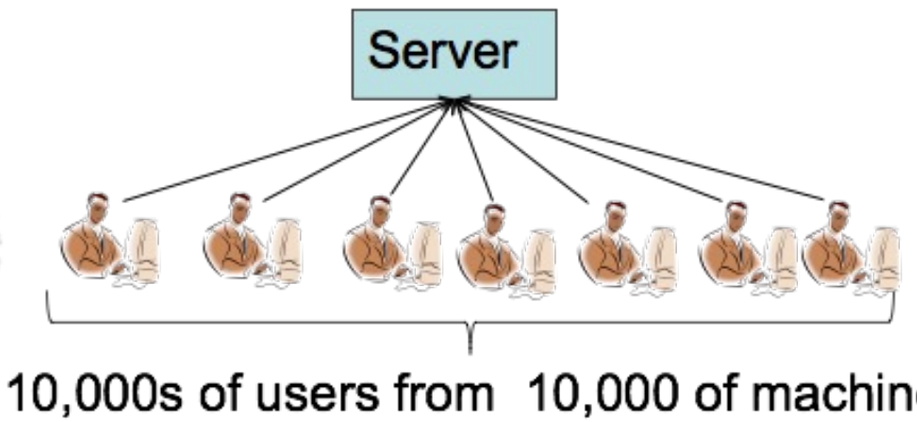
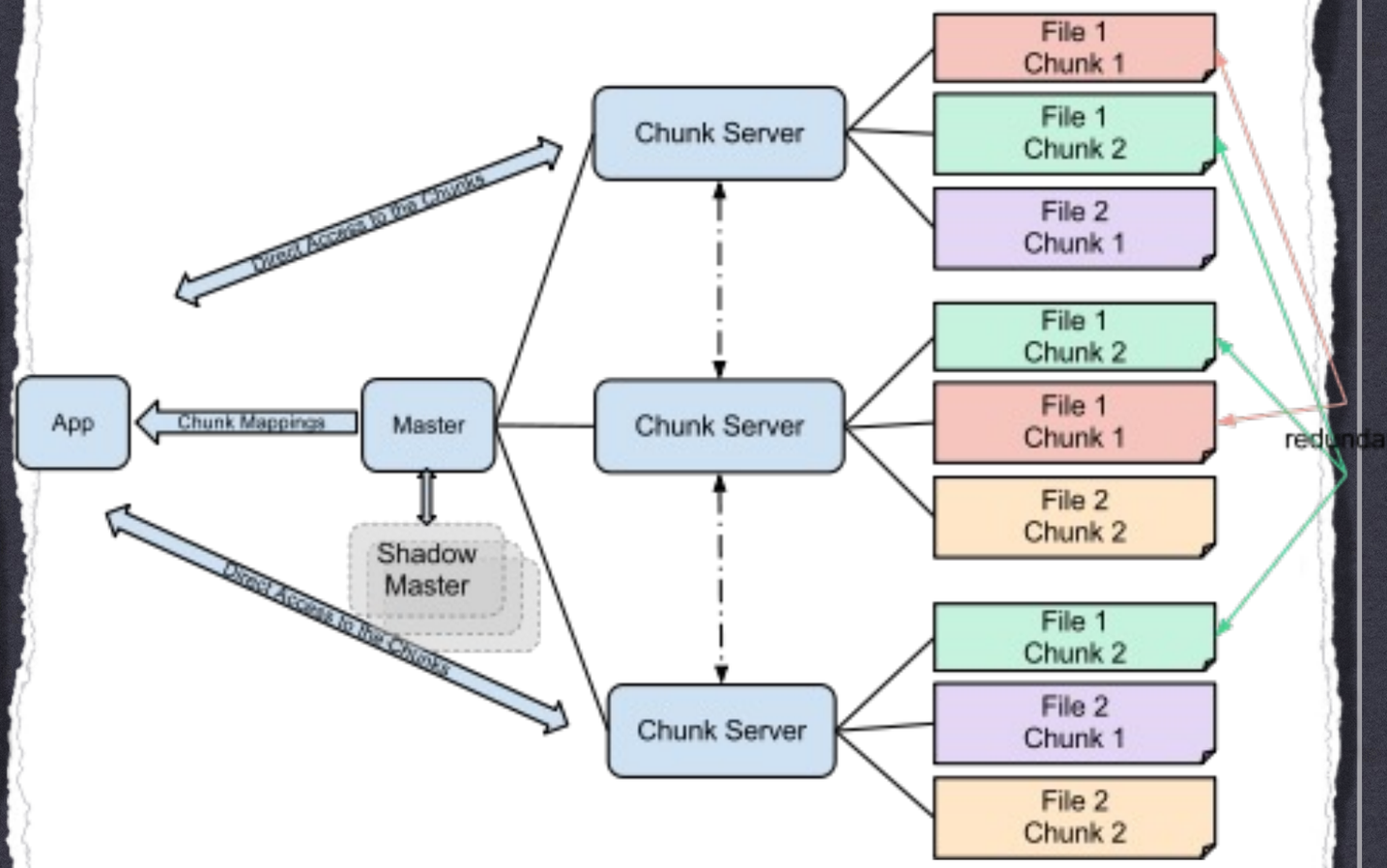



# DISTRIBUTED SYSTEMS

## DISTRIBUTED FILESYSTEMS

# Skype

# Skype

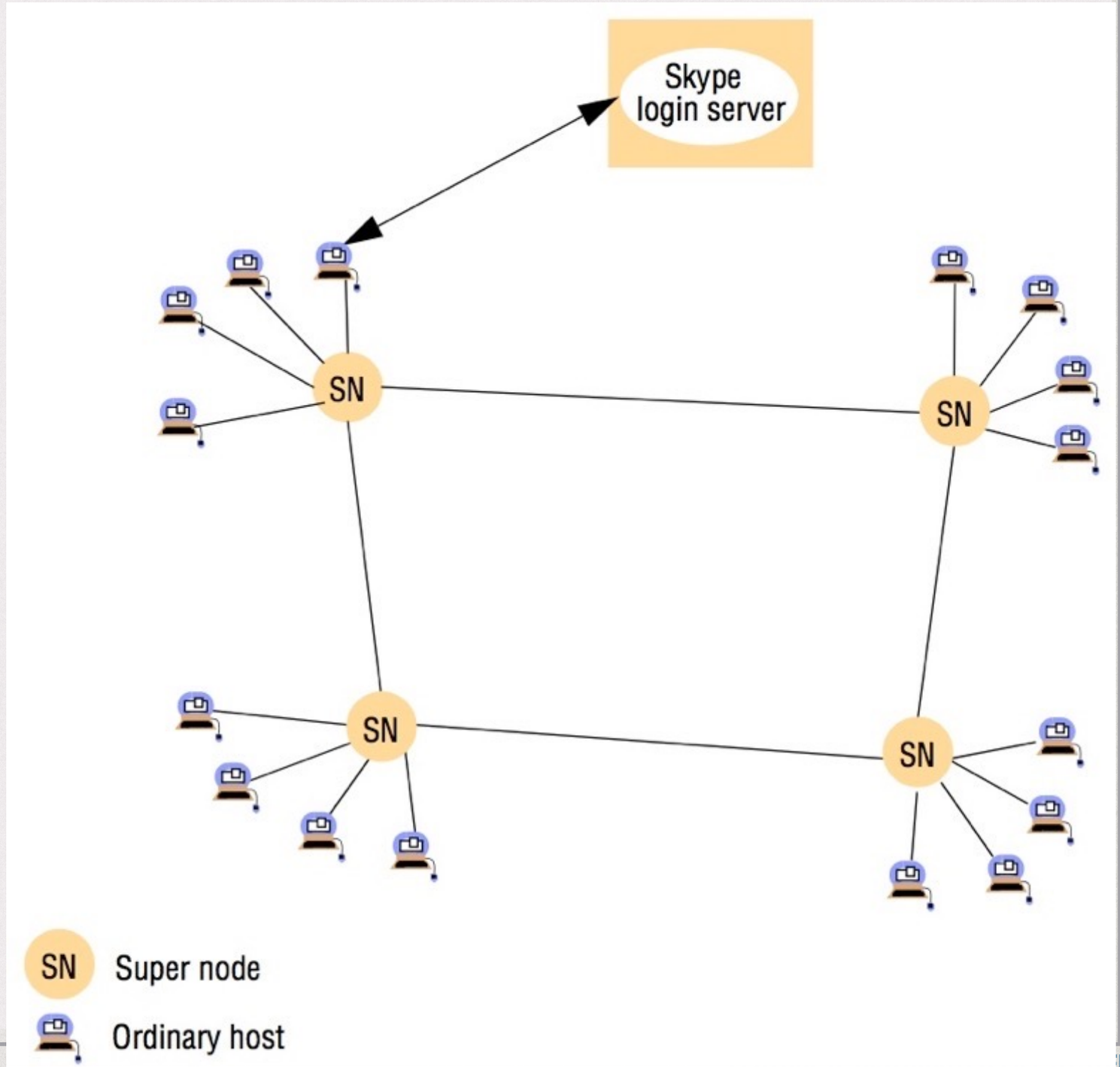
- Voice over IP peer-to-peer application
- Built by Kaaza in 2003
- Overlay network
  - Advanced functionality provided in an application-specific manner without modification of the core architecture of the Internet

# Skype

- Jan 2011: 27 million simultaneous online users
- March 2012: 35 million simultaneous online users
- June 2012: 70 million downloads on Android devices
- July 2012: over the quarter, Skype users have logged 115 billion minutes of calls
  - 218,798 years (218.8 millennia)

# Architecture

- P2P infrastructure
- Ordinary hosts
- Super nodes
  - Selected on demand
  - CPU
  - Bandwidth
  - Reachability
  - Availability



# User connection

- Users are authenticated via a well-known login server
- Users then contact a well known supernode
  - First time, around seven supernodes
  - “Host cache” continues to grow to hundreds of supernodes: IP and port
- Initial versions: the users’ buddy list & conversations were stored locally
- Multiple logins permitted: messages and calls routed to all user locations
  - 1

# Supernodes

- Main functionality: efficient search of the global index of users
  - Distributed among the supernodes
- Client contacts supernode -> supernode contacts on average 8 other supernodes: hashing + periodic flooding
- User search takes 3-4 seconds to complete for global IP addresses
- Intermediary nodes seem to cache results

# Voice connection

- Once user is discovered:
- TCP for call requests and terminations
- TCP or UDP for audio streaming (UDP preferred)
  - TCP + intermediary node to circumvent firewall
- Encoding and decoding: tailored to operate in Internet environments

# Skype

- May 2011: acquired by Microsoft
- May 2012: using only Microsoft-operated supernodes
- 2017: centralised Azure-based service
  - Transferred files are now saved on central servers

# Last lecture…

- Latency Reduction Issues
  - Network propagation delay is usually dominant
    - Latency reduction aims to reduce the mean count of propagation delays per remote call
- Dependencies constraints
- Abstractions to reduce propagations delays

# Latency Reduction with Parallel RPC

1

```
pcall {
    res1 = O1.m1(paramsA …)
    res2 = O2.m2(paramsB …)
    res3 = O3.m3(paramsC …)
}
```

- All three (or however many) calls are sent off together, in any order, then the client blocks until  the results of all calls have arrived back.
- *What is the overall latency for the calls in a parallel RPC?*
- *What is the average latency for the calls in a parallel RPC?*
- *What dependencies are permitted between the calls in a parallel RPC?*

# Latency Reduction with One-way RPC

2

- The ***One-way RPC*** idea is:

```
O1.m1(paramsA …)
res2 = O1.m2(paramsB …)
```

  - All except the last of this group of calls are one-way RPCs – they produce no result.

# Latency Reduction with Futures and Promises

3

- An example of the Future/Promises idea is:

```
res1 = O1.m1(paramsA …)
… computation …
res2 = O1.m2(paramsB …)
… computation …
… computation using res1 …
… computation using res2 …
res3 = O2.m3(paramsC …)
```

- Much more flexible than Parallel RPC!

# Latency Reduction with Batched Futures

4

- An example of the Batched Futures idea is:

```
res1 = O1.m1(paramsA …)
… computation …
res2 = O1.m2(…, res1, …)
```

- The result of the first call is able to be used as a parameter of the second call, without the possibility of blocking.

# Latency Reduction with Responsibilities

5

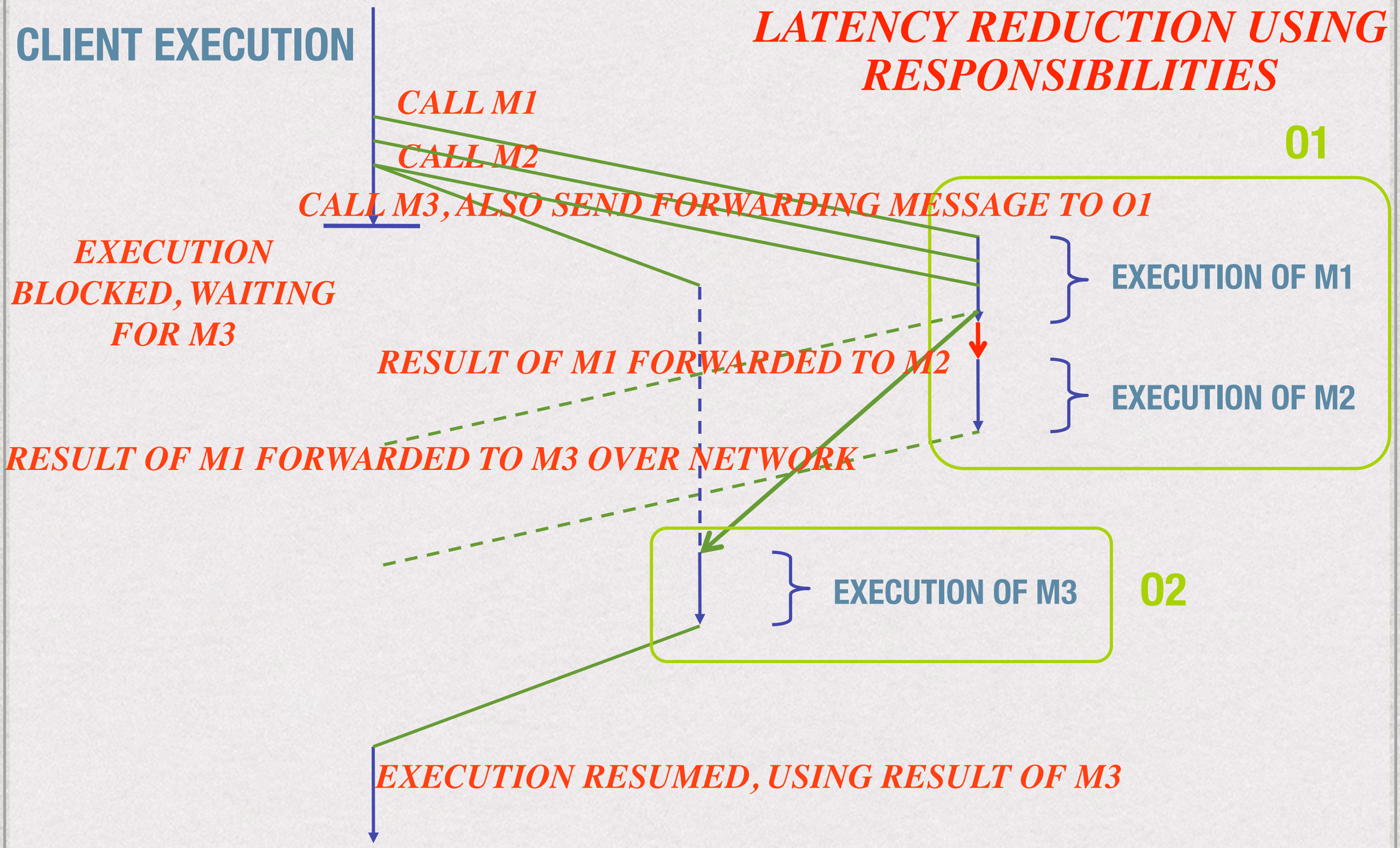
- An example of the Responsibilities idea is:

```
res1 = O1.m1(paramsA …)
res2 = O1.m2(…, res1, …)
res3 = O2.m3(…, res1, …)
x = res3 + 5
```

- With Batched Futures, the third call will block the client until res1 arrives, since the call is being sent to a different server (object).

- With Responsibilities, the third call is also non-blocking.

# LATENCY REDUCTION USING RESPONSIBILITIES



# Analyse That!

- One factor in requiring less blocking is the kinds of dependencies between calls that can be traversed without blocking:
  - Result dependent < Order dependent < Independent
- Another factor is whether dependencies between calls to different servers can be traversed without blocking:
  - Multi server < Single server

# Types of Dependencies

- Calls m1 and m2:
- ***Independent*** – m1 and m2 can execute in any order and the result of either is not used in the other.
- ***Order Dependent*** – m2 must begin execution after the end of the execution of m1, but m2 does not use the result of m1.
- ***Result Dependent*** – m2 takes as parameters one or more results of m1, and hence must begin execution after the end of the execution of m1:
- Res1 = O.m1(…)
  Res2 = O.m2(…, res1, …)

# Types of Dependencies

- ***Functionally Dependent*** – m2 takes as parameters one or more non-identity functions of one or more results of m1,
- must begin execution after the end of the execution of m1:
- Res1 = O.m1(…)
  Res2 = O.m2(…, res1 + 3 , …)

# This lecture ...

- Distributed file systems - one of the first uses of distributed computing
  - challenges
  - general design considerations
- NFS (Networked File System)
- GFS (Google File System)

# Challenges of DFS

- Heterogeneity (lots of different computers & users)
- Scalability
- Security (my files!  hands off!)
- Failures
- Concurrency
- Geographic distribution
- High latency

**HOW CAN WE BUILD THIS?**

# How to start?

- Prioritized list of goals
  - performance, scale, consistency - which one do you care more about?
- We are scientists! Therefore we measure and design, and revise
- Workload-oriented design:
  - Measure characteristics of target workloads to inform the design

# Workload Oriented Design

- User-oriented (NFS, AFS)
  - optimize how users use files
  - files are privately owned
  - not too much concurrent access
  - Sequential is common; reads more common than writes
- Program-oriented (GFS)
  - Focus on big-data workload: files are very very large
  - Failures are normal occurrences
  - Most files mutate by appending new data, not overwriting

# FS Interface

Client

...

Client

Client

Client

OPEN

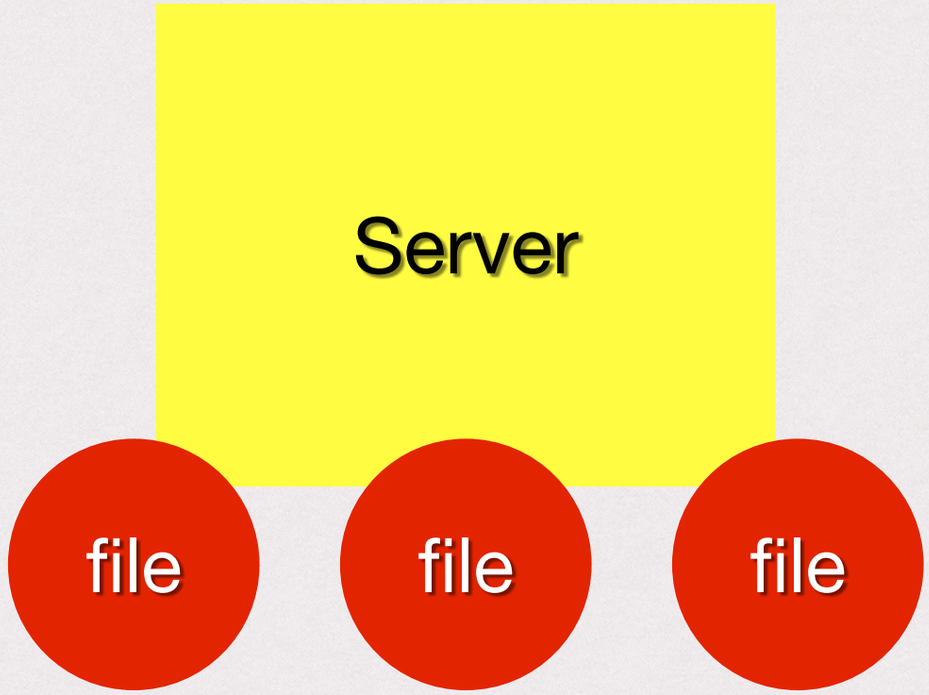
READ

WRITE

READ

WRITE

CLOSE



# Directory operations

- Create file
- create directory
- rename file
- delete file
- delete directory

# Naive DFS Design

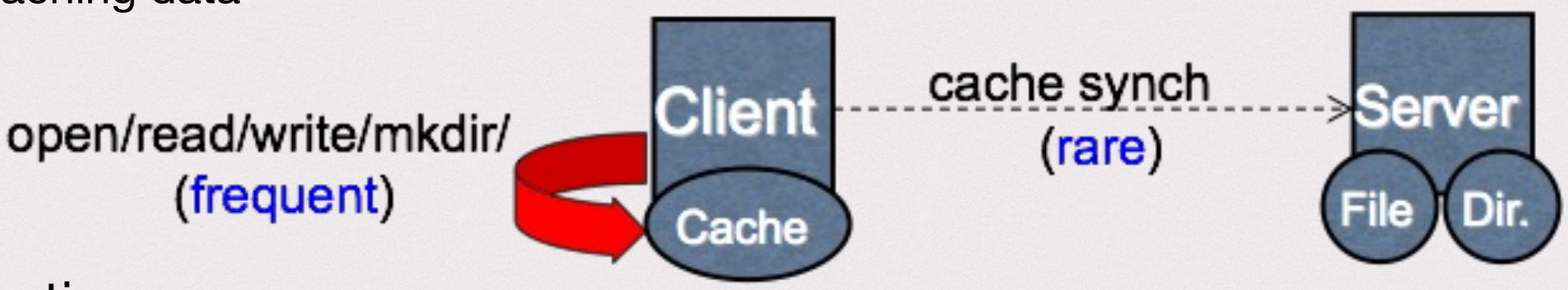
- Use RPC to forward *every* filesystem operation to the server
  - Server serializes all accesses, performs them, and sends back result.
- **Good**: Same behavior as if both programs were running on the same local filesystem!
- **Bad**: Performance will stink. Latency of access to remote server often much higher than to local memory.
- **Ugly**: server would get hammered!

**LESSON 1: NEEDING TO HIT THE SERVER FOR EVERY DETAIL IMPAIRS PERFORMANCE AND SCALABILITY.**

**QUESTION: HOW CAN WE AVOID GOING TO THE SERVER FOR EVERYTHING? WHAT CAN WE AVOID THIS FOR? WHAT DO WE LOSE IN THE PROCESS?**

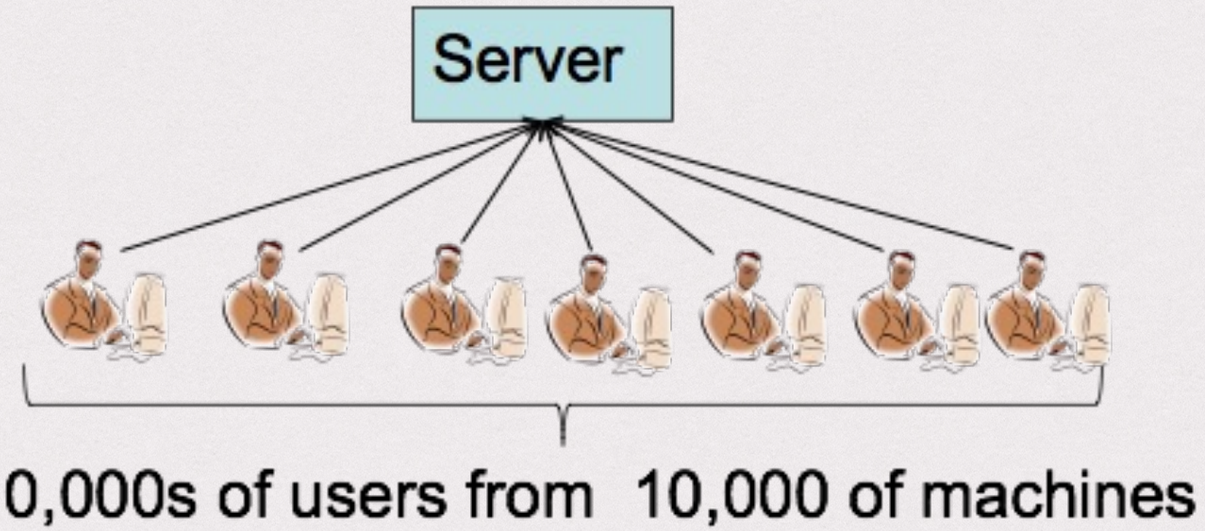
# Solution: Caching

- Lots of systems problems can be solved in two ways:
  - Adding a level of indirection
    - “All problems in computer science can be solved by adding another level of indirection.  But that will usually create another problem.” -- D. Wheeler
  - Caching data



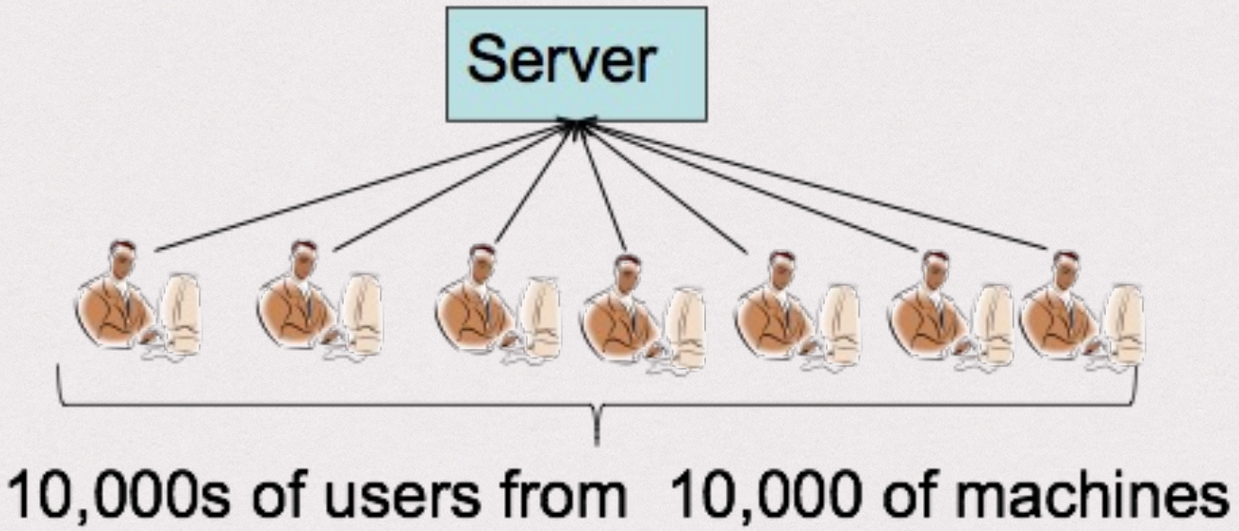

- Questions:
  - What do we cache?
  - If we cache... doesn’t that risk making things inconsistent?

# Sun NFS




- First commercially successful network file system:
- Developed by Sun Microsystems for their diskless workstations
- Designed for robustness and “adequate performance”
- Sun published all protocol specifications
- Many many implementations

# Sun NFS




- Networked file system with a single-server architecture
- Goals
  - consistent namespace for files across computers
  - allow authorized users to access their files form any computer

# Sun NFS

- Cache file blocks, file headers, etc., at both clients and servers.

- Advantage:  No network traffic if open/read/write/close can be done locally.

- But:  failures and cache consistency.

  - NFS trades some consistency for increased performance

# Caching problem #1: Failures

- **Server crashes**
  - Data in memory but not disk lost
  - So... what if client does `seek() ; /* SERVER CRASH */; read()`
    - If server maintains file position, this will fail.
    - same for open(), read()
- **Lost messages**: what if we lose acknowledgement for delete("foo")
  - And in the meantime, another client created foo again?
- **Client crashes**
  - Might lose data in client cache

# NFS's Solutions

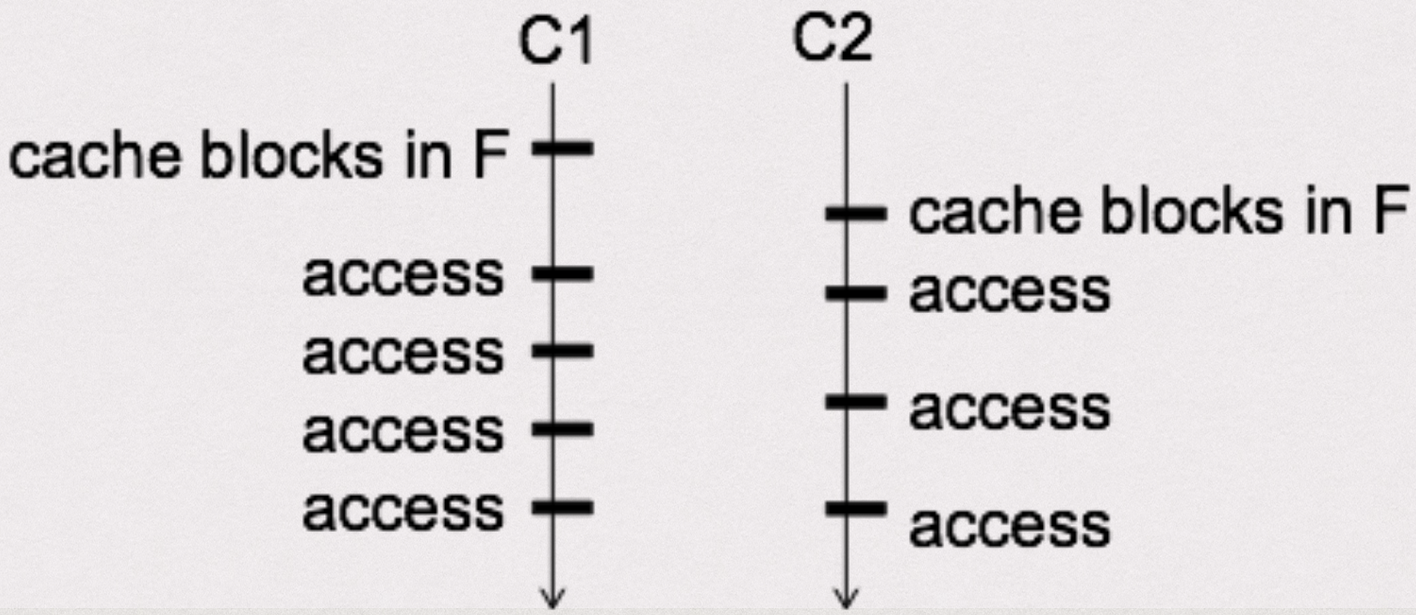
- **Stateless design**
  - Write-through caching: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
  - Stateless protocol: requests specify exact state. read() -> read( [position]). no seek on server.
- Operations are ***idempotent***
  - Have the same effect no matter how many times they are repeated
  - How can we ensure this? Unique IDs on files/directories. It's not delete("foo"), it's delete(1337f00f), where that ID won't be reused.

# Design tip

- **Idempotency** is a useful property when building reliable systems. When an operation can be issued more than once, it is much easier to handle failure of the operation; you can just retry it. If an operation is not idempotent, life becomes much more difficult!

# Caching problem #2: Consistency

- If we allow client to cache parts of files, file headers, etc.
- What happens if another client modifies them?



# Solution: Weak Consistency

- NFS flushes updates on close()
- How does other client find out?
- NFS's answer: It checks periodically.
  - This means the system can be inconsistent for a few seconds: two clients doing a read() at the same time for the same file could see different results if one had old data cached and the other didn't.

# Design choice

- Clients can choose a stronger consistency model: *close-to-open* consistency
  - How? Always ask server before open()
  - Trades a bit of scalability for better consistency

# What about multiple writes?

- NFS provides no guarantees at all!
- Might get one client's writes, other client's writes, or a mix of both!

# NFS and Failures

- You can choose -
  - retry until things get through to the server
  - return failure to client
- Most client apps can't handle failure of close() call. NFS tries to be a transparent distributed filesystem -- so how can a write to local disk fail? And what do we do, anyway?
- Usual option: hang for a long time trying to contact server
- More about failures next time!

# Summary

- NFS provides transparent, remote file access
- Simple, portable, *really popular*
  - (it's gotten a little more complex over time, but...)
- Weak consistency semantics
- Requires hefty server resources to scale (write-through, server queried for lots of operations)

# Google File System

- Using the Google SOSP 2003 paper:
  - *The Google File System*, S. Ghemawat, H. Gobioff, S-T Leung
  - some details might be out of date

# Design Constraints

- Machine failures are the norm
  - 1000s of components
  - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
  - Monitoring, error detection, fault tolerance, automatic recovery must be integral parts of a design
  - Jeff Dean (2008): In each cluster's first year, it's typical that 1,000 individual machine failures will occur; thousands of hard drive failures will occur; one power distribution unit will fail, bringing down 500 to 1,000 machines for about 6 hours; 20 racks will fail, each time causing 40 to 80 machines to vanish from the network; 5 racks will "go wonky," with half their network packets missing in action; **50% chance that the cluster will overheat**
- Big-data workloads
  - Search, ads, web analytics, Map/Reduce

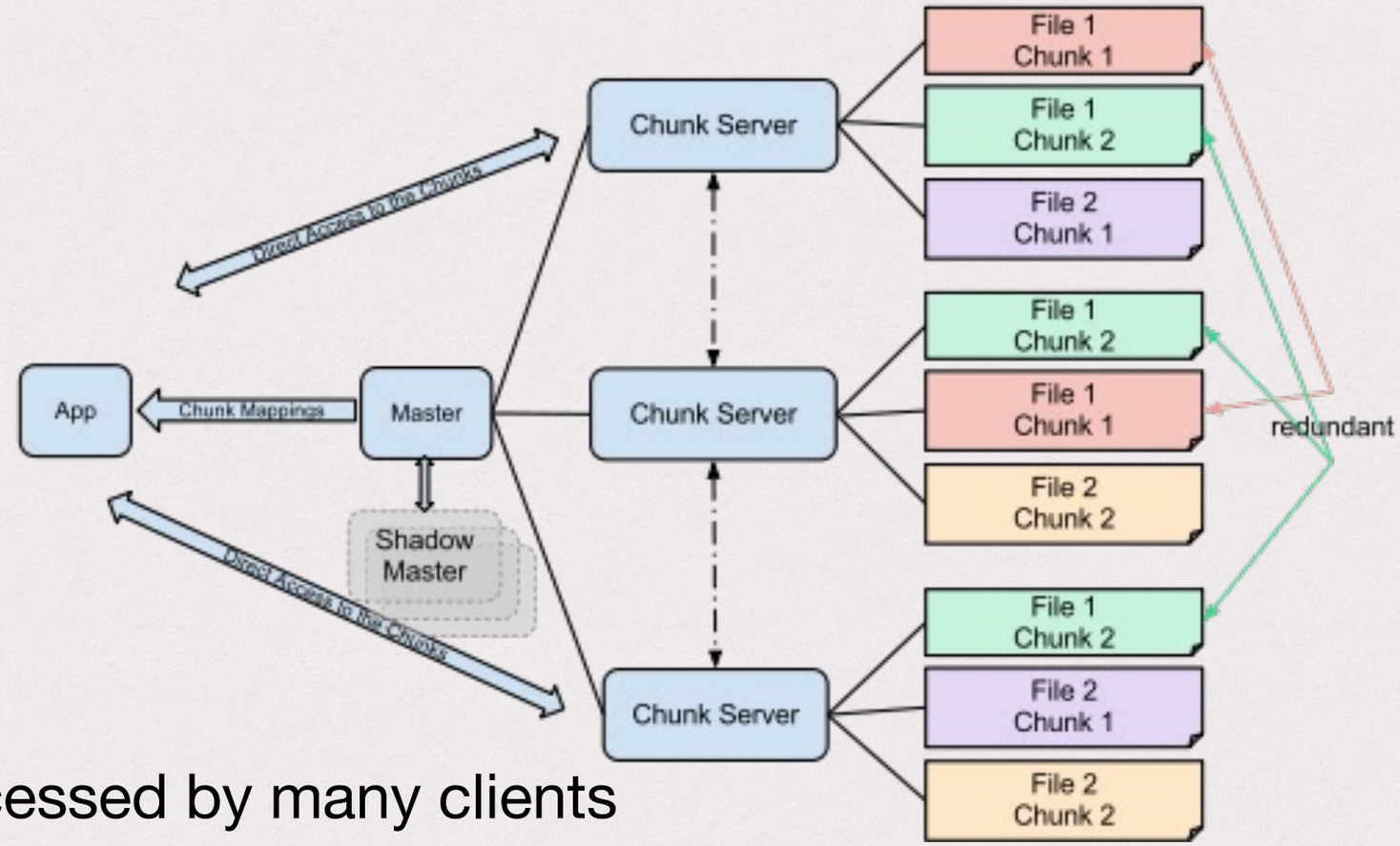
# Workload Characteristics

- Files are huge by traditional standards
  - Multi-GB files are common
- Most file updates are appends
  - Random writes are practically nonexistent
  - Many files are written once, and read sequentially
- High bandwidth is more important than latency
  - Lots of concurrent data accessing: multiple crawler workers updating the index file
- GFS' design is geared toward apps' characteristics
  - and Google apps have been geared toward GFS

# Additional operation

- Record append
  - Frequent operation at Google
  - Merging results from multiple machines in one file (Map/Reduce)
  - Using file as producer - consumer queue
  - Logging user activity, site traffic
  - Order doesn’t matter for appends, but atomicity and concurrency matter

# Design

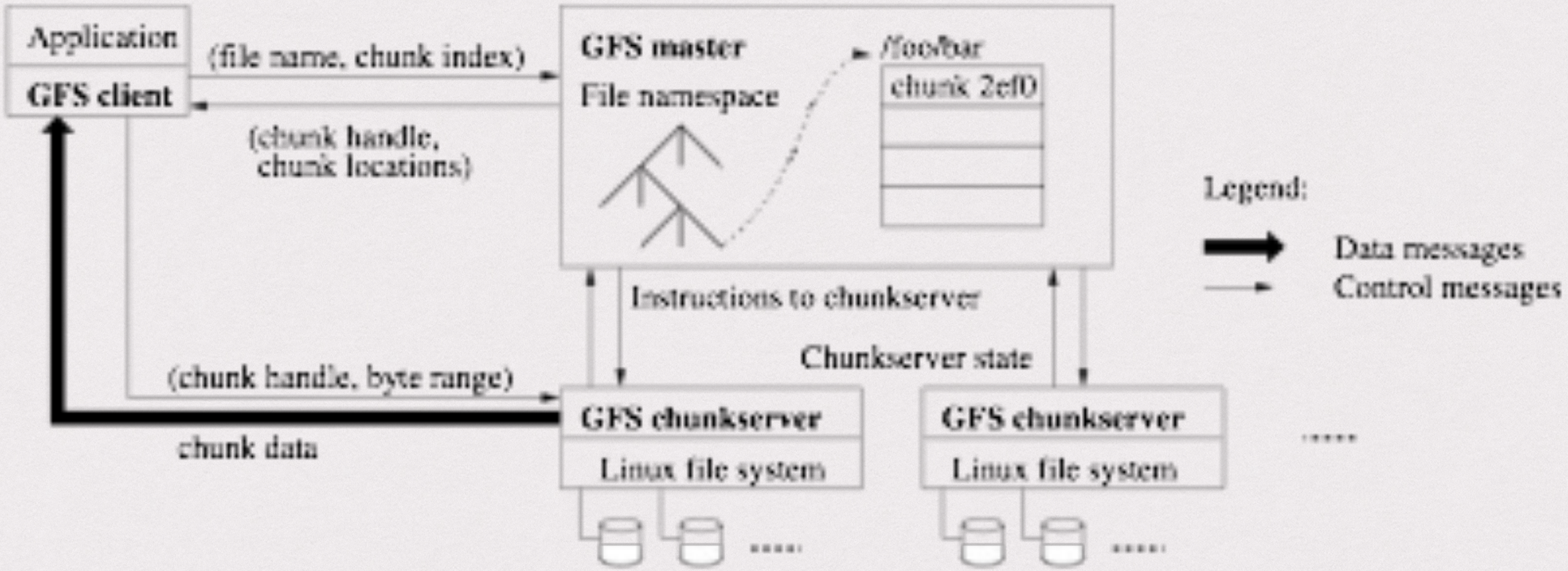



- A GFS cluster
  - One master
  - Many chunkservers - accessed by many clients
- A file
  - Divided into fixed-sized chunks
  - Labeled with 64-bit unique global IDs (called handles)
  - Stored at chunkservers
  - 3-way replicated across chunkservers
  - Master keeps track of metadata: which chunks belong to which files

# Basic Operations

- Client retrieves metadata from master
- Read/write data from client to chunckserver
- Master really not that involved: not a bottleneck

# In detail ...


Application
GFS client
(file name, chunk index)
(chunk handle,
chunk locations)
GFS master
File namespace
/foo/bar
chunk 2ef0
Instructions to chunkserver
Chunkserver state
(chunk handle, byte range)
chunk data
GFS chunkserver
Linux file system
GFS chunkserver
Linux file system
Legend:
Data messages
Control messages

# Chunks

- Similar to FS blocks, but bigger!
- 64 MB (contrary to 512KB - 8KB)
- **Advantages** of  a large chunk size?
- **Disadvantage**?

# Chunks

- Similar to FS blocks, but bigger!
- 64 MB (contrary to 512KB - 8KB)
- **Advantages** of  a large chunk size?
  - Less load on the server
  - suitable for big data
  - sustains large bandwidth
- **Disadvantage**?
  - what if small files are more frequent than initially believed?

# GFS Master

- One process running on a separate machine
  - at a later stage shadow masters were added for fault-tolerance
- Stores metadata in memory (fast!)
  - 64 bytes for 64MB of data
- Metadata types
  - File and chunk namespaces (hierarchical and flat respectively)
  - File-to-chunk mappings
  - Location of chunk’s replicas

# Master <->Chunkserver communication

- Heartbeat messages (regular communication)
  - Is chunkserver down?
  - Are there disk failures on chunkserver?
  - Are any replicas corrupted?
  - Which chunks does chunkserver store?
- Master sends instructions:
  - Delete chunk, create chunk
  - Replicate and start serving a particular chunk

# Chunk Locations

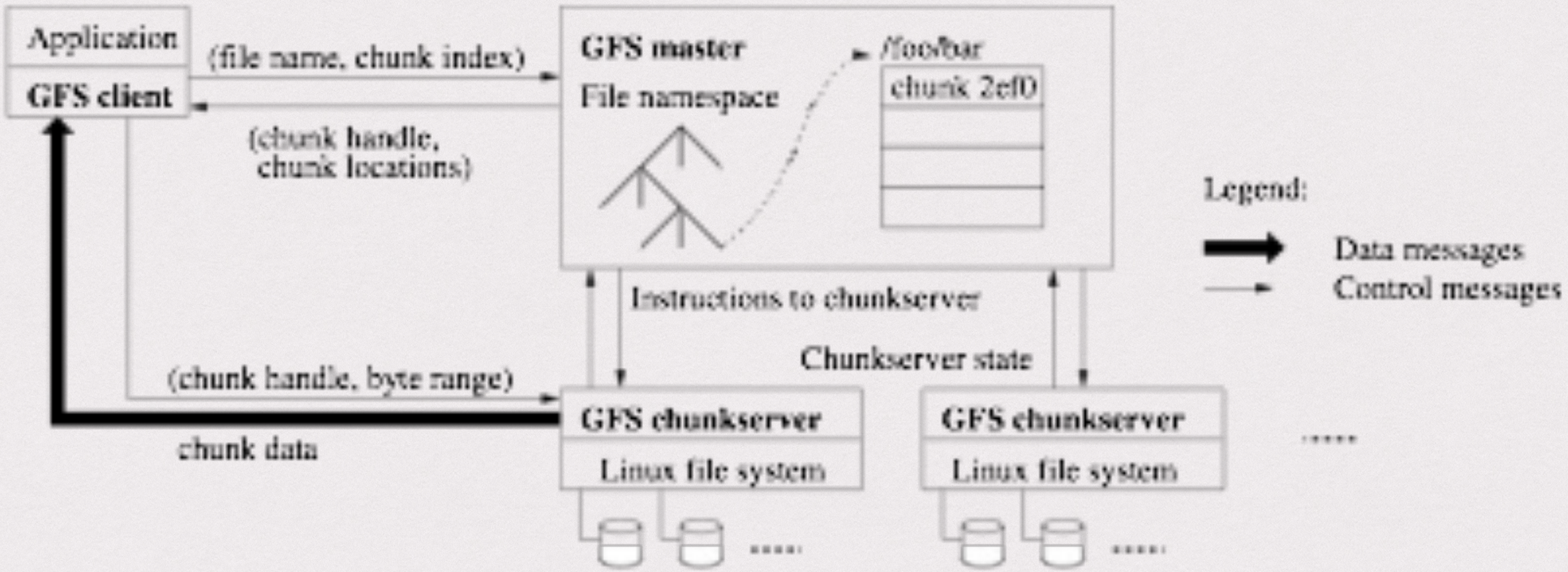
- No persistent location
  - Master polls chunkservers at startup
  - Use heartbeat messages to monitor servers
- **Advantages? (what if master dies?)**
- **Disadvantages? (what if master dies?)**

# Chunk Locations

- No persistent location
  - Master polls chunkservers at startup
  - Use heartbeat messages to monitor servers
- **Advantages?**
  - **if master dies, can recover chunks from chunkservers**
- **Disadvantages?**
  - **if master dies, it takes a loooong time to restart**

# Read Protocol

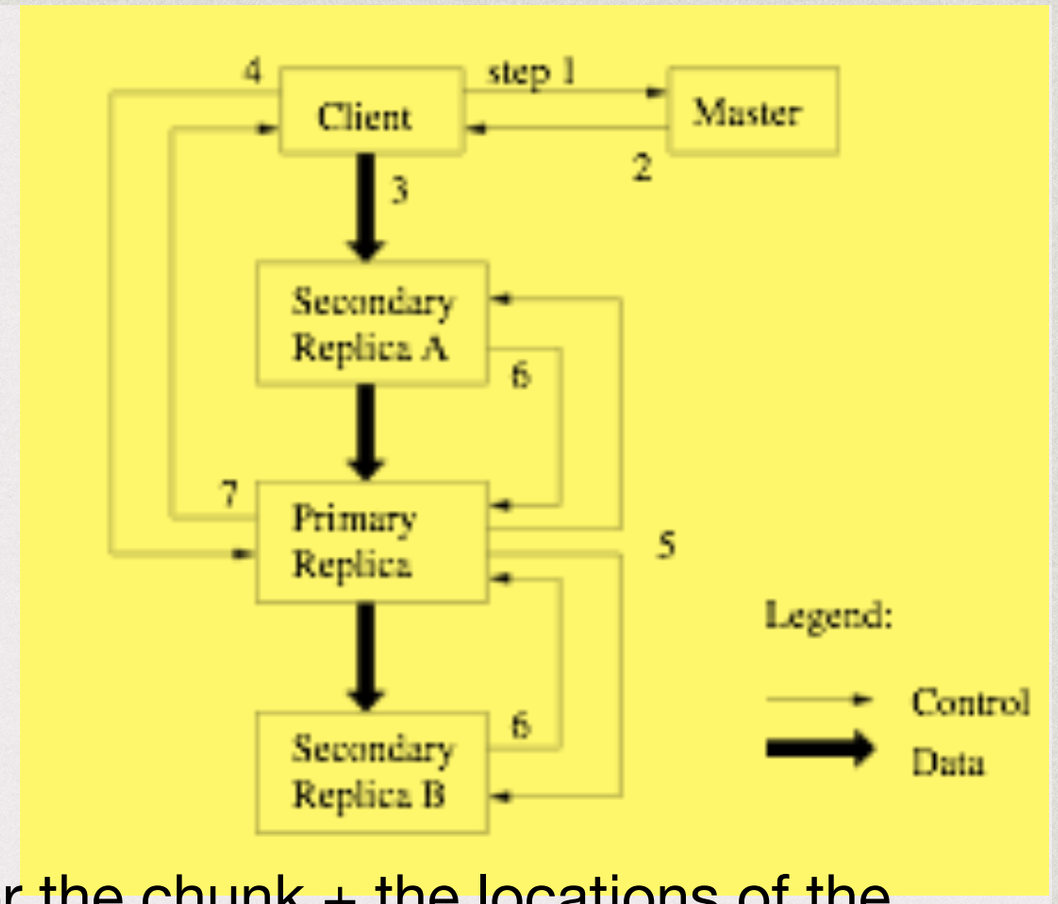
- Group activity: design the read protocol for GFS



# Read Protocol - Part 1

# Read Protocol - Part 2

# Write Protocol




1. Client to master
   - which chunkserver holds the current lease for the chunk + the locations of the other replicas.
2. Master replies with the identity of the primary and the locations of the other (secondary) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary dies
3. The client pushes the data to all the replicas.
4. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary.

# Replica placement

- Hundreds of chunkservers distributed across many many machine racks
- Chunk replica placement needs to
  - maximize data reliability and availability
  - maximize network bandwidth utilization
- Not sufficient to spread replicas across machines
  - need to spread across racks to ensure replicas survive even if an entire rack is down
- Chunks are re-replicated as soon as the number of replicas falls below a goal
  - need to place replicas on chunkservers with below average disk utilization

# Performance - 2 GFS clusters

| Cluster | A | B |
|---|---|---|
| Chunkservers | 342 | 227 |
| Available disk space | 72 TB | 180 TB |
| Used disk space | 55 TB | 155 TB |
| Number of files | 735k | 737k |
| Number of dead files | 22k | 232k |
| Number of chunks | 992k | 1550k |
| Metadata at chunservers | 13 GB | 21GB |
| Metadata at master | 48 MB | 60MB |

# Performance - 2 GFS clusters

| Cluster | A | B |
|---|---|---|
| Read rate (last minute) | 583 MB/s | 380 MB/s |
| Read rate (last hour) | 562 MB/s | 384 MB/s |
| Read rate (since restart) | 589 MB/s | 49 MB/s |
| Write rate (last minute) | 1 MB/s | 101 MB/s |
| Write rate (last hour) | 2 MB/s | 117 MB/s |
| Write rate (since restart) | 25 MB/s | 13 MB/s |
| Master ops (last minute) | 325 Ops/s | 533 Ops/s |
| Master ops (last hour) | 381 Ops/s | 518 Ops/s |
| Master ops (since restart) | 202 Ops/s | 347 Ops/s |

# Fault Tolerance in Distributed File Systems

- Many options...
  - Do nothing --> NFS
  - Hot, consistent replicas (every change affects multiple servers in case one dies)
  - Consistent snapshots - filesystem backup (beautiful but costly to make)
  - more about this in the next lecture ...