CS 4530
Software Engineering

Lecture 4 - Software Architecture

Jonathan Bell, John Boyland, Mitch Wand
Khoury College of Computer Sciences

Zoom Mechanics

- Recording: This meeting is being recorded
- If you feel comfortable having your camera on, please do so! If not: a photo?
- I can see the zoom chat while lecturing, slack while you're in breakout rooms
- If you have a question or comment, please either:
  - "Raise hand" - I will call on you
  - Write "Q: <my question>" in chat - I will answer your question, and might mention your name and ask you a follow-up to make sure your question is addressed
  - Write "SQ: <my question>" in chat - I will answer your question, and not mention your name or expect you to respond verbally

Today’s Agenda

Administrative:

- HW1 Discussion, due THIS Friday
- Please watch all 4 lesson videos for Thursday

Today’s lecture:

- Poll
- Overview of 4 software architectures and 4 software quality attributes
- Real-world system architectures

CS 4530 - Architecture Poll

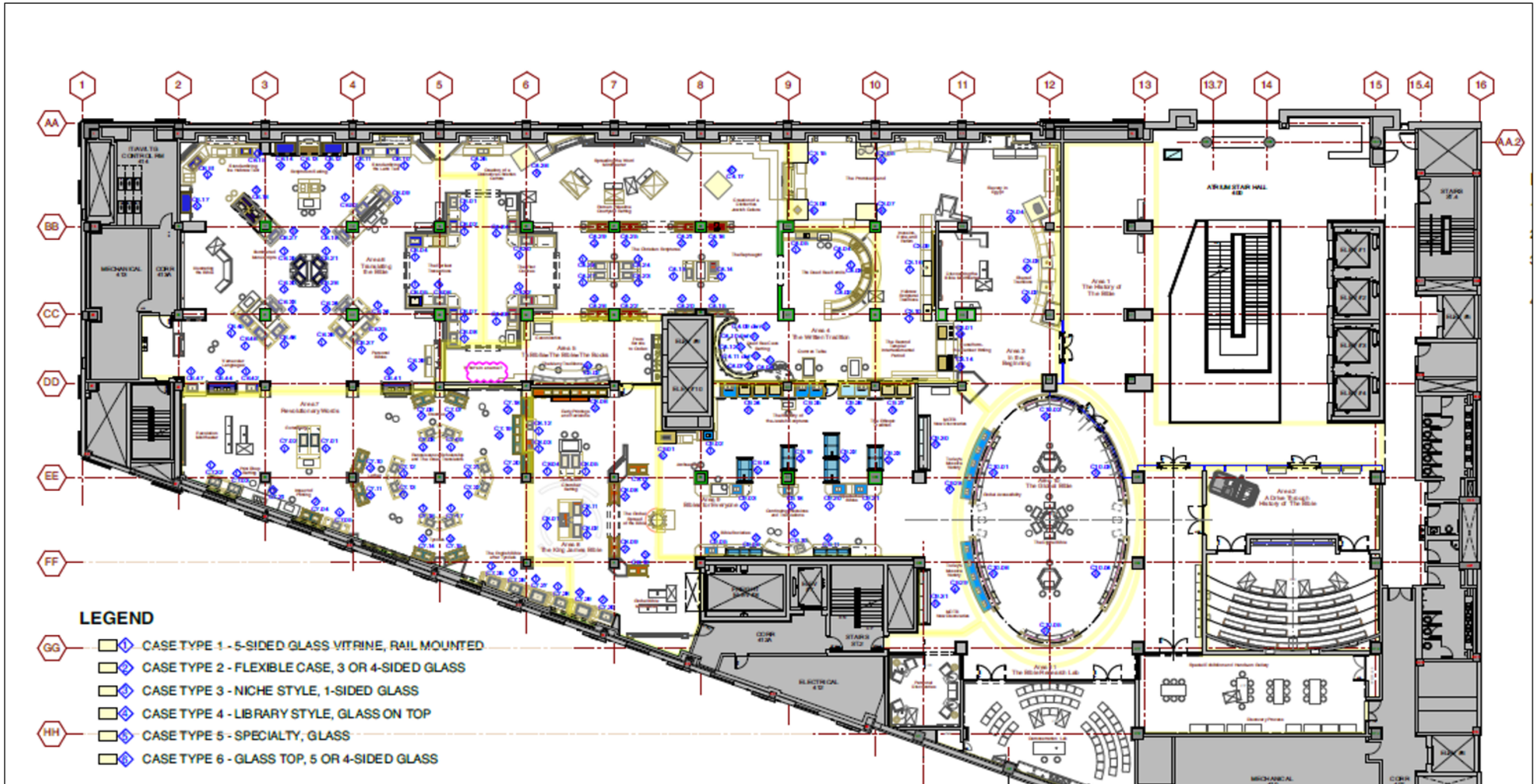
To complete the survey, go to PollEv.com/jbell

0 done

0 underway

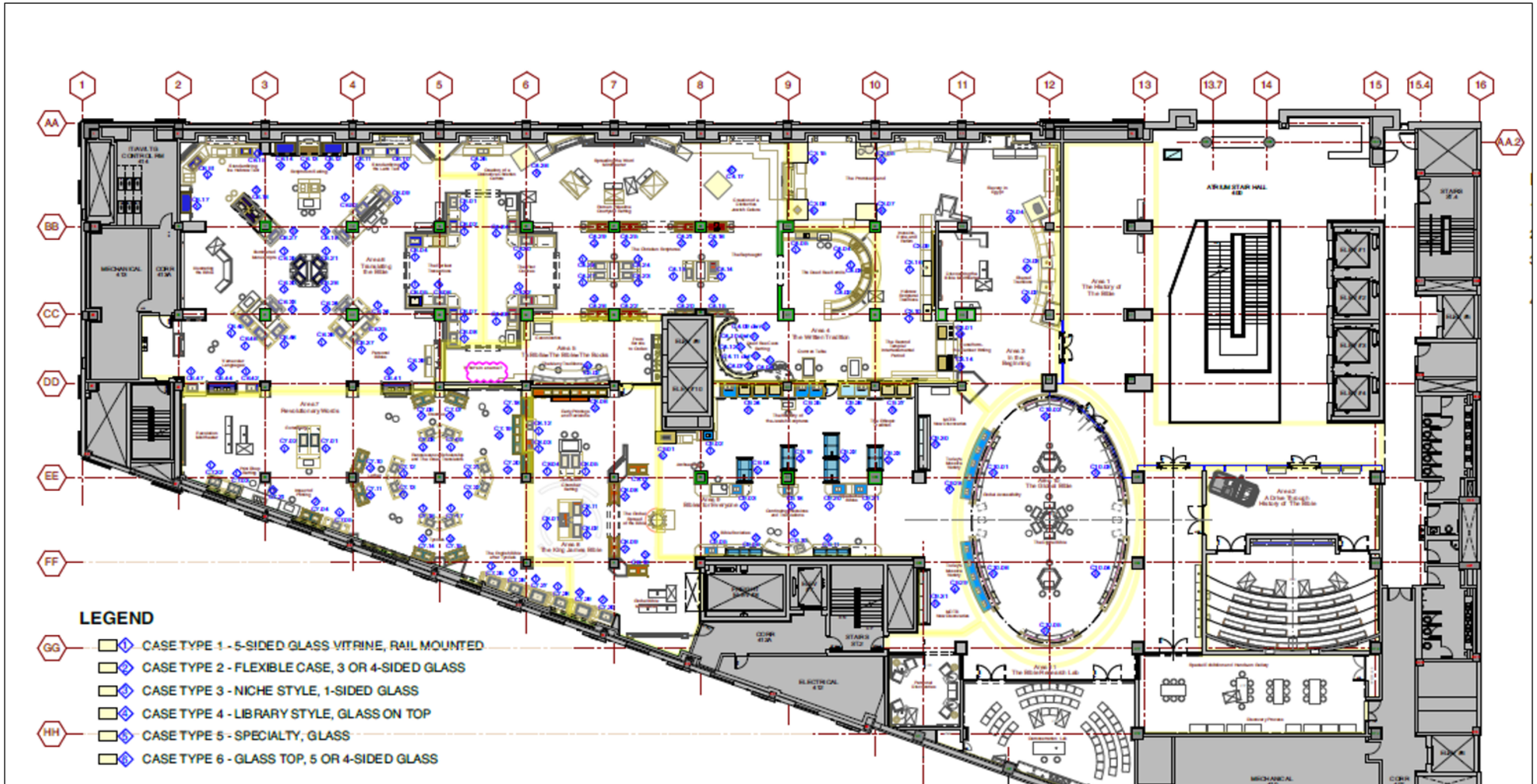
Focus for this week: zoom out

Metaphor: building architecture



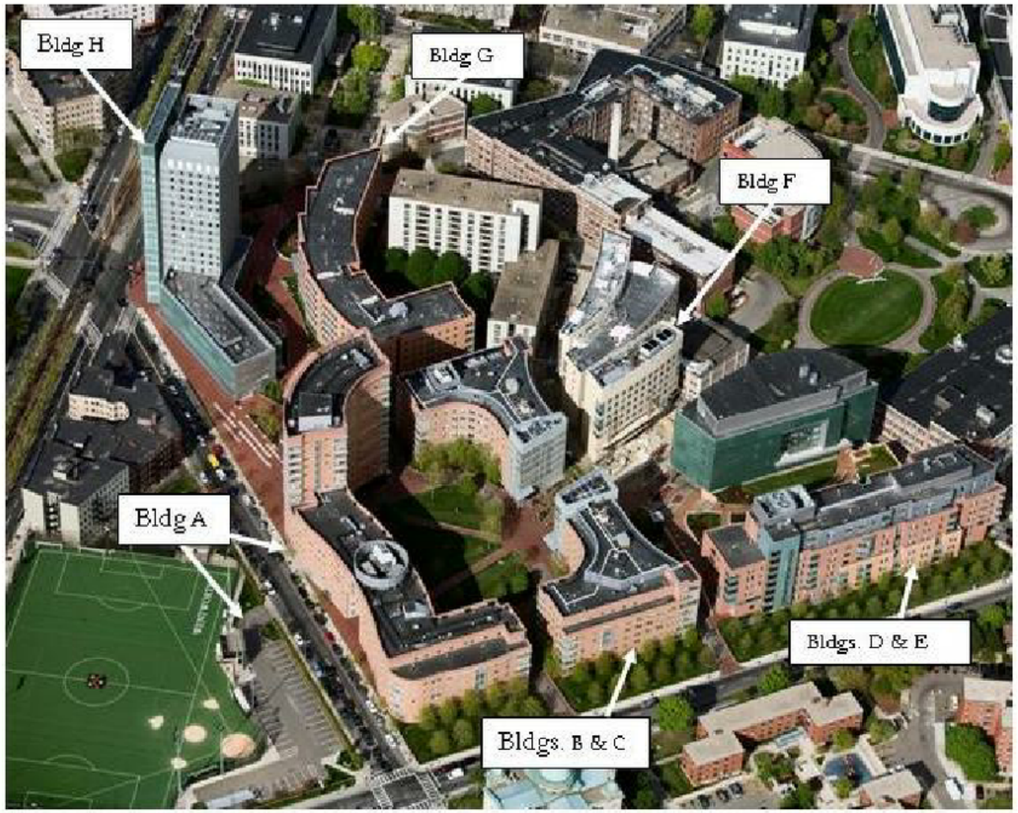
Focus for this week: zoom out

How do the pieces fit together? What do we reuse?



Focus for this week: zoom out

Architectural Patterns



Choose wisely

Architecture can make a HUGE difference in quality attributes like…

- Performance
- Reliability
- Scalability
- Cost
- Plus many more quality attributes, see Lesson 3.1

Architecture #0: Monolithic

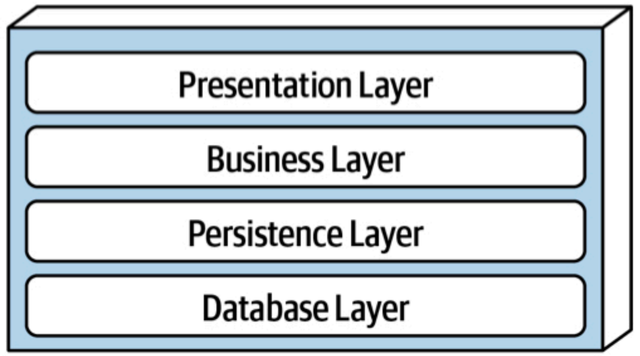
- A single app, with no particular organization
- Also known as: "spaghetti code"
- May still have useful interfaces for some degree of encapsulation and modularity.
  - but is there a method to his madness?

Shakespeare, *Hamlet*. The exact quote is: "Though this be madness, yet there is method in't" (Polonius, Act 2, Scene 2)

Brian Foote and Joe Yoder

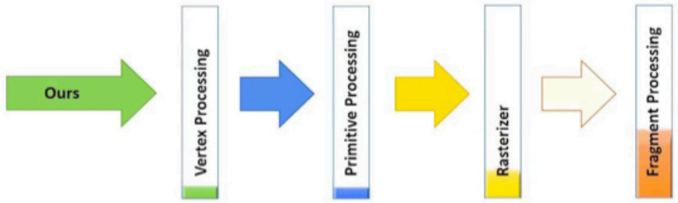
Architecture #1: Layered

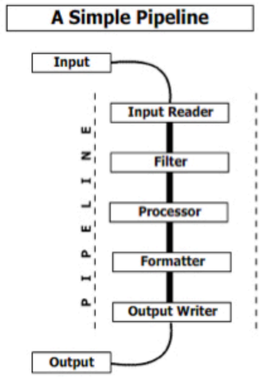
- Each layer depends on services from the layer or layers below
- Organize teams by Layer
  - different layers require different expertise
- When the layers are run on separate pieces of hardware, they are sometimes called "tiers"



Architecture #2: Pipeline

- Good for complex straight-line processes, eg:
  - image processing





Architecture #5: Microservices

- Overall task is divided into different components
- Each component is implemented independently
- Each component is
  - independently replaceable,
  - independently updatable
- Components can be built as libraries, but more usually as web services
  - Services communicate via HTTP, typically REST (see lesson 3.3)

Networks as Abstractions

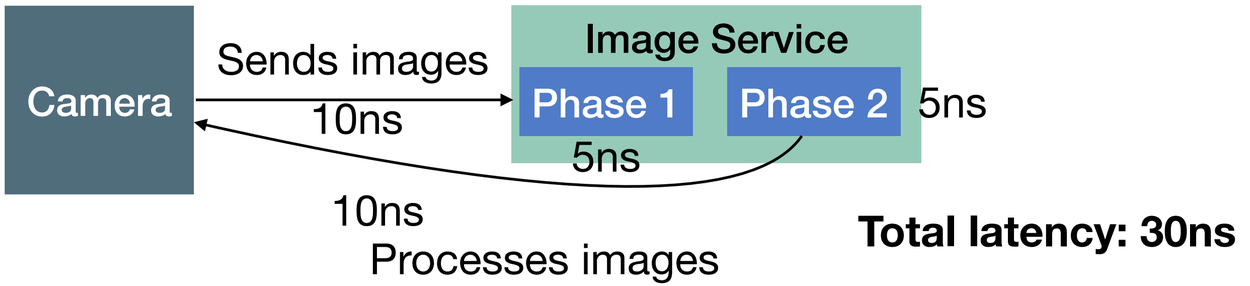
CS 3700, Summarized to a Slide

- A network consists of communication links. Ideally: “Stuff goes in, stuff comes out”
- Networks have several “interesting” properties we will look at
  - Latency
  - Failure modes

Designing for Performance: Metrics

Latency in a pipeline architecture

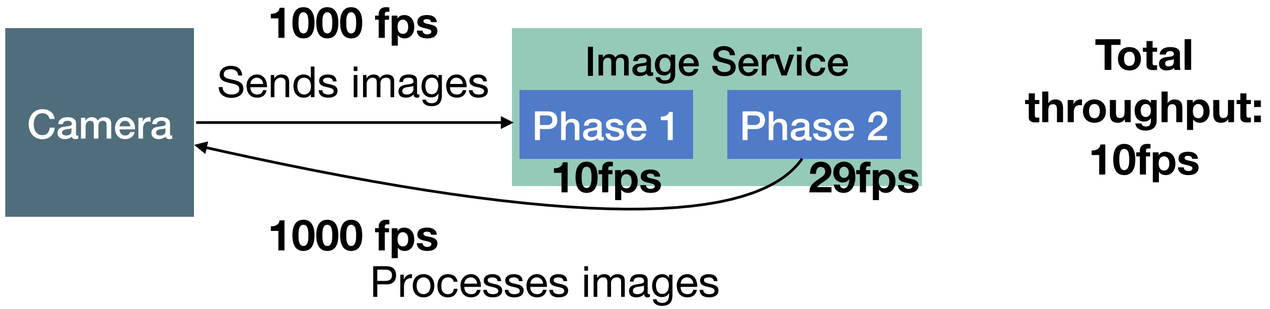
- Time between client sending request and receiving response
- What contributes to latency?
  - Latency sending the message
  - Latency processing the message
  - Latency sending the response
- Adding pipelined components -> latency is cumulative



Designing for Performance: Metrics

Throughput in a pipeline architecture

- Measure of the rate of useful work done for a given workload
- Example:
  - Throughput is camera frames processed/second
- When adding multiple pipelined components -> throughput is the minimum value



Designing for Performance: Metrics

Latency (In general)

- Often more challenging to improve than increasing throughput
  - Examples:
    - Physical - Speed of light (network transmissions over long distances)
    - Algorithmic - Looking up an item in a hash table is limited by hash function
    - Economic - Adding more RAM gets expensive

Designing for Performance: Case Study

Stock Trading

- Buy low/sell high
- Most of skill is in knowing what a stock will do **before** your competitors

Designing for Performance: Case Study

Algorithmic Trading

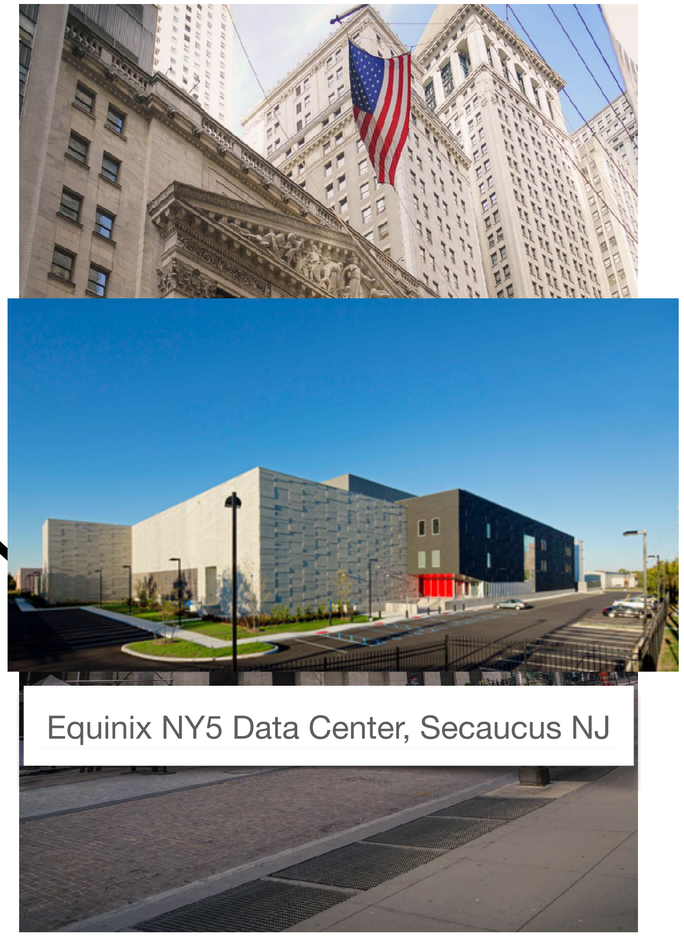
CME Data Center, Aurora, IL

Chicago Mercantile Exchange, Chicago, IL

~700 miles

Round trip:
~14 msec @ speed of light

3Ghz CPU would process
~22m instructions in this time!



Equinix NY5 Data Center, Secaucus NJ

NY Stock Exchange, New York, NY

Photos:
CME Building: By CME Group/Henry Delforn/Allan Schoenberg - CME Group, CC BY-SA 3.0
NYSE Building: By Jeffrey Zeldman, CC BY 2.0
CME data center: by CME Group
NY5 data center: by Equinix Inc

Designing for Performance: Case Study

Or: Reducing Latency with Billions of Dollars

ORIGINAL CABLE

Technology
Buried fiber-optic cable

Completion
Mid-1980s

Path length
~ 1,000 miles

Round-trip time for data
14.5
milliseconds and up

Approach
Multiple routes followed the easiest rights-of-way—along rail lines. But that means time-sucking jogs and detours.

SPREAD NETWORKS

Technology
Buried fiber-optic cable

Completion
August 2010

Path length
825 miles

Round-trip time for data
13.1
milliseconds

Approach
Spread bought its own rights-of-way, avoiding a Philadelphia-ward dip in favor of a shorter path northwest through central Pennsylvania.

MCKAY BROTHERS

Technology
Microwave beams through air

Completion
July 4, 2012

Path length
744 miles

Round-trip time for data
9
milliseconds

Approach
Microwaves generally move faster than photons in optical fiber, and McKay's network uses just 20 towers on a nearly perfect great circle.

TRADEWORX

Technology
Microwave beams through air

Completion
Winter 2012

Path length
~ 731 miles

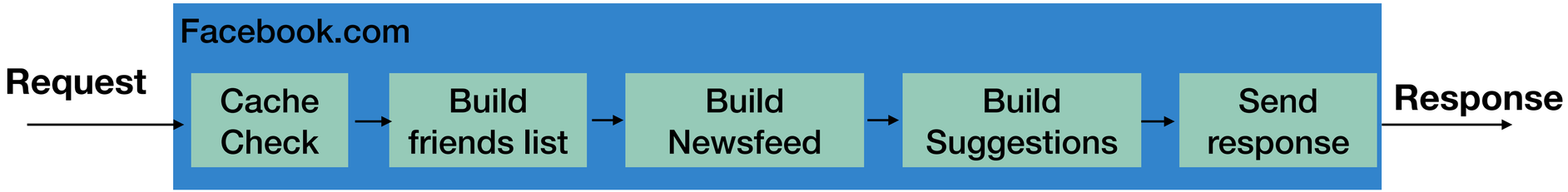
Round-trip time for data
8.5
milliseconds (est.)

Approach
Tradeworx is highly secretive, but the company is open about the price of a subscription: $250,000 a year.

Graphics: https://www.zerohedge.com/news/chicago-new-york-and-back-85-milliseconds

Software Architectures to Reduce Latency

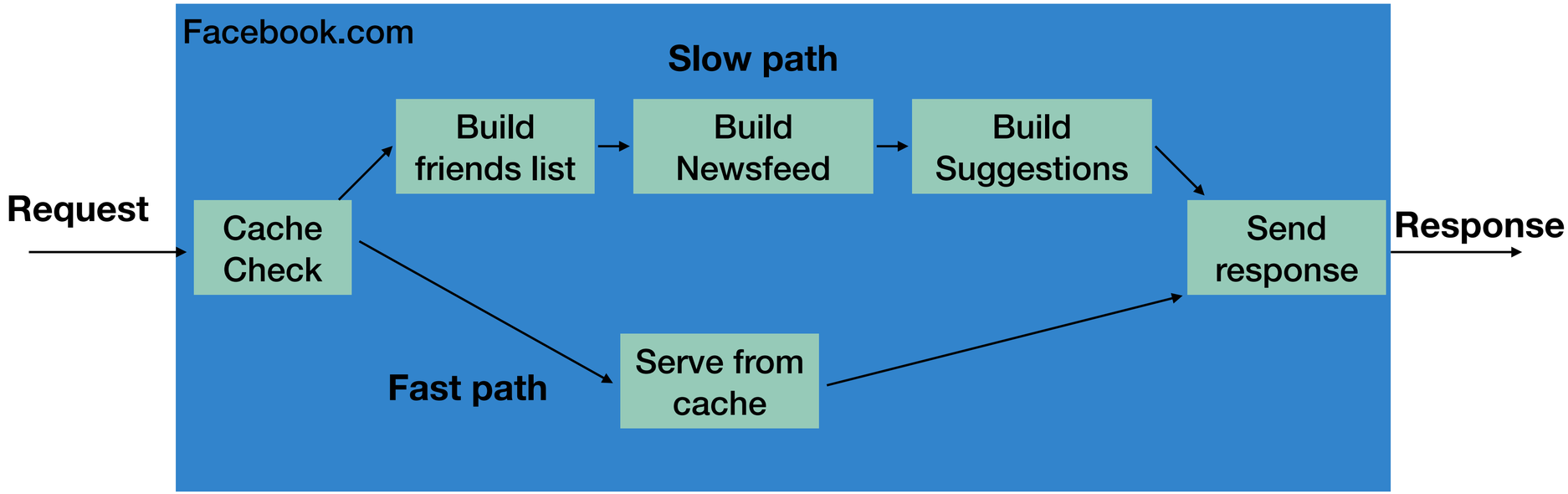
Pipeline architecture



Software Architectures to Reduce Latency

Pipeline architecture

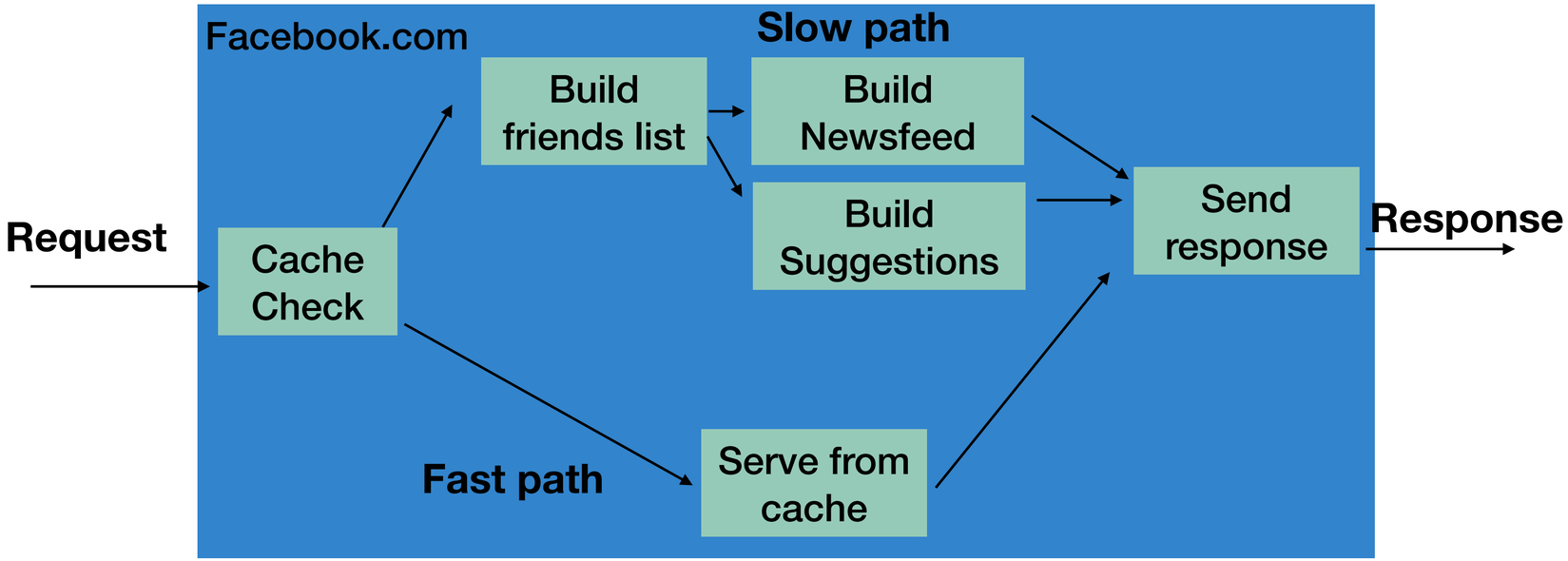
- Approach: **Optimize for the common case** (aka fast path and slow path)



Software Architectures to Reduce Latency

Pipeline architecture

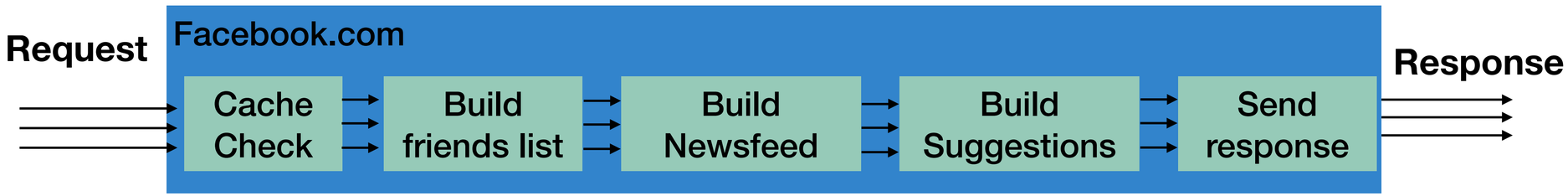
- Approach: use **concurrency**
- Limited by serial section



Software Architectures to Improve Throughput

Pipeline architecture

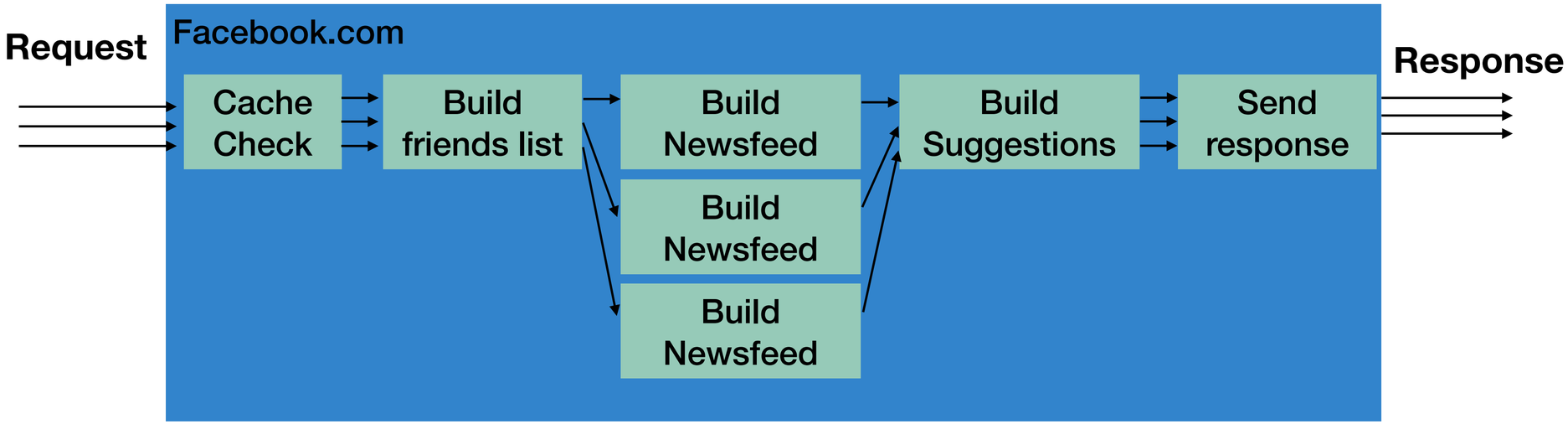
- Introduce concurrency into our pipeline
- Each stage runs in its own thread (or many threads, perhaps)
- If a stage completes its task, it can start processing the next request right away
  - E.g. our system will process multiple requests at the same time



Software Architectures to Improve Throughput

Pipeline architecture

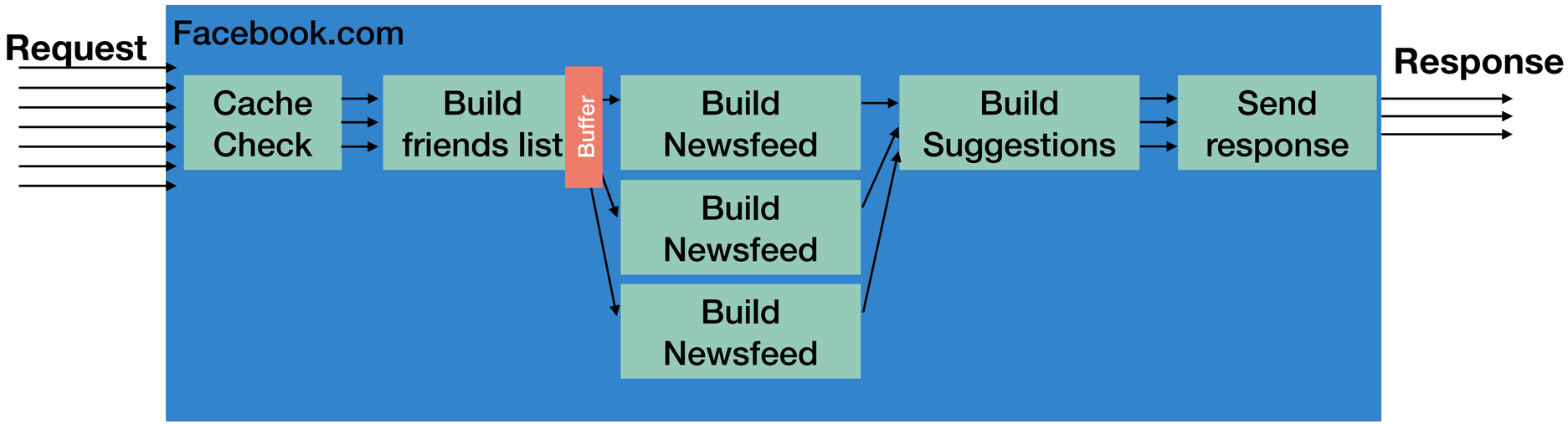
- Can also introduce concurrency to stages
- If one stage is a bottleneck, can we add more copies of it? (Hooray for micro services!)



Software Architectures to Improve Throughput

Solving Overload with Queueing and Buffering

- What happens when a slow component gets overloaded?
- Need to place a bounded buffer in between components!
  - When buffer becomes full, it prevents new requests from being accepted

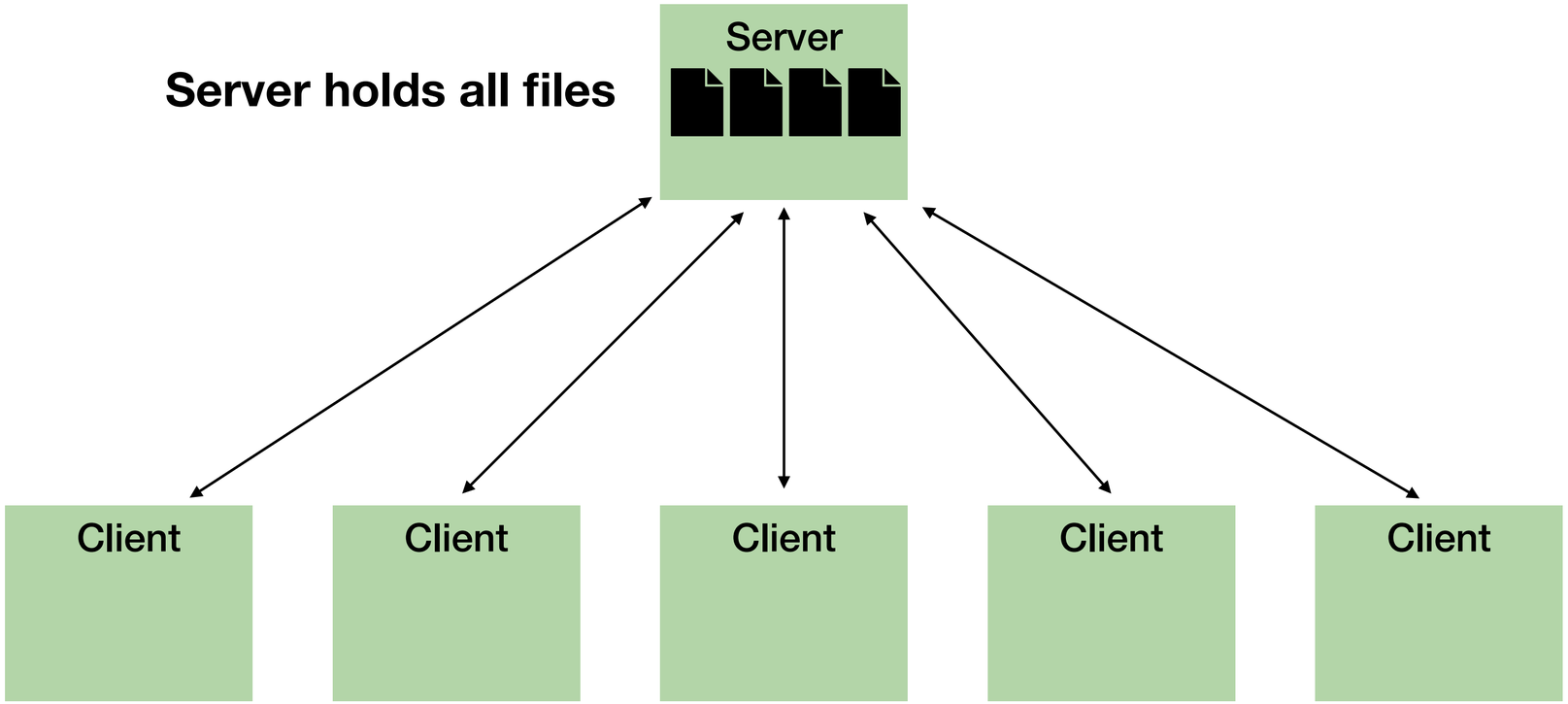


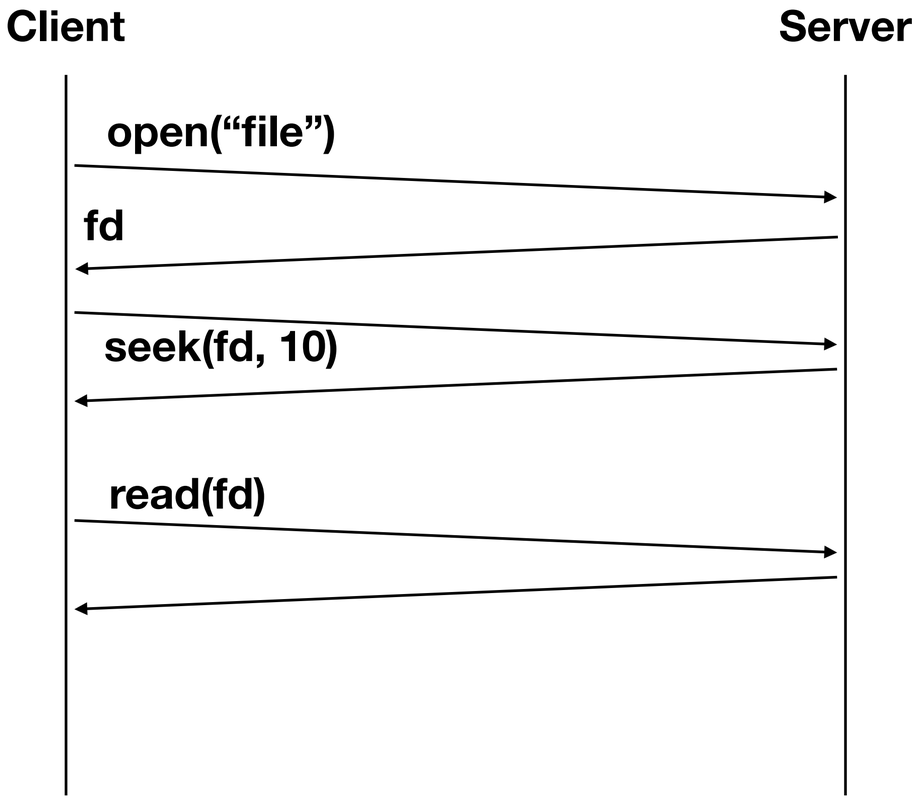
Architecture Case Study: Distributed File Systems

- Goals
  - Shared filesystem that will look the same as a local filesystem
  - Scale to many TB’s of data/many users
  - Fault tolerance
  - Performance

NFS: The Networked Filesystem

Architecture: Monolithic Server

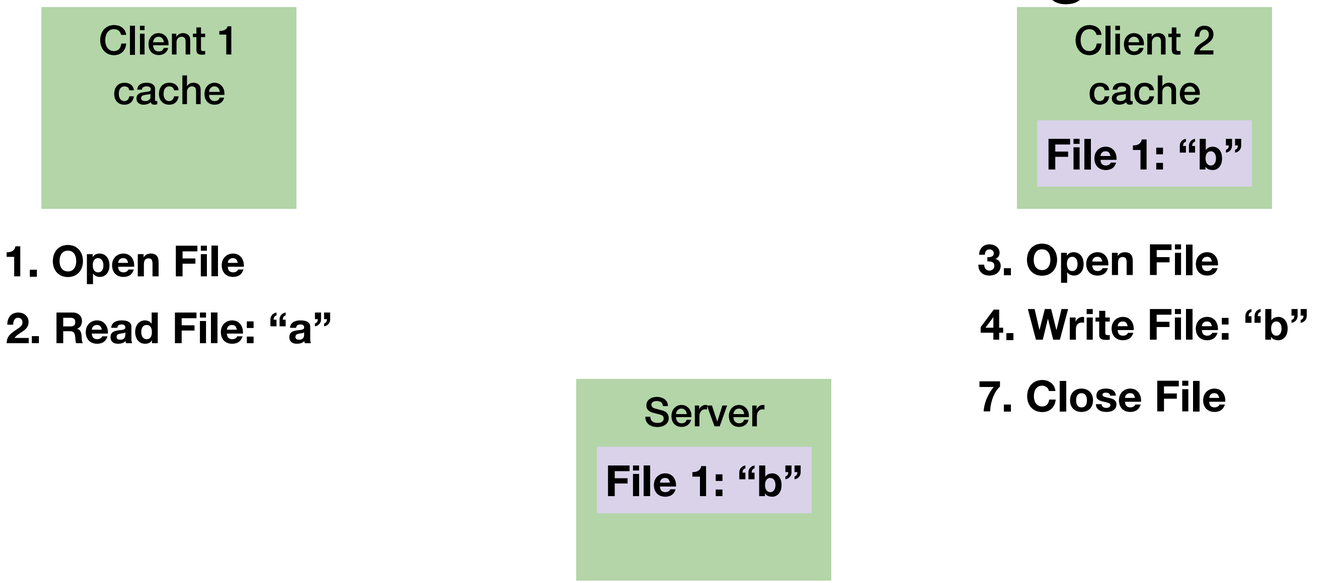




Clients send every read/write request to server

NFS: The Networked Filesystem

Not quite a monolithic architecture: caching



This is called “Open-to-Close” consistency. It is weird.

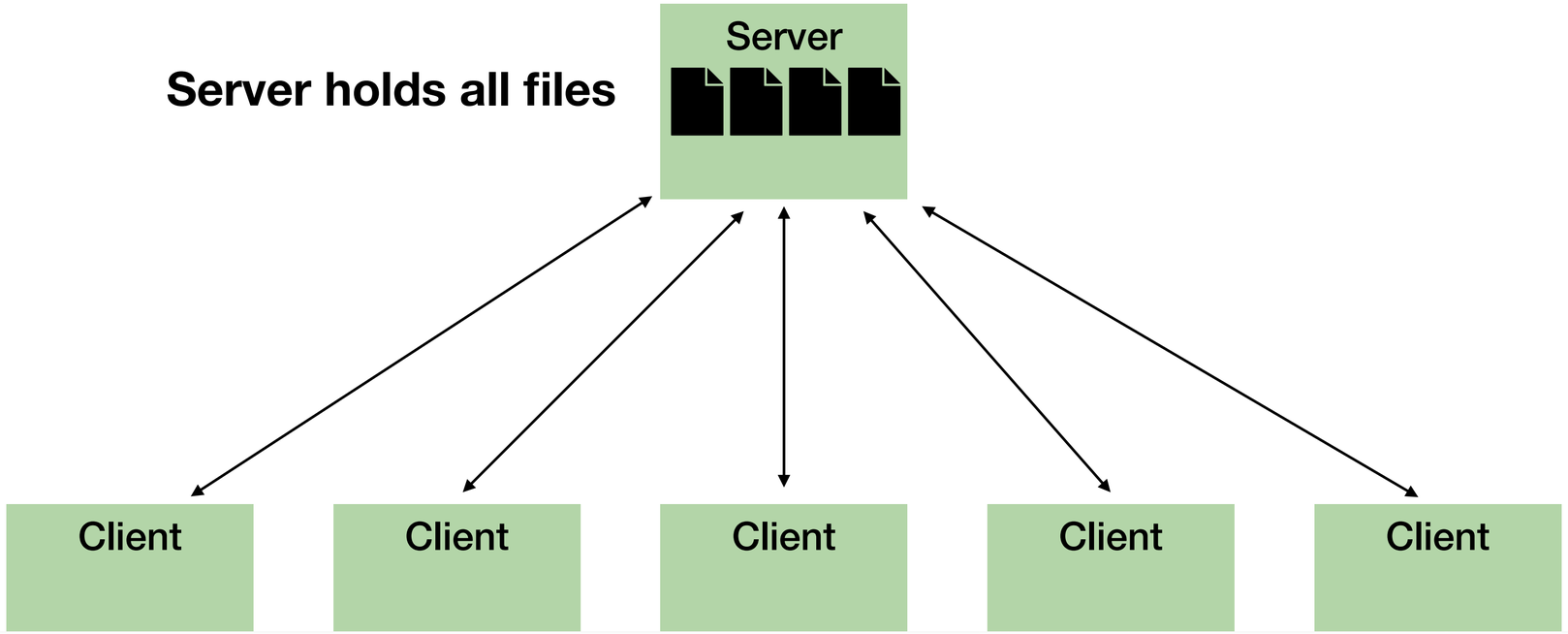
Sidebar: Cache Consistency

“The biggest problem in computer science”

- Strawman: Before you do a read, ask the server if the file has changed. Before you acknowledge a write, make server confirm the write
  - This works, but it’s a lot of network messages, and what happens when the network is slow or fails?
- This is a fundamental problem in distributed systems: when data is replicated, choose between “fast” or “safe” (a gross simplification of the CAP theorem)

NFS: The Networked Filesystem

What do we think of the monolithic architecture?

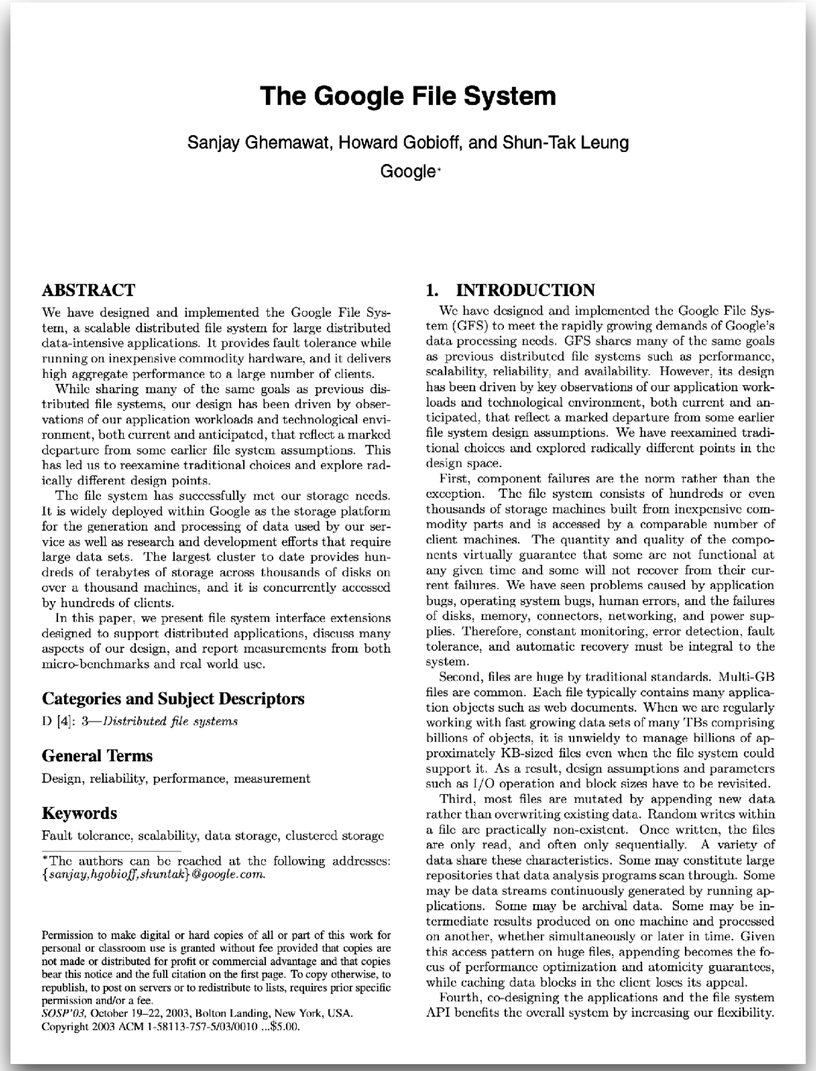


All files stored on the same server is bad because:
Fault tolerance (what if it crashes?)
Performance (what if we need to access 100's of GBs at a time?)
Scale (what if we need to store PBs of files?)
Plus, NFS' open-to-close caching can be weird

GFS: Google File System

Design Goals

"High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write."



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {*sanjay,hgobioff,shuntak*}*@google.com.*

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

GFS: Google File System

Big idea: Give up the old notion of a filesystem

- GFS does not look like a “regular” filesystem
- Google apps observed to have specific R/W patterns (usually read recent data, lots of data, append to end of file instead of overwriting middle)
- NFS tried to implement the same interface as a non-network filesystem. Why?

GFS: Google File System

Tiered Architecture: GFS Client, GFS Master, GFS Chunkservers

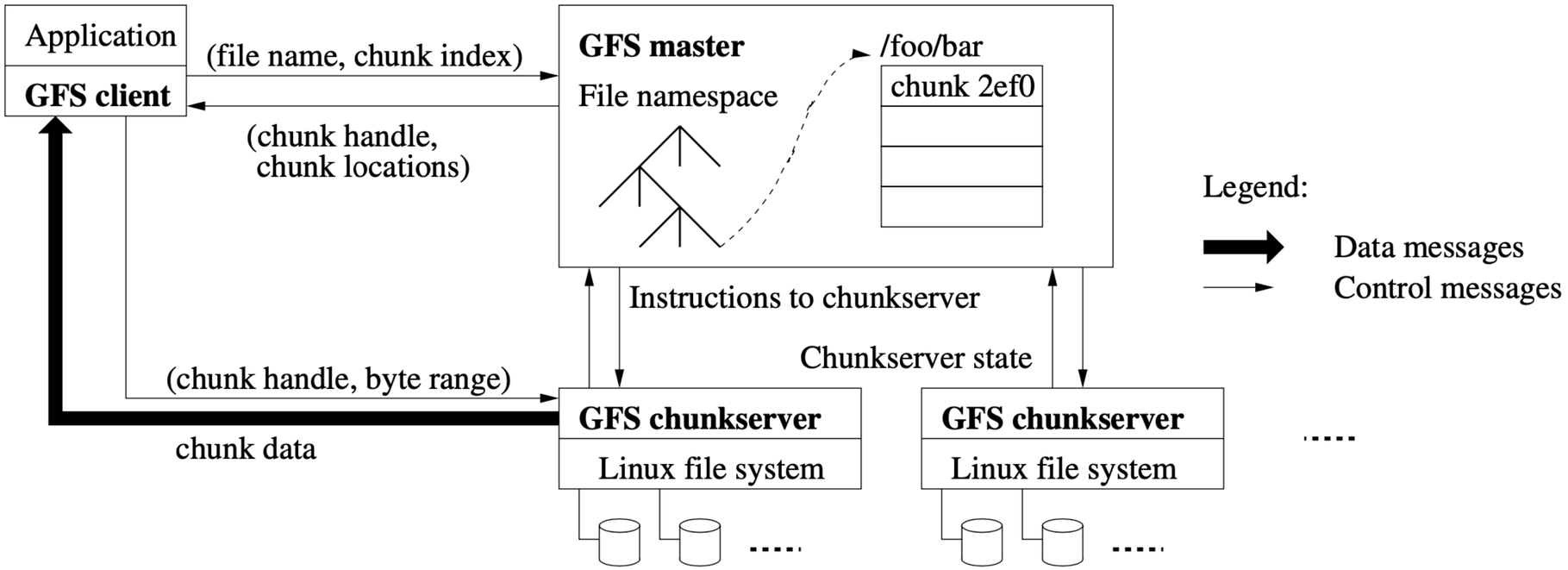


Figure: “The Google File System” by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, SOSP 2003

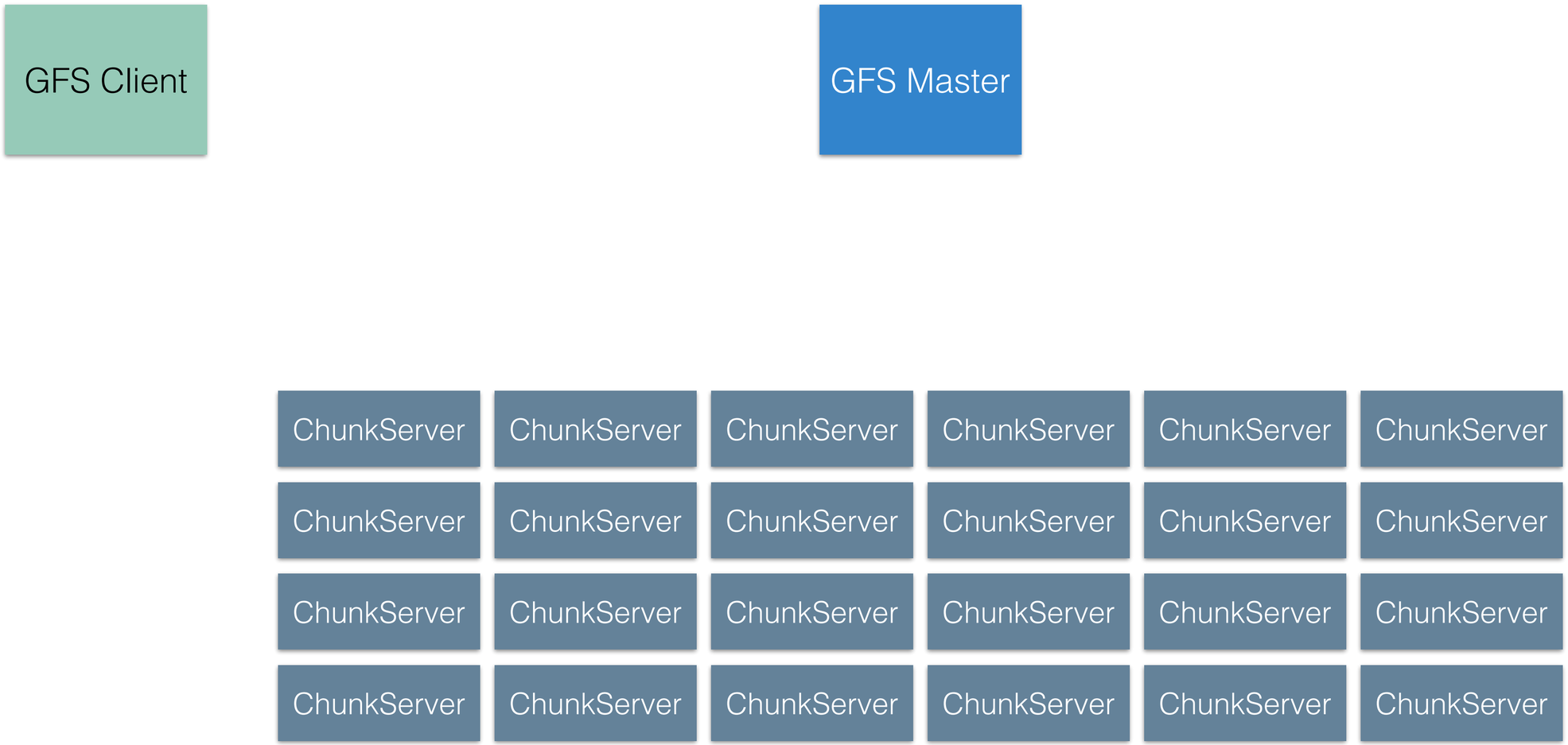
GFS: Google File System

Key architectural components

- Servers:
  - Single metadata server
  - Multiple data servers (“chunk servers”)
- Base unit is a “Chunk”
  - Fixed-part of a file (typically 64MB)
  - Read/write: specify chunk ID + byte range into file
  - Each chunk is replicated to at least 3 servers

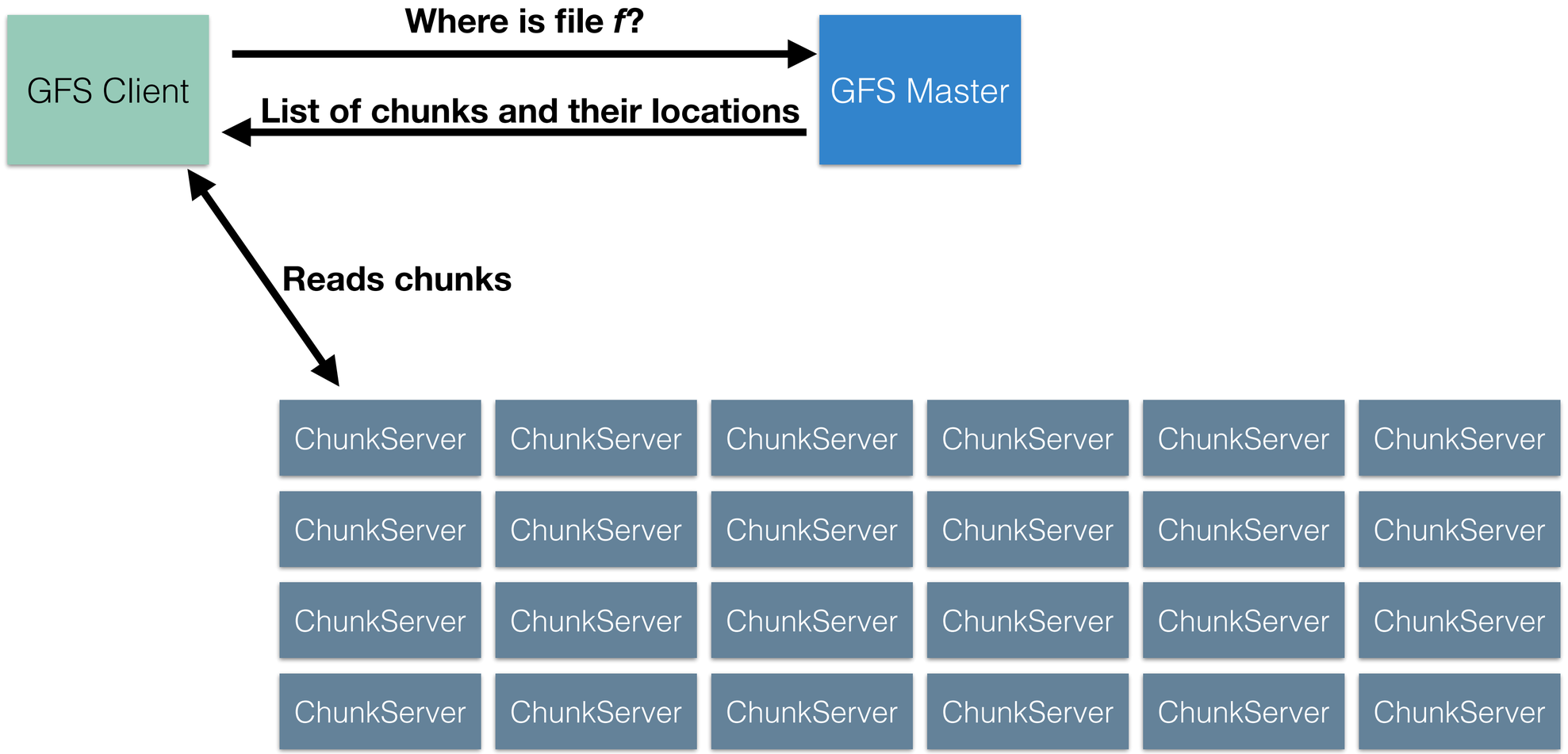
GFS: Google File System

Tiered Architecture: What are the tiers?



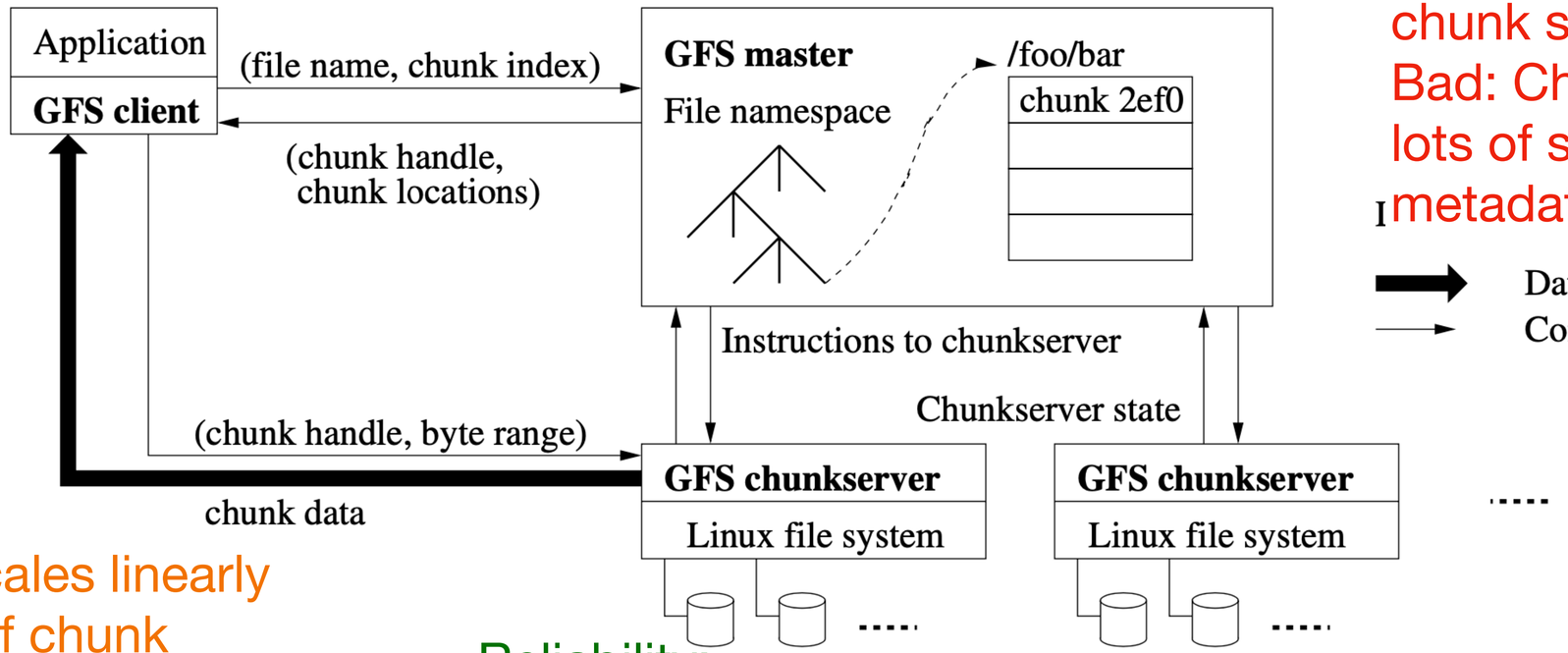
GFS: Google File System

Tiered Architecture: Reading Files



GFS: Google File System

Analyzing the architecture for quality attributes



Scalability:
Lots of big files? Just add more chunk servers
Bad: Chokes up if workload is lots of small files (too much metadata!)

Performance:
Throughput scales linearly with number of chunk servers

Reliability:
Chunks are replicated

Cost:
Chunk servers are cheap machines with cheap disks

Figure: “The Google File System” by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, SOSP 2003

Comparing GFS + NFS

NFS: Monolithic Server, GFS: Tiered Metadata + Chunk Servers

- Fault tolerance (what if it crashes?)
  - NFS: Crashing the server is bad
  - GFS: Crashing a chunk server is fine, crashing the primary is bad
- Performance (what if we need to access 100's of GBs at a time?)
  - NFS: Limited by single server's bandwidth
  - GFS: Limited only by number of chunk servers (can get good parallelism between 100 chunk servers each at 1GB/sec)
- Scale (what if we need to store PBs of files?)
  - NFS: Limited by storage of single server
  - GFS: Limited by amount of metadata that can be stored on single master
- Plus, NFS' open-to-close caching can be weird
  - GFS: No caching

Comparing GFS + NFS

NFS: Monolithic Server, GFS: Tiered Metadata + Chunk Servers

- NFS: All requests handled by a single server
- GFS: All *metadata* requests handled by a single metadata server, all reads/writes handled by chunk servers
- Which is better?
  - Performance?
  - Reliability/Safety?
  - Portability?
  - Supportability?
  - Scalability?
  - Cost?

Now that you've seen it once

Hadoop (Open source Map/Reduce implementation)

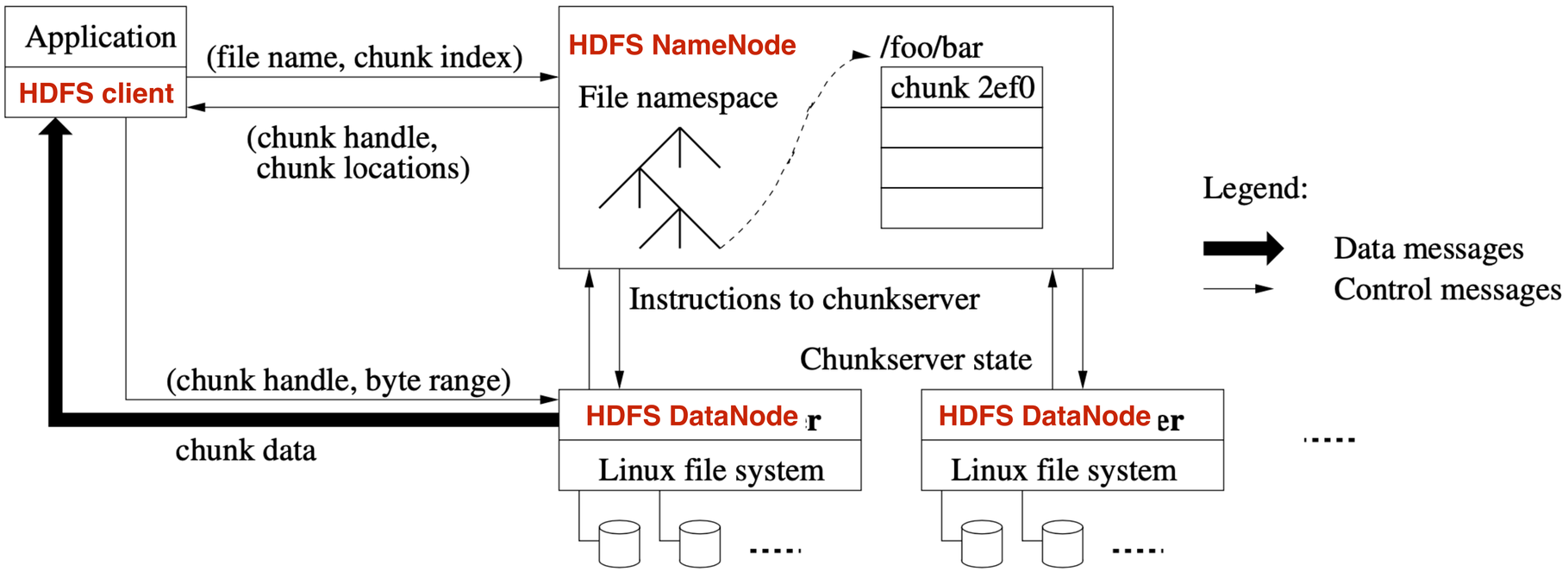


Figure: "The Google File System" by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, SOSP 2003

Discussion/Activity: Performance Bottlenecks

- What is a piece of software that you've written or worked with that faced a performance challenge, and how did you and/or your team overcome it?

This work is licensed under a Creative Commons Attribution-ShareAlike license