The Google File System

Outline

- **Review: Networking**
  - IP
  - TCP
  - UDP
  - DNS
- Google File System
  - Rationale
  - Reading
  - Consistency and definition
  - Write forwarding
  - Fault tolerance
  - Strengths/Weaknesses

Review: Internet Protocol (IP)

- IP address is just a number to reach your computer
  - **Routers** around the world forward incoming packets to get to the destination
  - “Ah, this packet is destined for 141.242*, let me send it down this next path”
  - You can **traceroute** to find all the **hops** required to get from one place to another

IP Traceroute Tool

Traceroute checks the route packets take to the specified host from the UltraTools server.

Enter a host name or IP address, and Maximum Hops:

google.com | 64 | Go »

Related Tools: Looking Glass Ping Ping-IPv6 Traceroute-IPv6 DNS Traversal

```
      Hop number: 1
   Connected to: assc-ultrafw-vlan2598.dc10.neustar.com ( 10.176.98.1 )
Roundtrip times: 1.487 ms
                 1.655 ms
                 1.646 ms

      Hop number: 2
   Connected to: ashlfnsw02-vlan102.dc10.neustar.com ( 10.176.2.3 )
Roundtrip times: 1.457 ms
                 1.453 ms
                 1.679 ms

      Hop number: 3
Roundtrip times: Timed out.

      Hop number: 4
   Connected to: et-0-0-43-3.cr2-was1.ip4.gtt.net ( 173.205.39.229 )
Roundtrip times: 5.129 ms
                 3.848 ms
                 3.114 ms
         Country: germany

      Hop number: 5
   Connected to: ae13.cr1-was1.ip4.gtt.net ( 213.200.115.178 )
Roundtrip times: 3.629 ms
                 6.307 ms
                 6.324 ms
         Country: united states

      Hop number: 6
   Connected to: as15169.cr3-was1.ip4.gtt.net ( 69.174.23.134 )
Roundtrip times: 4.85 ms
                 1.537 ms
                 2.816 ms
         Country: united states

      Hop number: 7
   Connected to: 108.170.246.67 ( 108.170.246.67 )
Roundtrip times: 3.318 ms
                 3.813 ms
                 2.735 ms
         Country: united states

      Hop number: 8
Roundtrip times: Timed out.

      Hop number: 9
   Connected to: 142.250.57.139 ( 142.250.57.139 )
Roundtrip times: 9.862 ms
                 9.642 ms
                 9.562 ms
         Country: united states
```

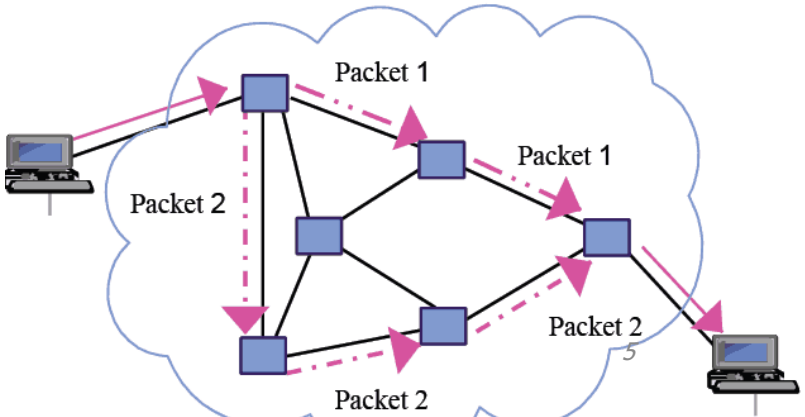
Review: Internet Protocol (IP)

- IP addresses are basically “agreed upon”
  - https://en.wikipedia.org/wiki/List_of_assigned_/8_IPv4_address_blocks
  - Organizations that run **internet backbones** play nice with this arrangement
    - If you try to assign your computer an IP address of someone else, the existing routers on the internet would not route traffic to you
- Every networked device has a **routing table** that tells it which interface to use for incoming requests
  - Routing often based on **hops** – how many nodes must the packet travel to reach its destination?

Inside a **switch** or **router**, the software looks at the IP address in the header, and decides which **interface** it should use to communicate that packet.

Review: Packet Switching and Hops

- Communicating one packet from one device to the next in a route is said to take one **hop**
  - Network routing can be characterized by the number of **hops** it takes to send a packet from one place to another
    - Each time a router sees a packet, it will increase that packet's hop count by 1
- Devices may set a max hopcount for each packet
  - "If the packet hasn't reached its destination after 64 hops, I'll just drop it"
  - Often to prevent cyclical routing in misconfigured networks



Review: TCP

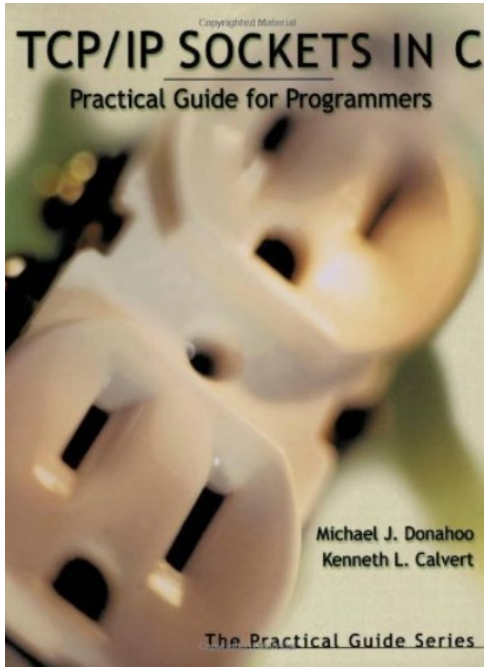
- TCP makes networking ordered and reliable
  - IP packet-switched network makes each packet *possibly* take a different route
    - Depends on routing hardware (“maybe we’ll send this packet down route A instead of B this time”)

- Idea: Establish a **connection** to a remote server
  - Client opens a random port for return communication
  - Server listens for incoming SYN packets for new connections
  - **Three-way handshake:** SYN, SYN-ACK, ACK
  - Use **sequence numbers** to track how many packets were received successfully

Aside: Sockets

- A **socket** is a data structure that networked applications maintain
  - Destination IP, Port
  - Source IP, Port (why?)
  - Other flags (TCP, UDP, blah blah)

- A socket is treated like a file
  - You read and write to it (or recv and send)
  - The OS provides this abstraction for you

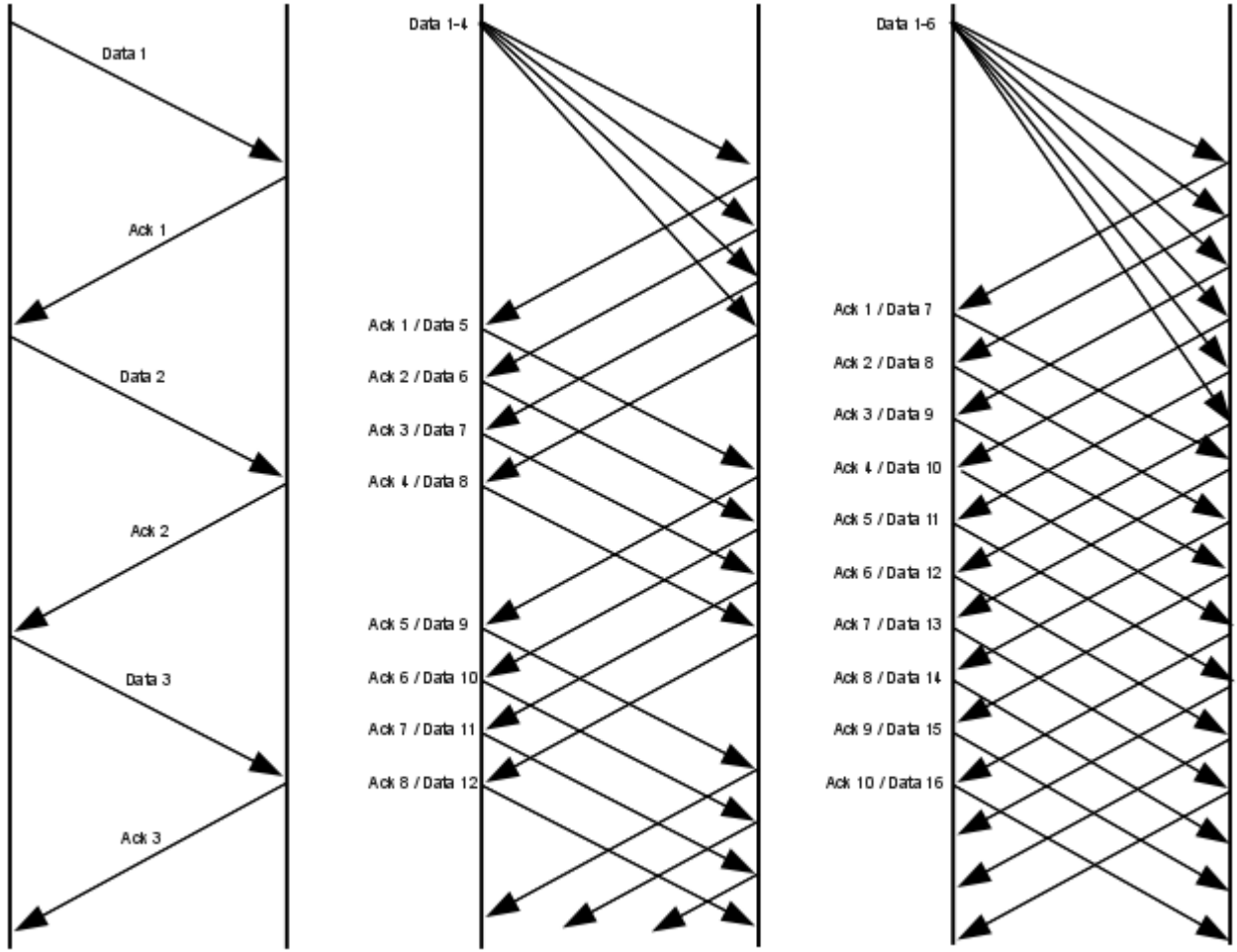


Review: Flow Control and Congestion Control

- TCP requires ACK's from the recipient before sender sends more
- We use **flow control** and **congestion control** to balance throughput and network utilization
  - Flow control: Receiver sets a maximum window size
    - Based on how fast the device and receiver application can process packets
  - Congestion control: Sender responds to network conditions
    - We use Additive Increase Multiplicative Decrease (AIMD) to *always try increasing* the number of in-flight packets *until* we observe dropped packets
    - Idea: send as much as possible, but if the network is too congested, slow down to not drop more
  - We implement flow and congestion control using **sliding windows**
    - You have a window of size X packets in a buffer. Send out X packets but wait for ACKs before proceeding

Sliding Windows

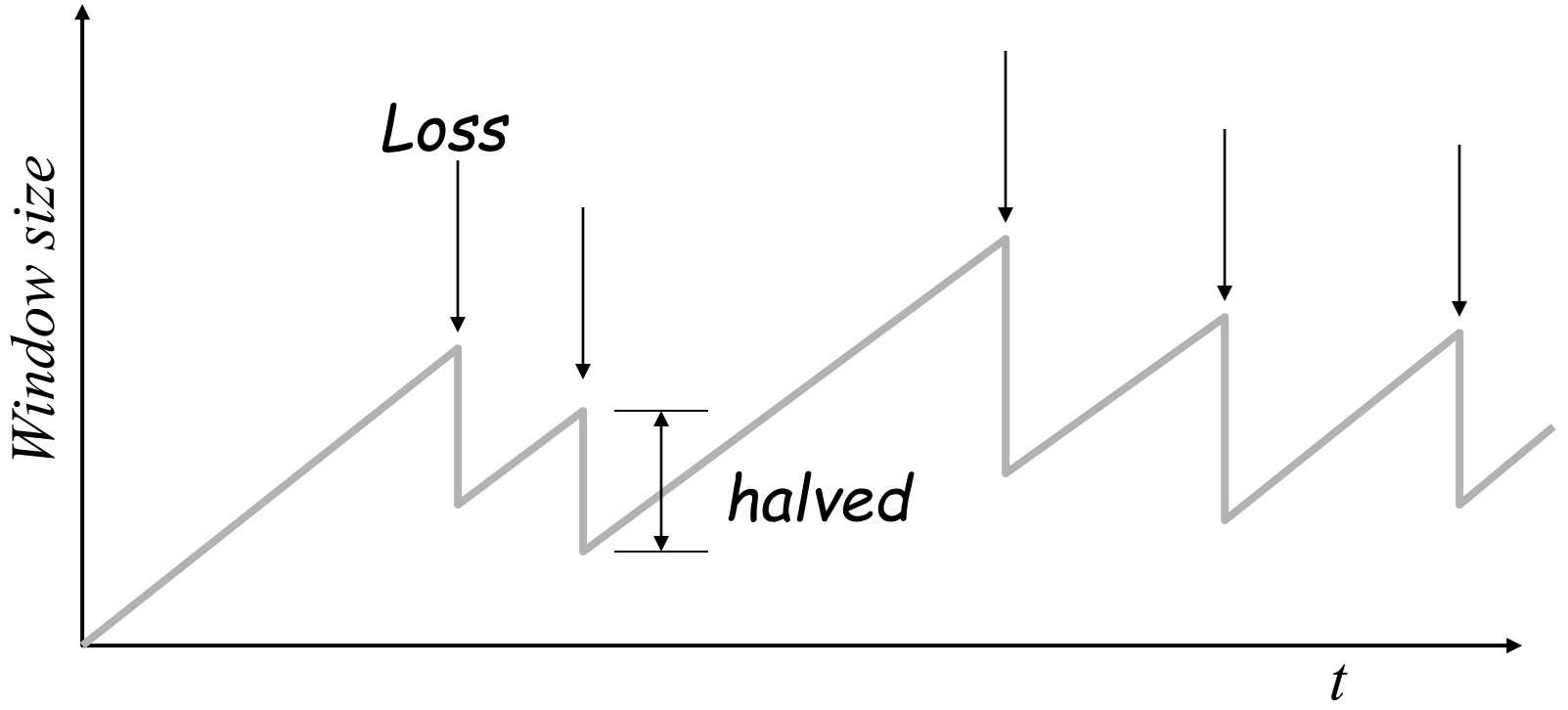
- We move the window forward only when we get ACKs back
  - If ACKs out of order, either:
    - Retransmit earlier un-ACK'd packets
    - Wait for ACKs on earlier packets
  - You wait until you have a contiguous block of ACK'd packets before moving the window forward



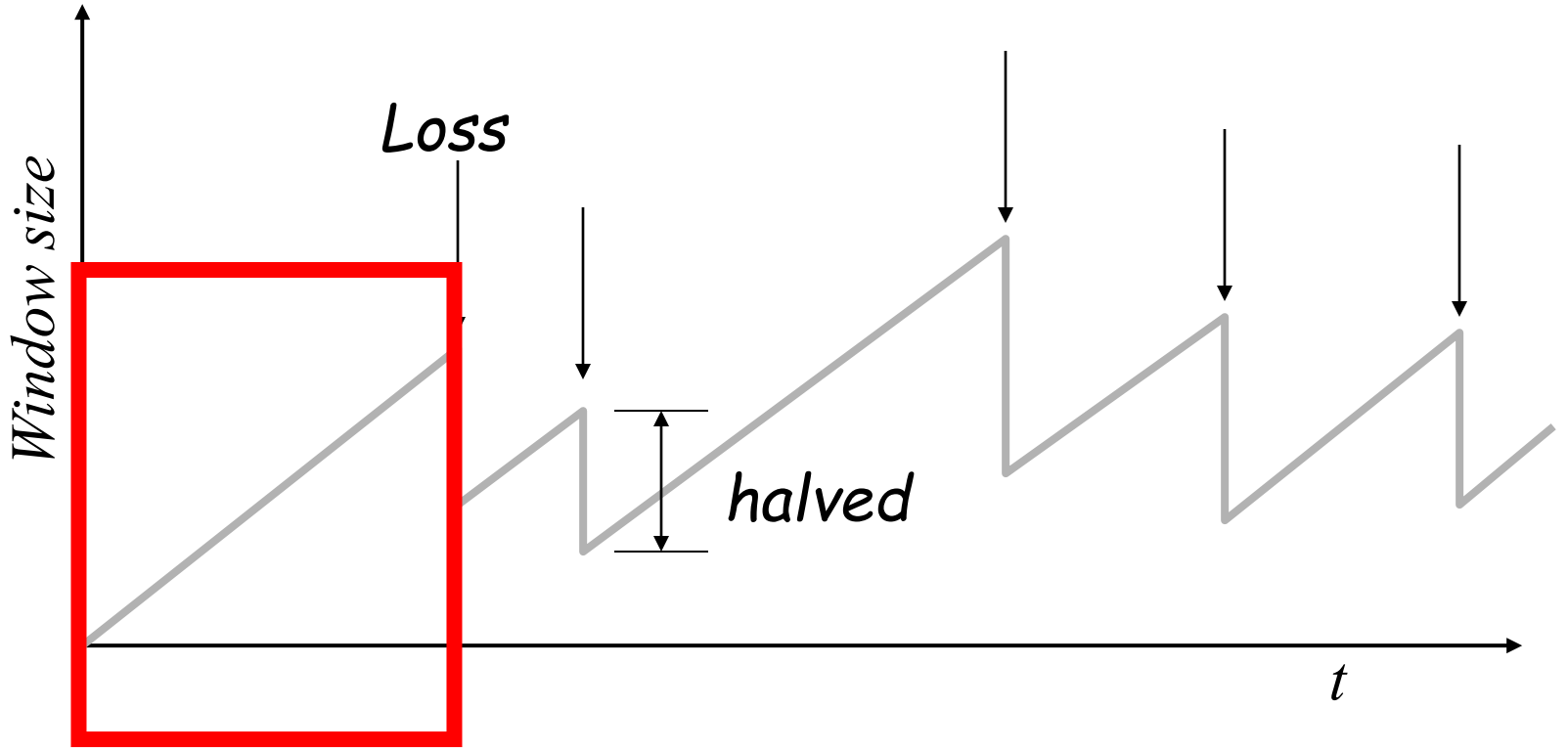
Window size: 5

Packet 1	Packet 2	Packet 3	Packet 4	Packet 5	Packet 6	Packet 7	Packet 8
SENT/ACK	SENT/ACK	SENT	SENT	SENT/ACK	SENT	SENT	---

AIMD sawtooth

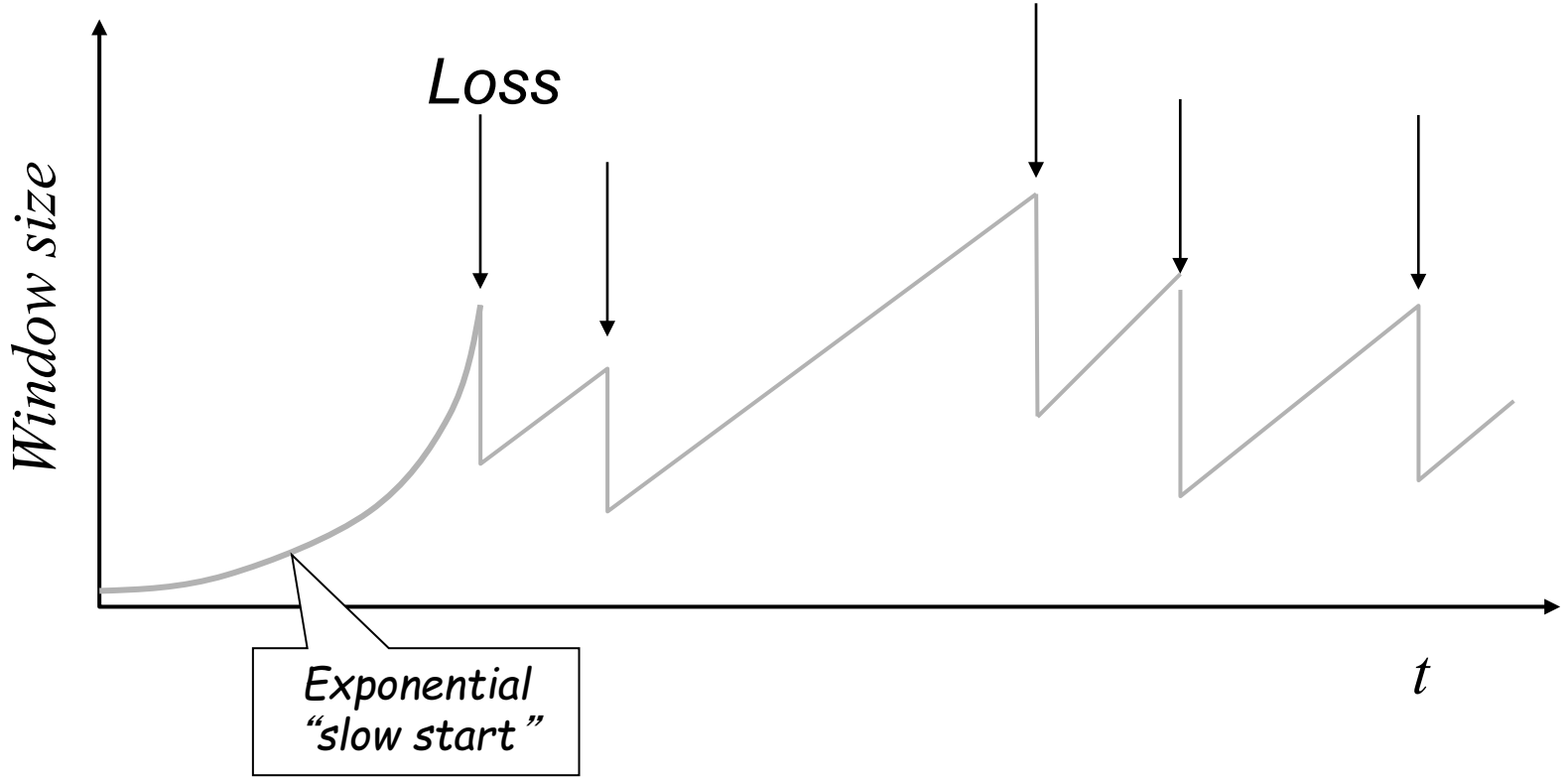


AIMD sawtooth



Problem: Starting up is *slow*

Slow start and TCP sawtooth

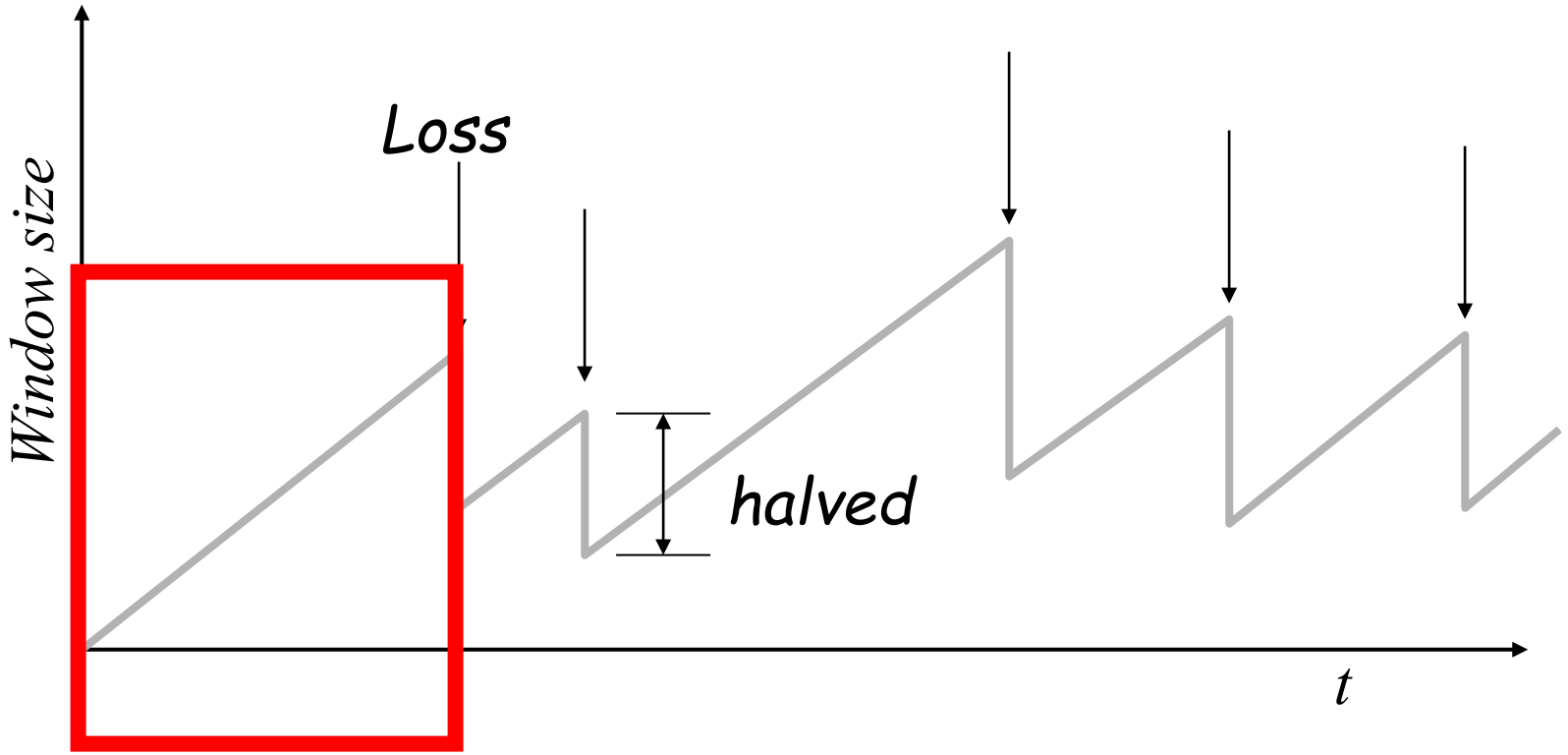


Speed up start using exponential, not linear increase

Pro-tip: "slow start" is a misnomer, it's actually quite fast

Thought Question

- What are the implications of congestion control on HTTP1.0 vs. HTTP 1.1?



Thought Question

- What will happen in a video chat app that uses TCP
  - if packets 0 and 2-100 have been received but packet 1 has not?

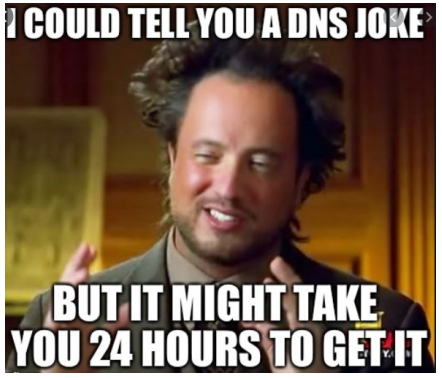
Reassembly Queue (Receiver)

Aside: UDP, the User Datagram Protocol

- The **user datagram protocol** is a simple way to exchange packets between applications
  - Like TCP, you specify **ports**
  - But UDP does **not** provide any reliability
    - No flow control, no congestion control, no sequencing, no retransmits, etc.
  - It's up to the "user" (the application) to know how to manage its own traffic
- UDP common where resending, reliability not necessary
  - Streaming video (if you drop a few frames, no big deal)
- No notion of "connection"
  - You just send a packet (a datagram) to a server… you don't know if it received it

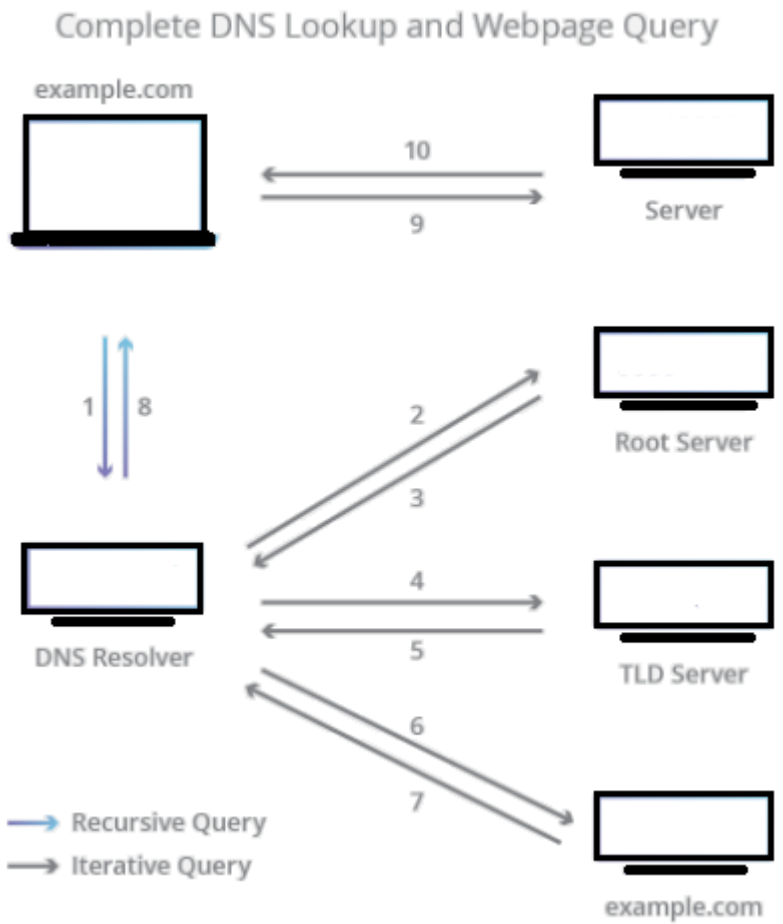
DNS, the Domain Name Service

- When you want a website, you pay for it to be listed in public DNS servers
  - Godaddy.com, other registrars will let you pay for DNS listings for a website
- Alternatively, you can get *subdomains* publicly listed
  - e.g., EECS DCO registered my host: kjleach.eecs.umich.edu
  - When someone requests that host:
    - DNS lookup for .edu, then for .umich.edu, then for .eecs.umich.edu
    - Eecs.umich.edu is a DNS server that DCO maintains, they have a listing for kjleach.eecs.umich.edu
      - Eecs.umich.edu is the *authority* for that domain
- Thought question: can you always trust what a DNS server tells you?



DNS Resolution

- You type “https://example.com/stuff/”
- Browser resolves example.com to an IP
  - Query DNS server for .com
    - “Where can I find all the **top-level domains?**”
    - Response: here’s a list of DNS servers to handle TLDs like: .com, .edu, .org, .net
  - Query TLD DNS server for example.com
    - “Where can I find example.com?”
    - Response: here’s a list of DNS servers for example.com
  - Query example.com DNS server for webserver
    - “Where is the example.com site hosted?”
    - Response: here’s the IP address of the server, thx



One Slide Summary: Google File System

- For large scale applications, we use **distributed filesystems** to spread the load of storing files across multiple physical computers
- Distributed filesystems seek to provide
  - **Large storage:** perception of large amounts of space (how much storage does each gmail user get?)
  - **Scalable access:** high throughput access that scales with the number of computing resources available to the FS (how many users access gmail at a time?)
  - **Location and fault transparency:** It doesn't matter where you access from or if a small number of nodes fail
- **Google Filesystem** is one example implementation of a distributed filesystem

Google File System (GFS)

- Goal: store more data than fits on one computer

- e.g. for building a search engine
  - ~100 billion pages
  - ~100 KB per page, on average
  - That's 10 PB!
  - Or 3,333 standard disk drives (3TB)!

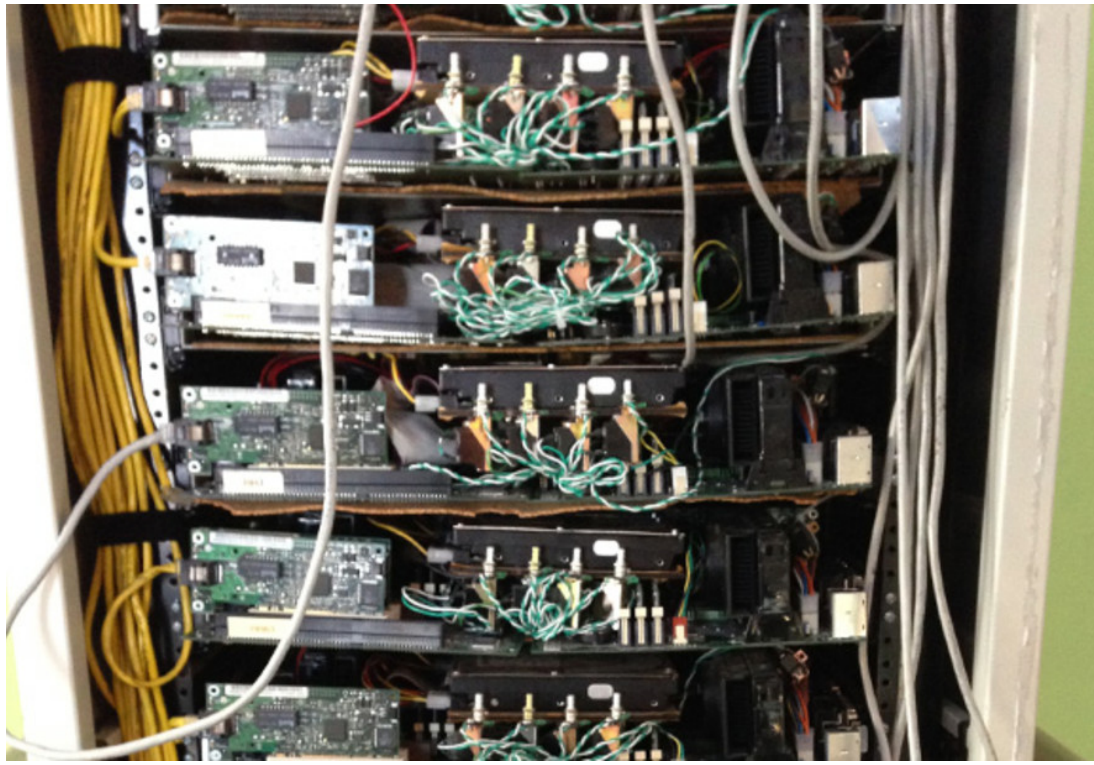
Inside a datacenter

- What it's like
  - Very loud fan noise
  - Hot in some spots and really cold in others
  - Unique smell
  - Usually some sort of water container collecting A/C condensation
- Lots of computers means lots of failures
- Google File System: 2003

Datacenter failures

- Old strategy: buy a small number of expensive servers
- New strategy: buy lots of unreliable cheap servers
  - Use software to make them look reliable and fast

- RAID: Redundant Array of Inexpensive Disks
  - Fine for local storage on one device
  - What if we want giant amounts of storage?



Why GFS?

- Store enormous amount of data

- In a way that data is not lost when some servers crash or stop working
  - **Failure** transparency and **location** transparency

- Solution:
  - store multiple copies of everything
  - keep track of where those copies are

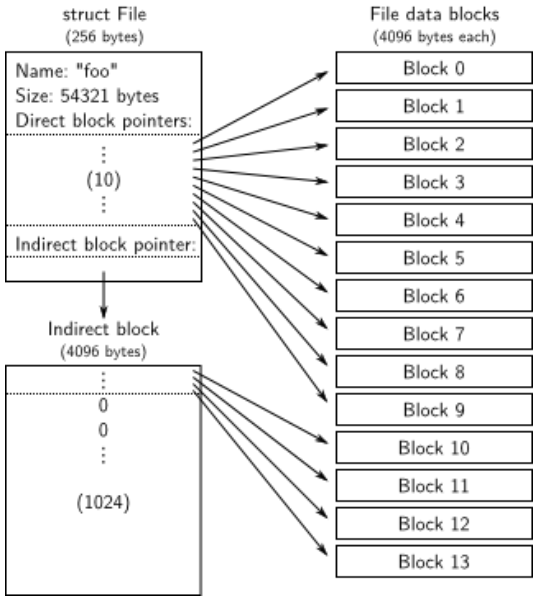
Why learn about this?

- GFS is a **distributed system**
  - No one computer is in charge of everything
- In P4, you will implement a distributed system
- Goal: see what the challenges are and what solutions look like

Traditional vs. Google File System

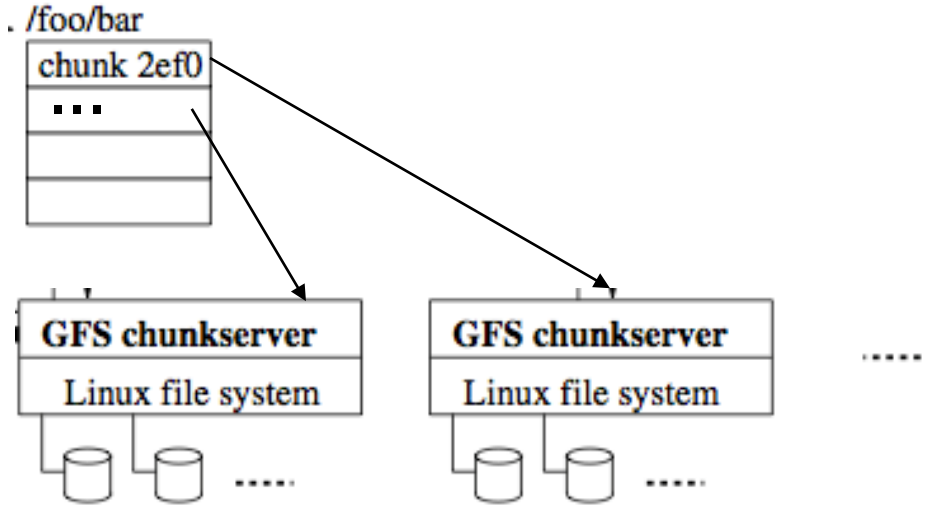
Traditional file system

- Divide file into ~kB *blocks*
- Place blocks on one HDD in one computer



Google File System

- Divide file into ~MB *chunks*
- Place chunks on different HDD in different computers



GFS Terminology

- One **master server** that *only* stores information about *where to find file data*
  - Master server basically maps filenames to lists of **chunkservers**

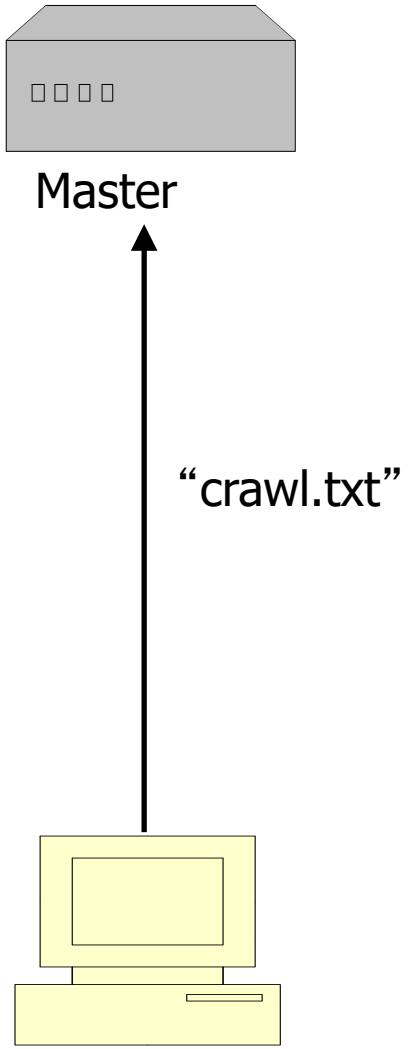
- Multiple **chunkservers** store the actual file data
  - Multiple chunkservers could be replicas
    - “file1.txt” could exists in the same state on both cserver1 and cserver2
  - One file can be split across multiple chunkservers
    - “file2.txt” could be split across cserver1, cserver2, and cserver3
  - Chunks of files can be in unrelated locations
    - “file3.txt” could have
chunk 1 on cserver1, cserver2
chunk 2 on cserver3
chunk 3 on cserver4, cserver5

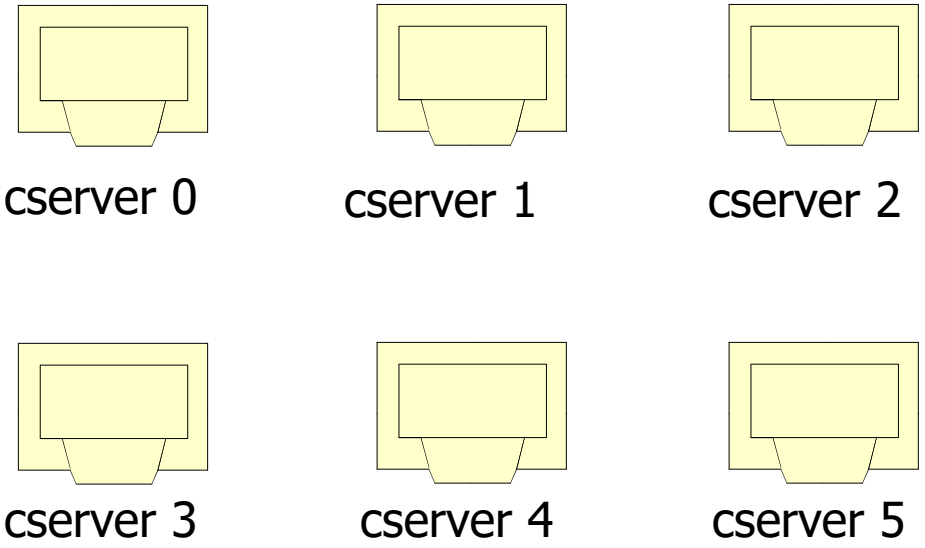
Outline

- TCP and UDP

- **Google File System**
  - Rationale
  - **Reading**
  - Consistency and definition
  - Write forwarding
  - Fault tolerance
  - Strengths/Weaknesses

GFS file read

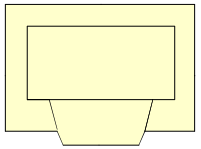




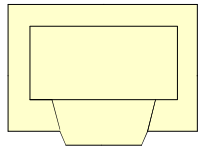
1. Client asks master for filename info

GFS file read

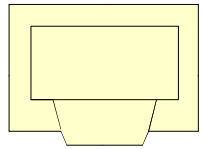
Master



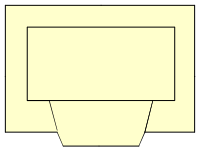
cserver 0



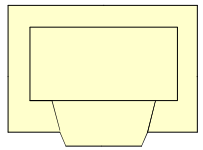
cserver 1



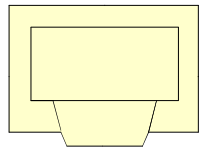
cserver 2



cserver 3

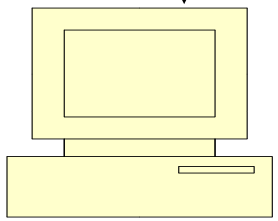


cserver 4



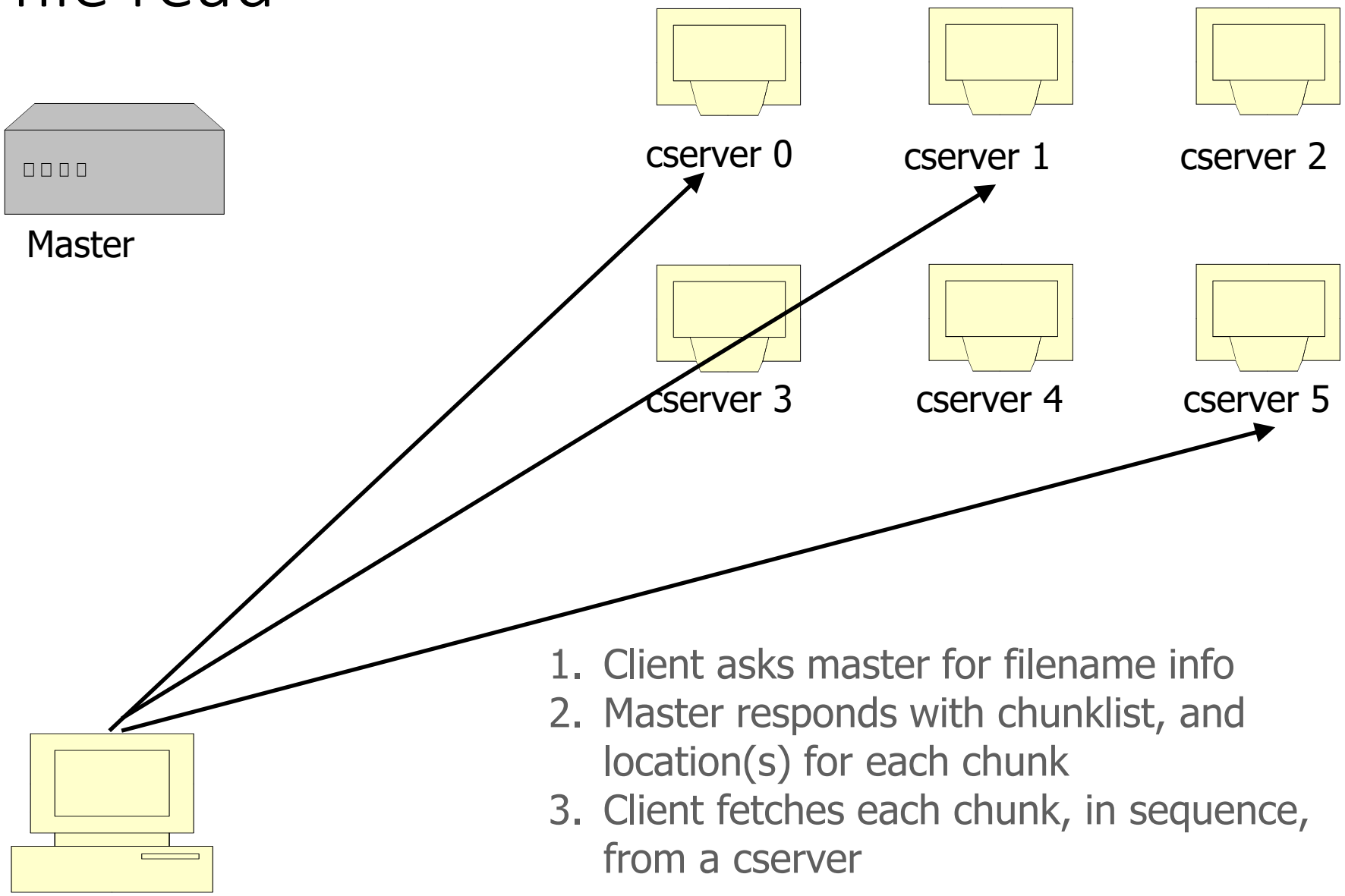
cserver 5

(chunk-33 / cserver 1, 4)
(chunk-95 / cserver 0, 2)
(chunk-65 / cserver 1, 4, 5)



1. Client asks master for filename info
2. Master responds with chunklist, and location(s) for each chunk

GFS file read



1. Client asks master for filename info
2. Master responds with chunklist, and location(s) for each chunk
3. Client fetches each chunk, in sequence, from a cserver

Outline

- TCP and UDP

- **Google File System**
  - Rationale
  - Reading
  - **Consistency and definition (aka how do we do writes?)**
  - Write forwarding
  - Fault tolerance
  - Strengths/Weaknesses

GFS file write (*mutation*)

- **Client** asks **master** for filename info
- **Master** responds with chunklist, and *location*(s) for each chunk
- **Client** sends data to *location*(s)
- Chunk **servers** apply changes *in the same order*

- What could go wrong?

Consistency model

- What if **multiple clients** *simultaneously* write the **same** file **region**?
  - A **region** is the collection of chunkservers that contain a file

- A file region is ***consistent*** if **all clients** will **always** see the same data, regardless of which replicas they read from.
  - **Inconsistent** would mean different replicas show different data

- A region is ***defined*** if, after a file data mutation, it is *consistent* and clients will see what the mutation writes in its entirety
  - **Undefined** means that the file does not reflect any one writer's mutation
    - e.g., writes got mixed and matched

Consistency model

- Why could different clients see different data?
  - Because there are copies
  - Because messages and data take time to traverse the network
  - Different messages might take different amounts of time

Consistency model

- What if multiple writes happen to the same region?
- GFS admits three outcomes:

1. Serial successful writes
   - One at a time, no errors
2. Concurrent successful writes
   - Many at a time, no errors, but undefined
3. Failed writes

1. Serial successful writes

- Serial successful writes
- All chunks see the same order of write operations

- **Consistent and defined**

2. Concurrent successful writes

- Concurrent successful writes
- All chunks see the same order of write operations

- **Consistent**, but ***undefined***

- All clients see the same data, but it may not reflect what any one mutation has written.
  - Typically, it consists of mingled fragments from multiple mutations.
- What if one server hasn't yet finished writing?

3. Failed writes

- Region is **inconsistent** (hence also **undefined**)

- Different clients may see different data at different times
- This is a huge problem in traditional relational database management systems (RDBMS)
  - In GFS, it's the client application's problem!

GFS guarantees (or lack thereof lol)

- GFS is designed for **consistency**

- Applications need to detect and handle undefined parts of files
  - At Google: did this by appending, not writing

- Why not guarantee more?
  - In a distributed system, this is really, really hard

Append

- GFS is optimized for file **append**

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

- A *record append* causes data (the "record") to be appended atomically at least once

Atomic operations

- Some operations really need to be consistent
- Example: file creation
- An *atomic* operation appears to the rest of the system as if it happened instantaneously

- For 482 friends: remember locks and synchronization?

- Basically, a sequence of operations is ***atomic*** if, *after* you start the first operation, you *finish* execution *all* of them before moving on to something else

Atomic operations

- How does GFS make an operation atomic?
- The **master node** *locks* a filename until it has been created and replicated
  - Btw, could we guarantee atomicity if we had more than one master server?
- Avoids two files with the same name

- Why not do this for all writes?
  - Pro-tip: It's slow
  - Analogy: remember how slow it would be if we had to wait for individual TCP packet ACKs?

Outline

- TCP and UDP

- **Google File System**
  - Rationale
  - Reading
  - Consistency and definition
  - **Write forwarding**
  - Fault tolerance
  - Strengths/Weaknesses

Write forwarding

- To guarantee consistency, writes need to be coordinated

How this produces consistency

- Goal is an ordering of which client wrote the chunk in which order
- There is only one primary for a chunk at once
- The primary decides on an order

Outline

- TCP and UDP

- **Google File System**
  - Rationale
  - Reading
  - Consistency and definition
  - Write forwarding
  - **Fault tolerance**
  - Strengths/Weaknesses

Fault tolerance

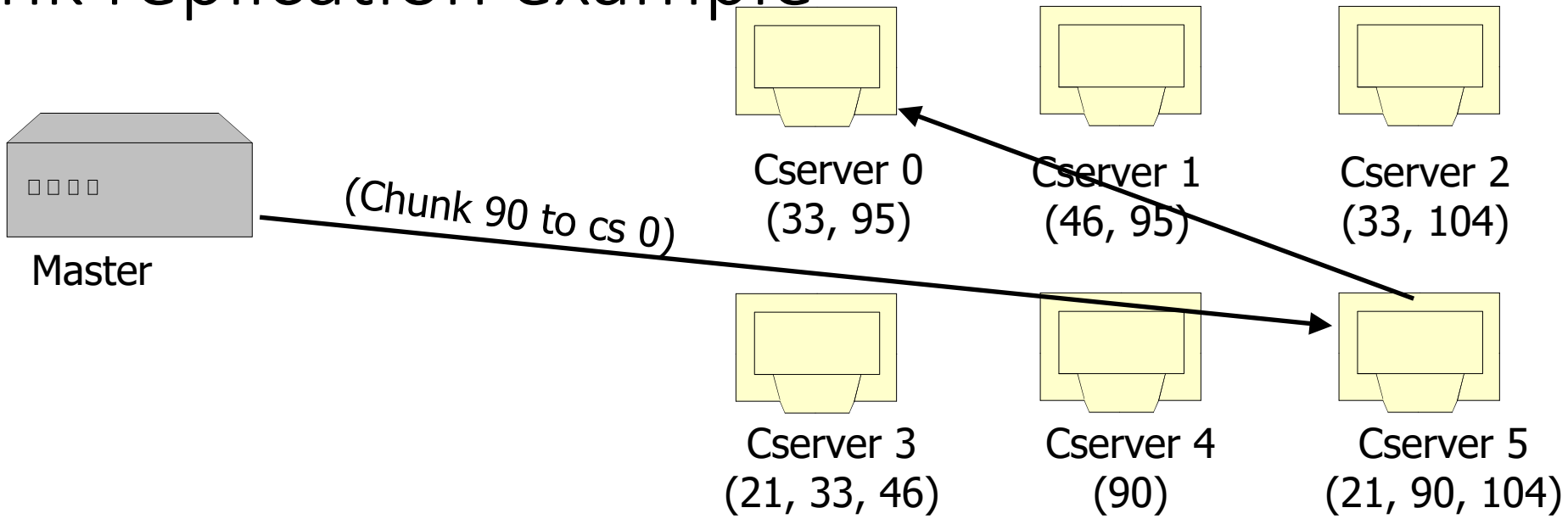
- Failed chunkserver
- Failed master

Chunkserver fault tolerance

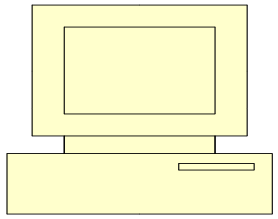
- Chunkservers report to master every few seconds
  - *Heartbeat* message
- If the master loses the heartbeat, marks the server as down
- Master asks chunkservers to reduplicate data

- Rule of thumb: 3 replicas of each chunk

Chunk replication example



1. Always keep at least *k* copies of each chunk
2. Imagine cserver 4 dies; #90 lost
3. Master loses heartbeat, decrements #90's reference count. Asks cserver 5 to replicate #90 to cserver 0
4. Choosing replication target is tricky



Chunk replication design decisions

- How many copies of each chunk?
- How do you choose a chunkserver target?

- GFS decisions:
  - As a rule, 3 copies, but configurable
  - A traditional rack of machines generally contains a bunch of servers (~20-40) and a network switch
  - Lots of intra-rack bandwidth, limited inter-rack bandwidth
  - Keep most copies within the rack

Master failure

- What if the master node goes down?
  - Entire system is down (lol whoops)
- Minutes down time
  - Automatic failover later added
- 10 seconds
  - Best achieved
  - Still too high!

Master failure

- Master maintains critical data structures
  - filename -> chunkid map
  - chunkid -> location map
- Master writes a log to disk when data structures change
  - **Shadow master** consumes log and keeps copies of these data structures up-to-date
- If master goes down, shadow master provides *read-only access* until master restart

Outline

- TCP and UDP

- **Google File System**
  - Rationale
  - Reading
  - Consistency and definition
  - Write forwarding
  - Fault tolerance
  - **Strengths/Weaknesses**

GFS strengths

- Store lots of data
- Fault tolerant
  - Too big to back up!
- High *throughput*
  - Can read lots of data from many chunkservers at same time

GFS weaknesses

- Bad for small files
  - Chunks are ~64MB
- Master node single point of failure
- High *latency*
  - Talk to two servers to fetch any data, might need multiple chunks