RPC Failure Models

- How many times is an RPC done?
  - Exactly once?
    - Server crashes before request arrives
    - Server crashes after ack, but before reply
    - Server crashes after reply, but reply dropped
  - At most once?
    - If server crashes, can’t know if request was done
  - At least once?
    - Keep retrying across crashes, but may be done multiple times
    - Example: NFS idempotent ops (ex: read/write file block)
Exactly Once RPC

- Example: buy something over Ebay, Amazon
  - want exactly one widget, book, 100 shares of kozmo
- Want RPC to be
  - done exactly once
  - done completely or not at all
  - done atomically with respect to other requests
  - once done, stays done (independent of later crashes)
- Analogous to distributed database transactions

Exactly Once RPC

- Can implement using disk on both ends
  - client writes “about to make request” to disk
    - keep retrying until there is a reply (done/abort)
  - client sends request
  - server gets request; computes result
  - server writes “about to reply” to disk
    - along with contents of reply message
  - server sends reply
  - client writes “got response” to disk
    - to remove request; if crash, don’t want to retry
General’s Paradox

Can we use messages and retries to synchronize two machines so they are guaranteed to do some operation at the same time?

No.

General’s Paradox Illustrated

A
3:30 ok for launch?

B

ok, 3:30 is good for me

so, its 3:30?

yeah, but what if you don’t get this ack?
Bandwidth Allocation

- How do we efficiently share network resources among billions of hosts?
  - Congestion control
    - Sending too fast causes packet loss inside network -> retransmissions -> more load -> more packet losses -> …
    - Don’t send faster than network can accept
  - Fairness
    - How do we allocate bandwidth among different users?
    - Each user should get fair share of bandwidth

Congestion

- Buffer absorbs bursts when input rate > output
- If sending rate is persistently > drain rate, queue builds
- Dropped packets represent wasted work

Chapter 6, Figure 1
Fairness

Each flow from a source to a destination should get an equal share of the bottleneck link ... depends on paths and other traffic

Chapter 6, Figure 2

The Problem

- Original TCP sent full window of data
- When links become loaded, queues fill up, and this can lead to:
  - Congestion collapse: when round-trip time exceeds retransmit interval -- every packet is retransmitted many times
  - Synchronized behavior: network oscillates between loaded and unloaded
Jacobson Solution

- Modify retransmission timer to adapt to variations in queuing delay
  - Timeout based on measured RTT and variance
- Infer network bandwidth from packet loss
  - drops => congestion => reduce rate
    - drops also caused by link noise!
  - no drops => no congestion => increase rate
- Limit send rate based on network bandwidth in addition to receiver buffer space
  - minimum of what network and receiver can accept

TCP Congestion Control

- Adjust rate to match network bandwidth
  - Additive increase/multiplicative decrease
    - oscillate around bottleneck capacity
  - Slow start
    - quickly identify bottleneck capacity
  - Fast retransmit
  - Fast recovery
Tracking the Bottleneck Bandwidth

- Sending rate = window size/RTT
- Multiplicative decrease
  - Timeout => dropped packet => cut window size in half
    - and therefore cut sending rate in half
- Additive increase
  - Ack arrives => no drop => increase window size by one packet/window
    - and therefore increase sending rate a little

TCP “Sawtooth”

- Oscillates around bottleneck bandwidth
  - adjusts to changes in competing traffic

![Graph showing Additive Increase/Multiplicative Decrease over round-trip times and window sizes.](image-url)
How do we find bottleneck bandwidth?

- Start by sending a single packet
  - start slow to avoid overwhelming network
- Multiplicative increase until get packet loss
  - quickly find bottleneck
- Remember previous max window size
  - shift into linear increase/multiplicative decrease when get close to previous max ~ bottleneck rate
  - called “congestion avoidance”

**Slow Start**

- Quickly find the bottleneck bandwidth
Slow Start Problems

- Bursty traffic source
  - will fill up router queues, causing losses for other flows
  - solution: ack pacing
- Slow start usually overshoots bottleneck
  - will lose many packets in window
  - solution: remember previous threshold
- Short flows
  - Can spend entire time in slow start!
  - solution: persistent connections?

Avoiding burstiness: ack pacing

Window size = round trip delay * bit rate
Ack Pacing After Timeout

- Packet loss causes timeout, disrupts ack pacing
  - slow start/additive increase are designed to cause packet loss
- After loss, use slow start to regain ack pacing
  - switch to linear increase at last successful rate
  - “congestion avoidance”

Putting It All Together

- Timeouts dominate performance!
Fast Retransmit

- Can we detect packet loss without a timeout?
  - Receiver will reply to each packet with an ack for last byte received in order
- Duplicate acks imply either
  - packet reordering (route change)
  - packet loss
- TCP Tahoe
  - resend if sender gets three duplicate acks, without waiting for timeout

Fast Retransmit Caveats

- Assumes in order packet delivery
  - Recent proposal: measure rate of out of order delivery; dynamically adjust number of dup acks needed for retransmit
- Doesn’t work with small windows (e.g. modems)
  - what if window size <= 3
- Doesn’t work if many packets are lost
  - example: at peak of slow start, might lose many packets
Fast Retransmit

- Regaining ack pacing limits performance

Fast Recovery

- Use duplicate acks to maintain ack pacing
  - duplicate ack => packet left network
  - after loss, send packet after every other acknowledgement
- Doesn’t work if lose many packets in a row
  - fall back on timeout and slow start to reestablish ack pacing
**Fast Recovery**

- Slow Start + Congestion Avoidance
- Fast Retransmit + Fast Recovery

- Round-trip times
- Window (in seqs)

**Delayed ACKS**

- **Problem:**
  - In request/response programs, server will send separate ACK and response packets
    - computing the response can take time

- **TCP solution:**
  - Don’t ACK data immediately
  - Wait 200ms (must be less than 500ms)
  - Must ACK every other packet
  - Must not delay duplicate ACKs
Delayed Ack Impact

- TCP congestion control triggered by acks
  - if receive half as many acks => window grows half as fast
- Slow start with window = 1
  - ack will be delayed, even though sender is waiting for ack to expand window