindow to half of its previous value.
  - This halving of the CongestionWindow for each timeout corresponds to the “multiplicative decrease” part of AIMD.

TCP Congestion Control

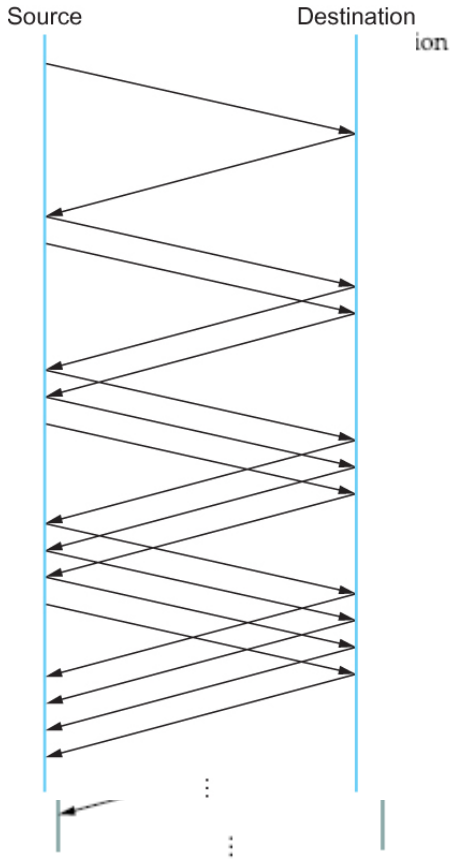
- Additive Increase Multiplicative Decrease
  - Although CongestionWindow is defined in terms of bytes, it is easiest to understand multiplicative decrease if we think in terms of whole packets.
    - For example, suppose the CongestionWindow is currently set to 16 packets. If a loss is detected, CongestionWindow is set to 8.
    - Additional losses cause CongestionWindow to be reduced to 4, then 2, and finally to 1 packet.
    - CongestionWindow is not allowed to fall below the size of a single packet, or in TCP terminology, the *maximum segment size (MSS).*

Additive Increase Multiplicative Decrease

- A congestion-control strategy that only decreases the window size is obviously too conservative.
- We also need to be able to increase the congestion window to take advantage of newly available capacity in the network.
- This is the "additive increase" part of AIMD, and it works as follows.
  - Every time the source successfully sends a CongestionWindow's worth of packets—that is, each packet sent out during the last RTT has been ACKed—it adds the equivalent of 1 packet to CongestionWindow.

TCP Congestion Control

- Additive Increase Multiplicative Decrease

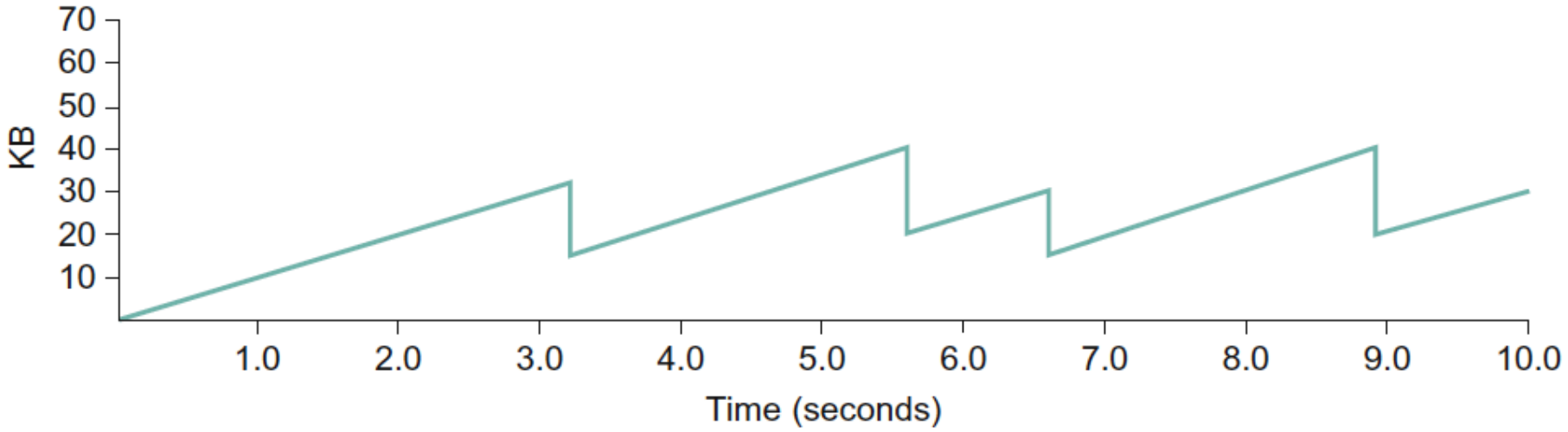


Packets in transit during additive increase, with one packet being added each RTT.

Additive Increase Multiplicative Decrease

- In practice, TCP does not wait for an entire window's worth of ACKs to add 1 packet
- Instead it increments CongestionWindow by a little for each ACK that arrives.
- Specifically, the congestion window is incremented as follows each time an ACK arrives:
  - Increment = 1/CongestionWindow
  - CongestionWindow+= Increment

Additive Increase Multiplicative Decrease



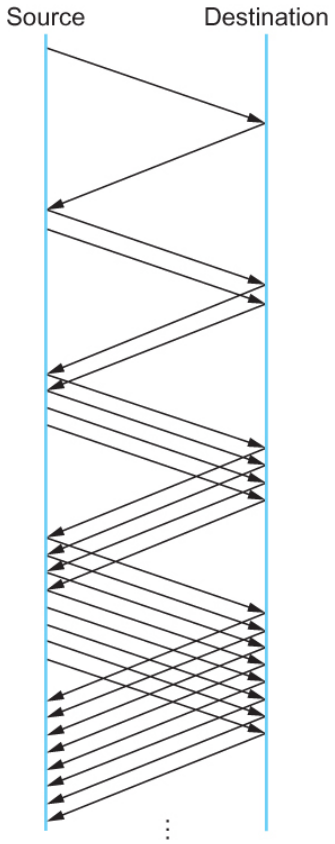
Slow Start

- The additive increase mechanism just described is the right approach to use when the source is operating close to the available capacity of the network
  - But it takes too long to ramp up a connection when it is starting from scratch.
- TCP uses a second mechanism, called *slow start,* that is used to increase the congestion window rapidly from a cold start.
- Slow start effectively increases the congestion window exponentially, rather than linearly.

Slow Start

- Specifically, the source starts out by setting CongestionWindow to one packet.
- When the ACK for this packet arrives, TCP adds 1 to CongestionWindow and then sends two packets.
- Upon receiving the corresponding two ACKs, TCP increments CongestionWindow by 2—one for each ACK—and next sends four packets.
- The end result is that TCP effectively doubles the number of packets it has in transit every RTT.

Slow Start



Packets in transit during slow start.

Slow Start

- There are actually two different situations in which slow start runs.
- The first is at the very beginning of a connection, at which time the source has no idea how many packets it is going to be able to have in transit at a given time.
  - In this situation, slow start continues to double CongestionWindow each RTT until there is a loss, at which time a timeout causes multiplicative decrease to divide CongestionWindow by 2.

Slow Start

- The second situation in which slow start is used is a bit more subtle; it occurs when the connection goes dead while waiting for a timeout to occur.
  - When a packet is lost, the source eventually reaches a point where it has sent as much data as the advertised window allows, and so it blocks while waiting for an ACK that will not arrive.
  - Eventually, a timeout happens, but by this time there are no packets in transit, meaning that the source will receive no ACKs to "clock" the transmission of new packets.
  - The source will receive a single cumulative ACK that reopens the entire advertised window, but the source then uses slow start to restart the flow of data rather than dumping a whole window's worth of data on the network all at once.

Slow Start

- Although the source is using slow start again, it now knows more information than it did at the beginning of a connection.
- Specifically, the source has a current (and useful) value of CongestionWindow; this is the value of CongestionWindow that existed prior to the last packet loss, divided by 2 as a result of the loss.
- We can think of this as the "target" congestion window.
- Slow start is used to rapidly increase the sending rate up to this value, and then additive increase is used beyond this point.

Slow Start

- We have a small bookkeeping problem to take care of
  - We want to remember the “target” congestion window resulting from multiplicative decrease as well as the “actual” congestion window being used by slow start.
- TCP introduces a temporary variable to store the target window (CongestionThreshold)
  - Set equal to the CongestionWindow value that results from multiplicative decrease.

Slow Start

- The variable CongestionWindow is then reset to one packet, and it is incremented by one packet for every ACK that is received until it reaches.

- CongestionThreshold, at which point it is incremented by one packet per RTT.

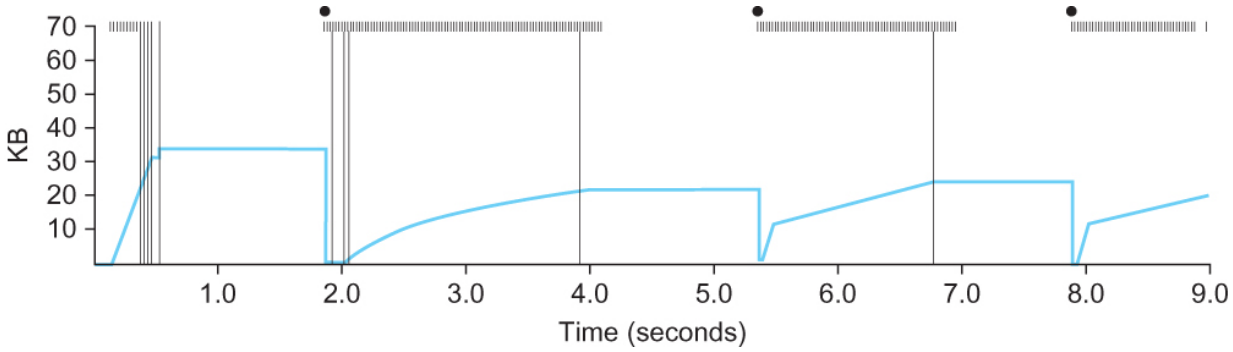
TCP Congestion Control

- Slow Start
  - In other words, TCP increases the congestion window as defined by the following code fragment:

```
{
        u_int cw = state->CongestionWindow;
        u_int incr = state->maxseg;
        if (cw > state->CongestionThreshold)
                incr = incr * incr / cw;
        state->CongestionWindow = MIN(cw + incr,
TCP_MAXWIN);
}
```

    - where state represents the state of a particular TCP connection and TCP MAXWIN defines an upper bound on how large the congestion window is allowed to grow.

Slow Start



Behavior of TCP congestion control. Colored line = value of CongestionWindow over time; solid bullets at top of graph = timeouts; hash marks at top of graph = time when each packet is transmitted; vertical bars = time when a packet that was eventually retransmitted was first transmitted.