CSc72010

TCP

Outline

TCP provides reliable, in-order, at most once delivery of packets (ie., exactly once delivery), and attempts to provide good performance, in the sense that it utilizes bandwidth in a reasonable way.

1) Exactly-once in-order delivery: sliding window
2) Performance/utilization:
    1. Flow control: Advertised window (receiver tells sender how many more bytes the receiver can accept).
    2. Congestion control: Congestion window ++ (sender keeps a maximum window size that is determined by success in delivering packets).

TCP provides an abstraction to the higher layer sending application, which can assume when it sends a byte that the byte will be received in the order that it was sent.

Flow Control Basics

Comer: Section 13.9
Peterson: Section 2.5.2 and 5.2.4
The sliding window protocol guarantees reliable, in-order delivery of frames at the data link layer. It does not provide flow control – the sender could over-run the receiver buffer if it sends too fast.

TCP sliding window differs in two respects from link layer sliding window: It operates at the level of bytes and it provides flow control by allowing the receiver to tell the sender how many more bytes it can accept with each ACK that it sends.

Review sliding window

The sender keeps track of:
1. The left side of the window, separating acknowledged bytes from those that have been sent but not acknowledged (LAR).
2. The right side of the window, defining the last byte that can be sent before receiving more acknowledgments (LAR+SWS).
3. The last byte sent (LBS/LFS).

The receiver keeps track of:
1. The left side of the window, separating bytes received and processed (passed to the higher layer) from those that have been received but not processed. (LAB/LAF-RWS).
2. The right side of the window, defining the end of the local buffer (the last acceptable byte, LAB/LAF).
3. The last byte received (LBR/LFR), which is actually the last consecutive byte

received.

The issue

Suppose the receiver has a buffer of 1000 bytes; the sender higher-layer process is passing data continually to TCP; the sender sends them and the bytes arrive in order; and the receiver acks them all. However, the higher-layer process at the receiver is busy or blocked, and doesn't read any of the bytes from the receiver buffer. Then what happens when byte 1001 is ready to be sent?

The receiver needs a way to tell the sender, “hold those bytes until I ask for more.” This is the flow control piece of sliding window.

At the link layer, we assume that the frames are processed roughly as fast as they are received (or that the buffers are adequate). In other words, the problem is solved in hardware

The TCP solution

Comer, Section 13.10
The idea is quite simple: Each time the receiver sends an ACK, it also sends the number of bytes that it's willing to accept (advertised window size). The sender should never allow more unACKed bytes to be in flight than the receiver advertised with its last ACK.

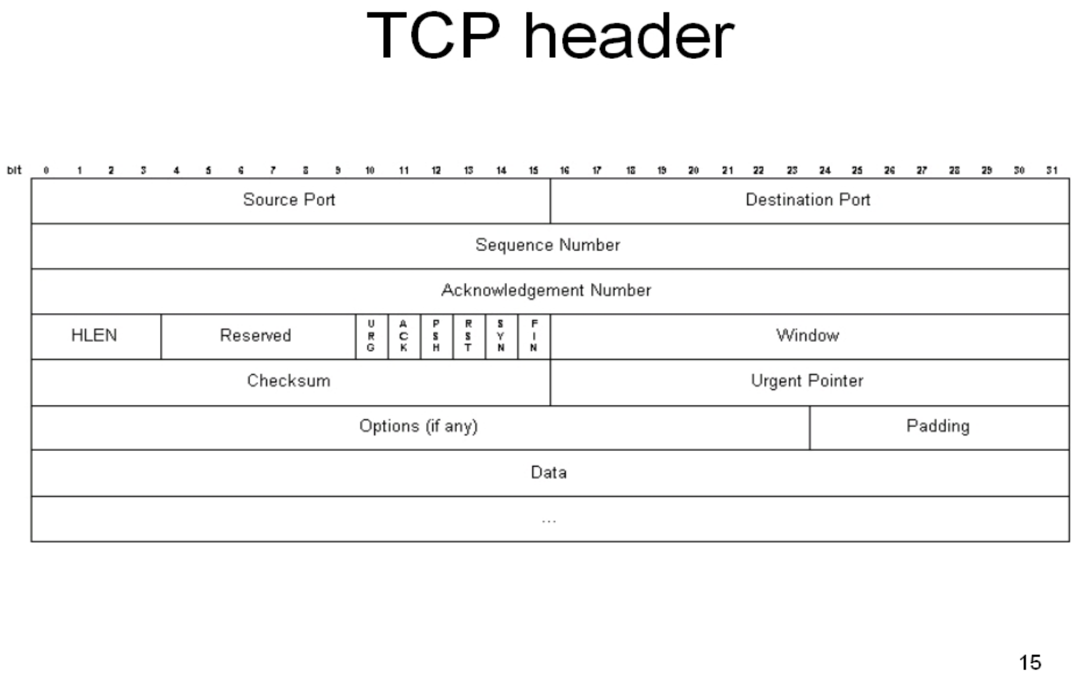
> *Example*: If the window is 1000 bytes and there are 500 bytes in flight (unACKed bytes) then 500 more bytes can be sent. When an ACK arrives, it usually reduces the number of bytes in flight (if the next expected byte is > the previous next expected byte).
>
> Note that bytes that have arrived but have not been ACKed (because there are missing bytes that should have arrived first) count as being in flight.

Successively advertised window sizes should be sensible, in the sense that they are reduced at most by the number of bytes actually sent (so that it doesn't shrink past a previously acceptable position in the byte stream – that byte might already be on the way).

> *Example*. Suppose that the receiver says it can accept 1000 bytes and the sender sends 500 bytes. It will not be acceptable for the receiver to say that <100 bytes when it ACKs the first 400 bytes (because that says that some of the outstanding bytes cannot be accepted.

Thus the advertised window is also a commitment to accept that many bytes.

Review TCP Header



The relevant fields:

Sequence Number: number assigned to first byte of data in seqment
Acknowledgment number: number of next byte expected (when this process is acting as receiver).
Window: number of unacked bytes that this process can accept as receiver.
Data: data bytes from buffer

When to send a packet

Peterson, Section 5.2.5

1. After collecting MSS (Maximum Segment Size) bytes.
2. The sending process invokes a PUSH operation, telling TCP to go ahead and send whatever is in the buffer (could just be one byte).
3. A timer fires.

In cases 2 and 3, the sender sends bytes from the buffer up to the min of the receiver's advertised window and MSS.

Silly Window Syndrome

If the receiver is processing messages more slowly than the sender is sending them, the advertised window size will decrease over time. Say the receiver is running so slowly that it only processes a byte at a time. Sooner or later the sender gets close to over-running the receiver, and let's say we see the following:

Seconds	Bytes in receive buffer	Adv window size	Send

t	999	1	
t+delay			1 byte packet
t+2*delay	999	1	
t+3*delay			1 byte packet
t+4*delay	999 ...		

> *Analogy*: Think of a TCP stream as a conveyor belt of "full" boxes going from sender to receiver and "empty" boxes coming back. MSS-sized packets are big boxes; 1 byte packets are small boxes. If a sender always uses a container as soon as it appears, then once there are single byte containers in the system they will stay forever.

If the receiver is processing more slowly than the sender, this is inevitable! (Suppose for example you have a server with a high load).

Receiver fix

The sender may need to send small containers (e.g., telnet will typically involve short commands from the client to the server and short prompts from the server to the client). However, the receiver can delay ACK's to send a larger rather than smaller one. However, this may cause the sender to re-send data, depending on his timeout.

Nagle's Algorithm

It's always ok to send large packets (available data >= MSS, window >= MSS). Don't send smaller packets while there's already data in flight; ie, if there are any outstanding ACK's, wait to hear about them. This should enlarge the window somewhat.

Impact on interactive applications like telnet: Roughly, it can send one message per RTT.

Estimating RTT: Adaptive Retransmission

We don't want to re-transmit too often – instead, we would like to retransmit only when a packet is actually lost. Roughly, if it is not lost, we should get an ACK back at around RTT after we sent the packet. However, how do we know how long the RTT is? Note that it may change as routing tables change and as network congestion increases or decreases.

Original Technique for Estimating RTT

When TCP sends a data packet, it records the time; when the ACK arrives, it finds the difference between the two times. This is a sample RTT. Then it uses a moving average formula: MeanRTT = alpha * MeanRTT + (1-alpha)*SampleRTT.
The original TCP specification recommended an alpha between .8 and .9, which weights history somewhat heavily, and computes the timeout as twice the RTT.

Flaw

If the first time a packet is sent, it is lost, and then it's sent a second time, the ACK will arrive RTT after the retransmitted packet, not RTT after the original packet. In fact, it's not possible to determine if the ACK should go with the first or the second packet. The solution is to do the computation using only packets that have not been retransmitted (ie, ignore the re-transmitted packets) – this is the Karn/Partridge algorithm.

Jacobson/Karels

Timeout = mu*MeanRTT + phi*Deviation

Must estimate devation as well as mean:

Difference = Sample – MeanRTT
MeanRTT = MeanRTT + delta*Difference
Deviation = Deviation + delta*(|Difference| - Deviation)

Congestion Control in TCP

Peterson, Section 6.3
Assume FIFO or Fair Queuing in network:
FIFO: first-in, first-out
FQ: round-robin service for each connection (through a router)

The Basics: AIMD

Add a congestion window. This gives the maximum remaining unACKed bytes that can be in flight. The actual bytes allowed to be in flight is the max of the advertised window and the congestion window.

```
MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
EffectiveWindow = MaxWindow – (LastByteSent – LastByteAcked)
```

The idea: decrease the congestion window size when a packet is dropped, increase it when a packet is acked. By how much?

The Algorithm: Additive Increase, Multiplicative Decrease

TCP interprets timeouts as a sign of congestion, and halves the window size when a timeout occurs (signaling a lost packet). Each time a new packet arrives, as signaled by an ACK, add the equivalent of a packet to the congestion window.

```
        Increment = MSS * (MSS/CongestionWindow)
        CongestionWindow += Increment
```

The minimum congestion window is MSS; effectively, if the Congestion Window allows 1 MSS (min size), we add 1 MSS to the window; when it allows 2 MSS, it takes two packets to add 1 MSS to the window; and so on. In other words, each time the source successively sends a CongestionWindow worth of packets, it adds the equivalent of 1 packet to the congestion window.

This gives a sawtooth pattern in response to packet losses.

Why is this reasonable? Because packets delayed by congestion get re-sent, increasing the congestion, ie, we have a positive feedback loop – too much traffic induces more traffic. So the sender needs to be aggressive about reducing traffic and cautious about increasing it.

Improving TCP

Slow Start

AIMD is too conservative when a session starts up – there may be plenty of bandwidth available. So, the idea is to start the congestion window at 1 packet and double it each time a new ACK arrives, until a packet times out. Then it switches over to AIMD.

This is called "slow" because it's the alternative to using the full advertised window immediately – this is slower than that.

Slow start runs at the very beginning of a connection. It also runs when a packet has been lost (rather than delayed), and the sender blocks waiting for an ACK that never arrives. When the timeout occurs, the sender uses slow start.

In the slow start used a the beginning of a session, the sender has no information about available bandwidth and increases until the first packet is lost. But after a lost packet, the sender has the Congestion Window that had been computed – this becomes the target congestion window. Instead of looking for a packet to be lost, slow start increases to the size of the Congestion Window right after the lost packet and then switches over to AIMD.

Fast Retransmit and Fast Recovery

There's a relatively long fallow period following a lost packet, while the sender waits for a time-out. We can tweak the receiver to improve this: The basic sliding window algorithm for TCP sends an ACK when packet arrives that increases the number of contiguous (in-order) bytes received.

If instead, the receiver sends an ACK each time a packet arrives at the receiver, but t*he ACK still contains only the next expected byte,* then the sender knows that a packet has gotten through but that a previous packet was either lost or delayed. Then the sender can re-send the next expected packet without waiting for the timeout.

However, there are still congestion issues, so we want to be sure that the packet has actually been lost, not merely delayed. This is heuristic, but at present a TCP sender waits until 3 duplicate acks have been received before re-sending the lost-or-delayed packet.

This greatly reduces the period of time between loss of a packet and its re-transmission.

Modeling TCP

The model will have a bad channel in it, which loses, duplicates, and re-orders messages. It will also have a sender and a receiver. These could be decomposed into various components – e.g., sliding window, RTT computation, flow control, etc – or not. Probably (considering the properties we want to prove) we want the TCP ioa to have actions corresponding to getting data from an application on the sender side and giving data to the partner application on the receiver side.

Important TCP Properties

Basic properties correspond to implementing a reliable FIFO channel. These could be proved by defining a simulation relation between the above composition of machines and the reliable FIFO.

Liveness

Basic: Every message that is sent is eventually received in a fair trace.

Safety

Basic: No messages are received out of order.
Basic: No messages are received twice.
Flow control: There are never more bytes in flight than the advertised window allows.
Flow control (preliminary): The receiver never rolls back the advertised window.
Congestion control: There are never more bytes in flight than the congestion window allows.