CS 4530: Fundamentals of Software Engineering

Module 10.2: Case Studies

Jonathan Bell, Adeel Bhutta, Mitch Wand

Khoury College of Computer Sciences

Learning Goals for this Lesson

- By the end of this lesson you should be able to:
  - Briefly describe several typical examples of distributed systems
  - Briefly describe how each of them deals with scalability, fault tolerance, etc.

Case Study 1: the Network File System NFS

- NFS is a distributed file system: multiple clients can read/write the same files
- Created in 1984, still widely used
- In a UNIX (POSIX-compliant) operating system, files are stored in a tree from "/"
- Access a remote file by name like /username@remotehost/path/to/remote/file
- Or you could "mount" a remote filesystem to access it locally

NFS is a Monolithic Shared Filesystem

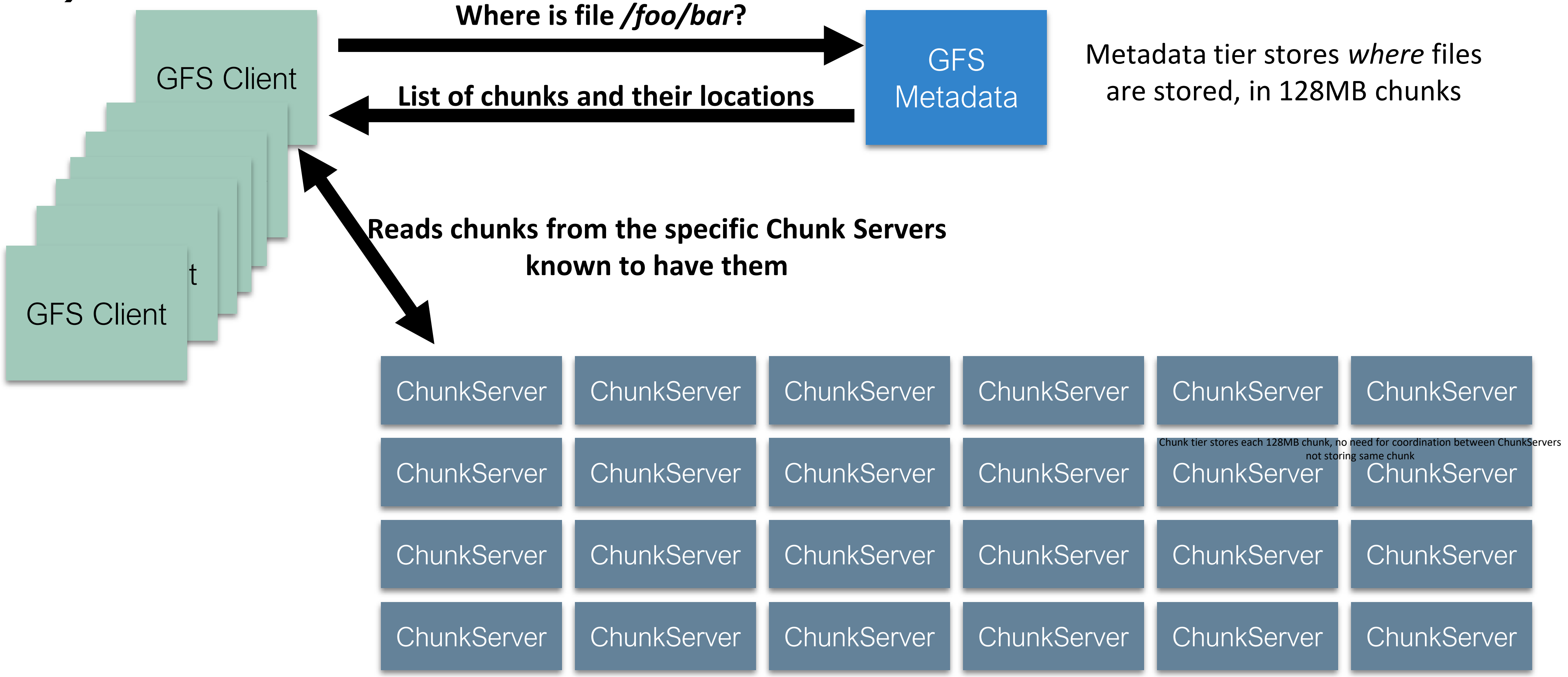
- All files are stored on a single server
- To list files in a directory, clients make request to server
- To read or write files, clients make request to server
- Clients might “lock” files to prevent concurrent updates
- Assuming that scale, throughput, fault tolerance requirements are relatively low, this is an acceptable architecture
- This architecture is the *easiest* to build fast and correctly

Case Study 2: GFS (Google File System, ~2010)

- Stated requirements:
- “**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.**”

GFS is a tiered filesystem with two tiers: Metadata and File Chunks

- Example: GFS (Google File System, c 2010)


GFS Client
GFS Client
Where is file */foo/bar*?
List of chunks and their locations
GFS Metadata
Reads chunks from the specific Chunk Servers known to have them
ChunkServer

Metadata tier stores *where* files are stored, in 128MB chunks

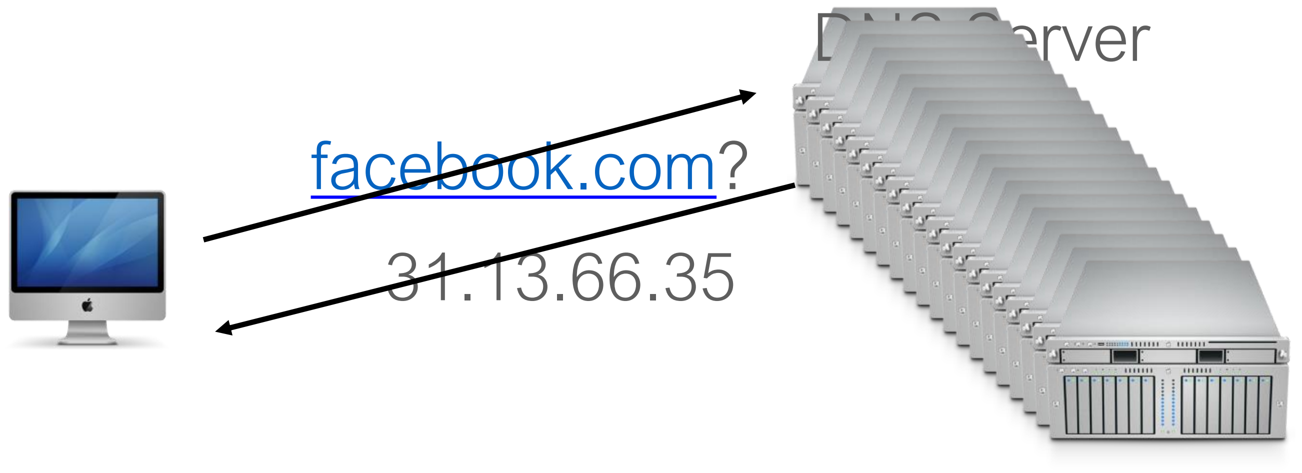
Chunk tier stores each 128MB chunk, no need for coordination between ChunkServers not storing same chunk

Case Study 3: Domain Name System (DNS)

- Nodes (hosts) on a network are identified by IP addresses
- E.g.: 142.251.41.4
- We humans prefer something easier to remember: calendar.google.com, facebook.com, www.khoury.northeastern.edu
- We need to keep a directory of domain names and their addresses
- We also need to make sure everybody gets directed to the correct host

Requirements for the DNS system

- Need to handle millions of DNS queries per second
- Not immediately obvious how to scale: how do we maintain replication, some measure of consistency?

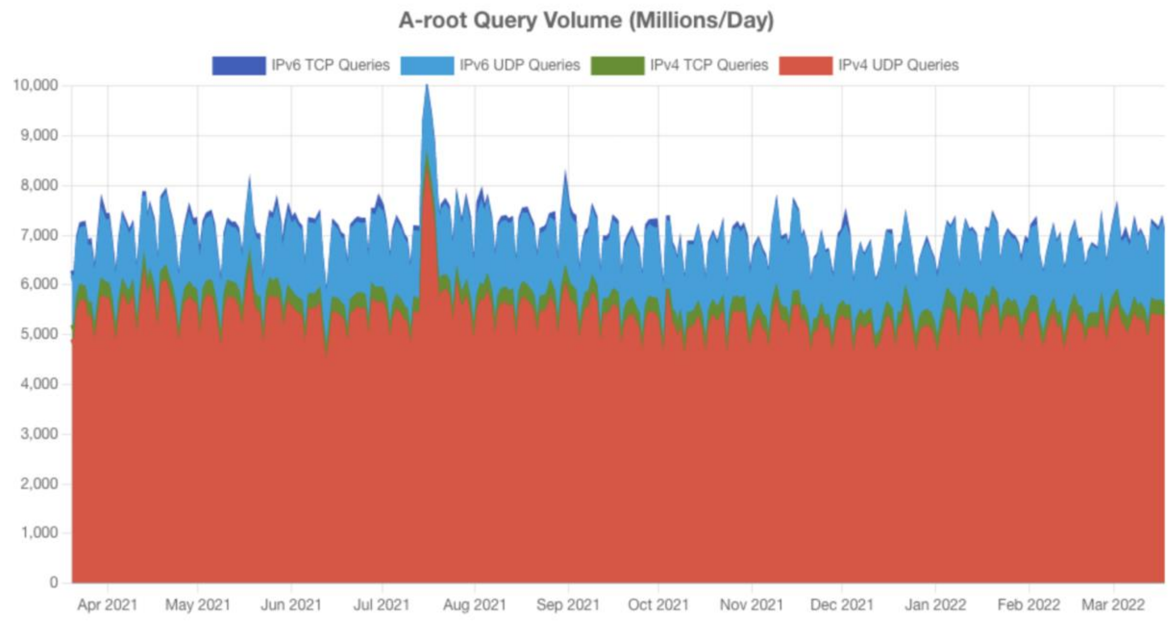


DNS distributed system goals

- We need a **scalable** solution
  - New hosts keep being added
  - Number of users increases
  - Need to maintain speed/responsiveness
- We need our service to be **available** and **fault tolerant**
  - It is a crucial basic service
  - A problematic node shouldn’t “crash the internet”
  - Reads are more important that writes: far more queries to resolve records than to update them
- Global in scope
  - Domain names mean the same thing everywhere

Strawman solution A: monolithic architecture

- Route all requests to a server with a well-known address.
- All requests made to this server:
  - Single point of failure
  - Bottleneck for throughput and access time (billions of queries per day; access time in msecs)
  - Bottleneck for administration (adding/changing records?)
  - Ultimately, **not scalable**!



https://a.root-servers.org/metrics

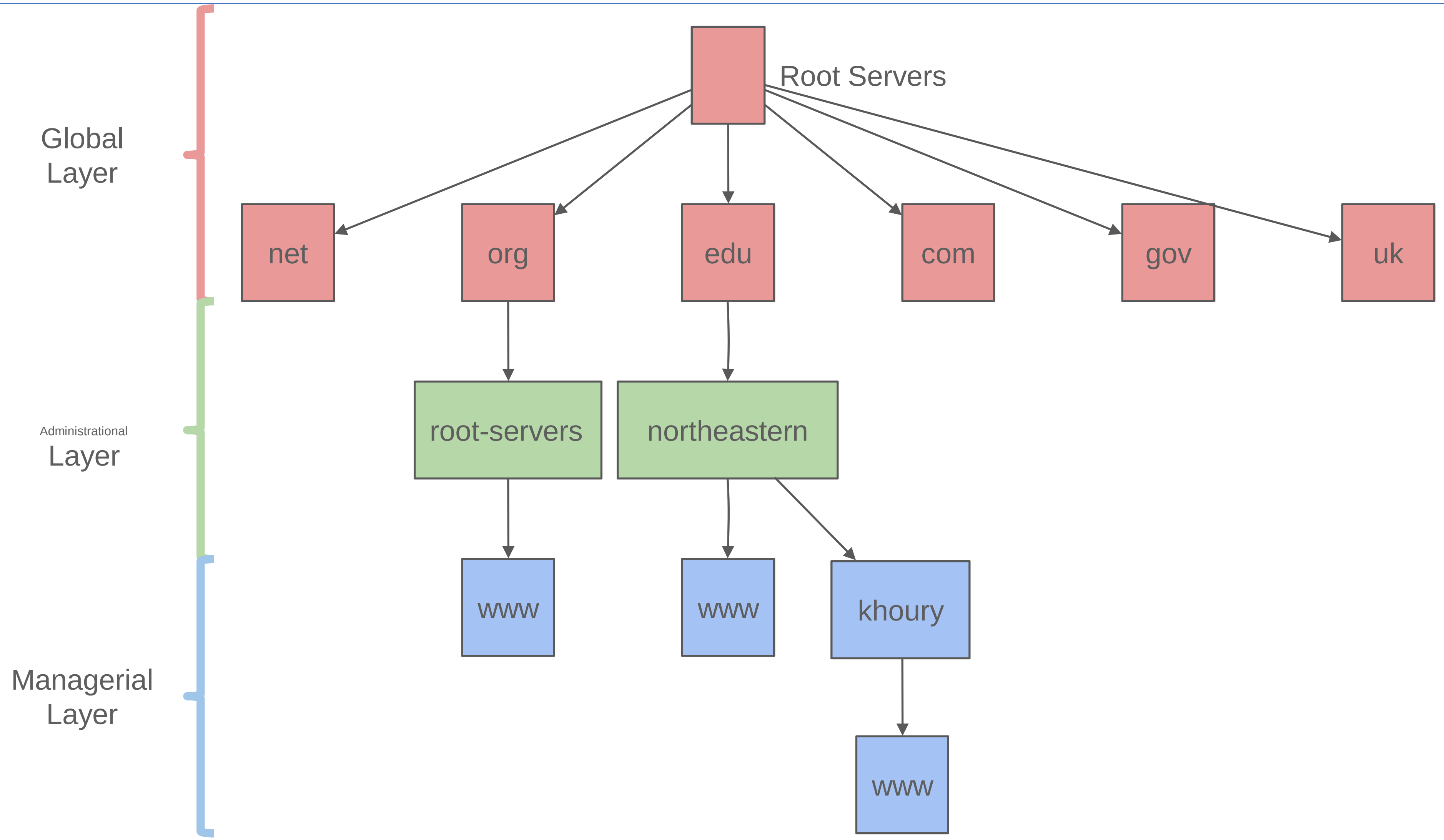
Strawman solution B: Use a local file

- Keep local copy of mapping from all hosts to all IPs (e.g., /etc/hosts)
- Space requirements are feasible now
  - IPv4 space is now full
  - 32-bits: 4,294,967,296 addresses
  - At 1 byte per address, file would be 4GB
  - Not a lot of disk space now, but DNS was introduced in the late 80s.
- **BUT** hosts change IPs regularly, so need to download file frequently
- Lot of constant internet bandwidth use
- Not scalable!

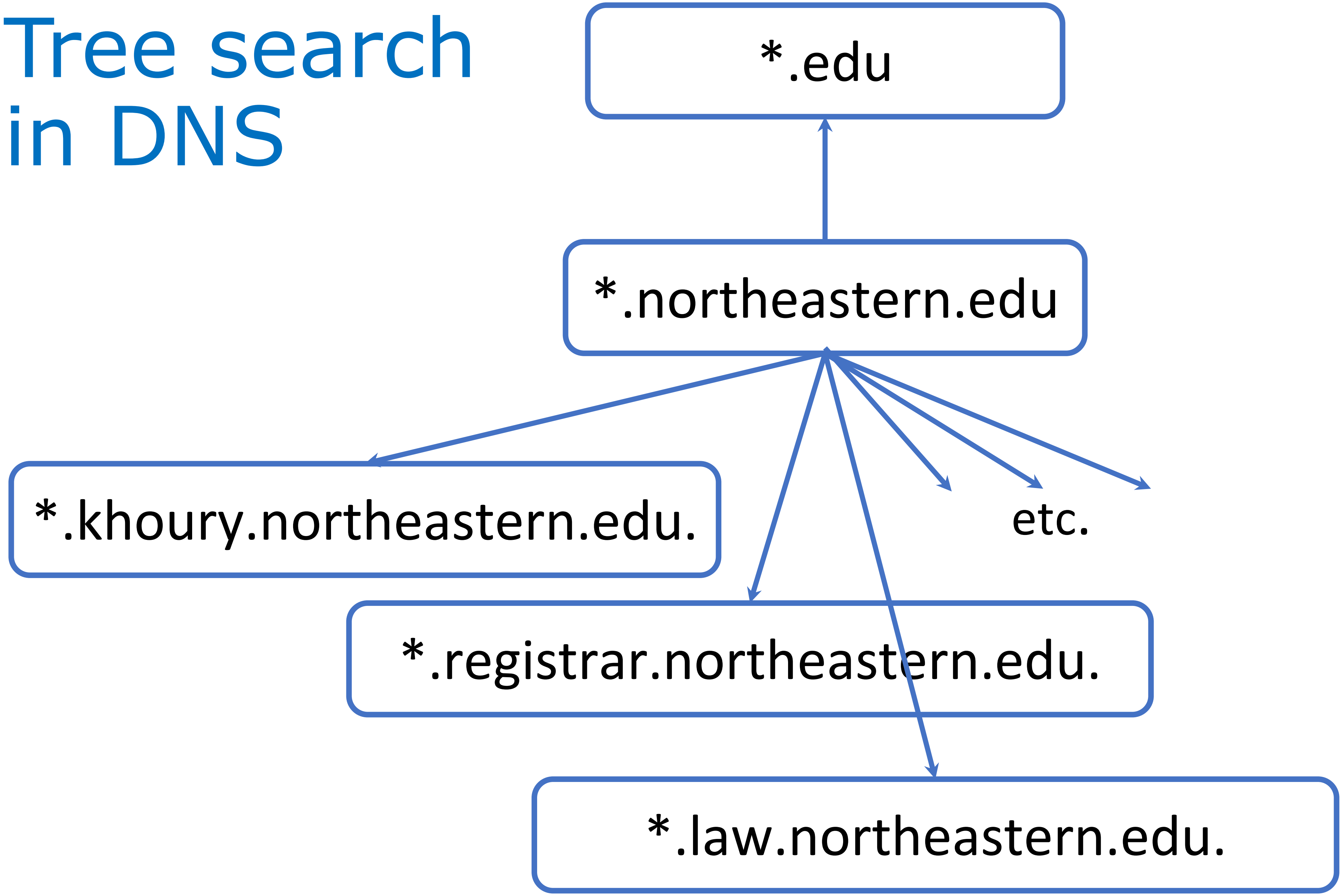
A tiered architecture yields a scalable solution

- Idea: break apart responsibility for each part of a domain name (**zone**) to a different group of servers
- Each zone is a continuous section of the name space, eg *.northeastern.edu
- Each zone has an associated set of name servers.

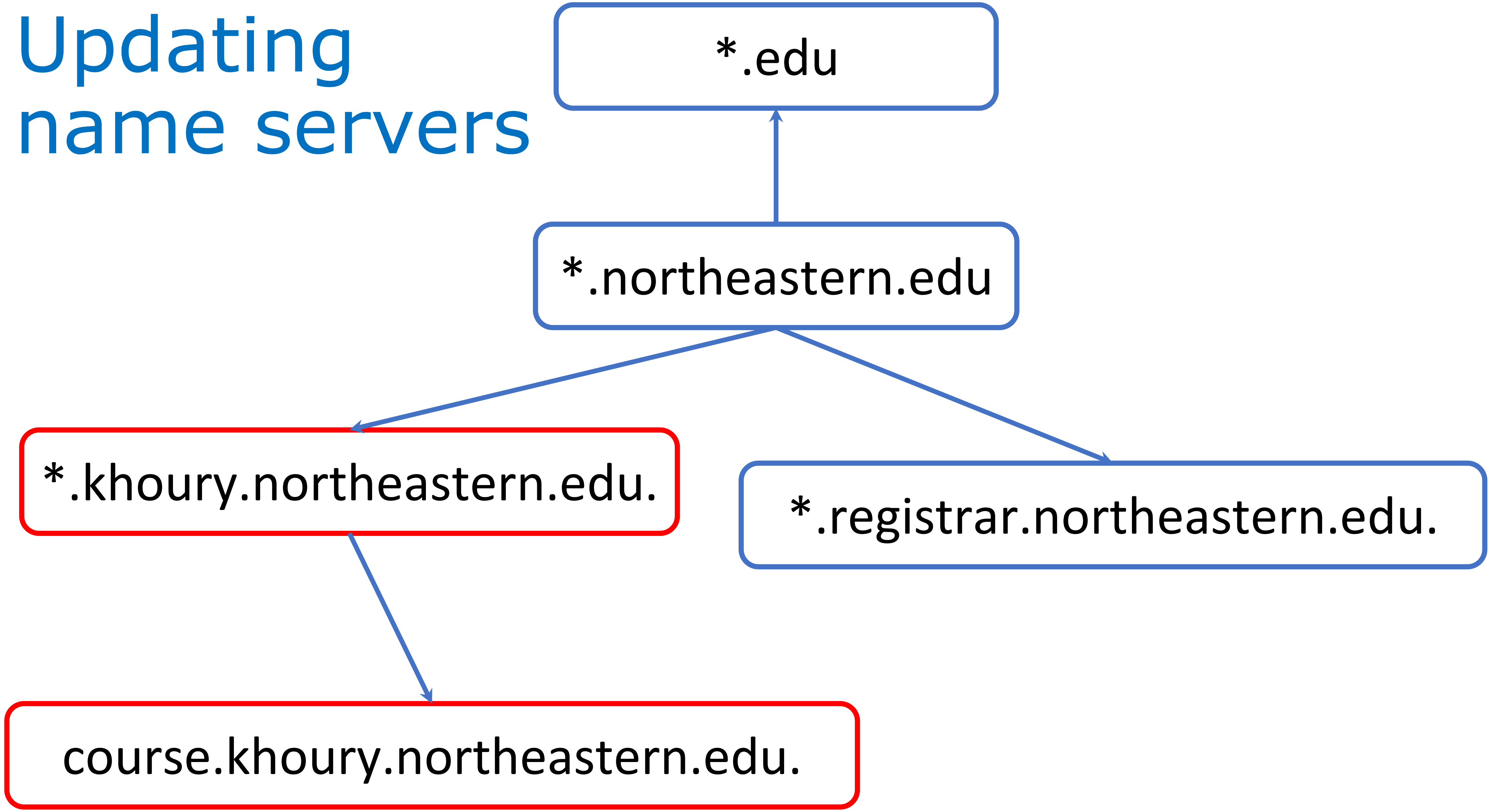
DNS partitions responsibility by “layers”.



Tree search in DNS



Updating name servers


*.edu
*.northeastern.edu
*.khoury.northeastern.edu.
*.registrar.northeastern.edu.
course.khoury.northeastern.edu.

This is an example of a tiered architecture

- Each server need only needs to know about its immediate descendants in its zone.
- It only processes requests about a single zone.
- Both data and processing are limited to requests about this zone– any other requests are delegated to this server’s parent server.

But some zones are too big and too busy to be handled by a single server

- Eg, .edu, .com, .gov, etc.
- So these servers are **replicated**.

There is replication even within the root servers

- 13 root servers
  - `[a-m].root-servers.org`
  - E.g., d.root-servers.org
- But each root server has multiple copies of the database, which need to be kept in sync.
- Somewhere around 1500 replicas in total.

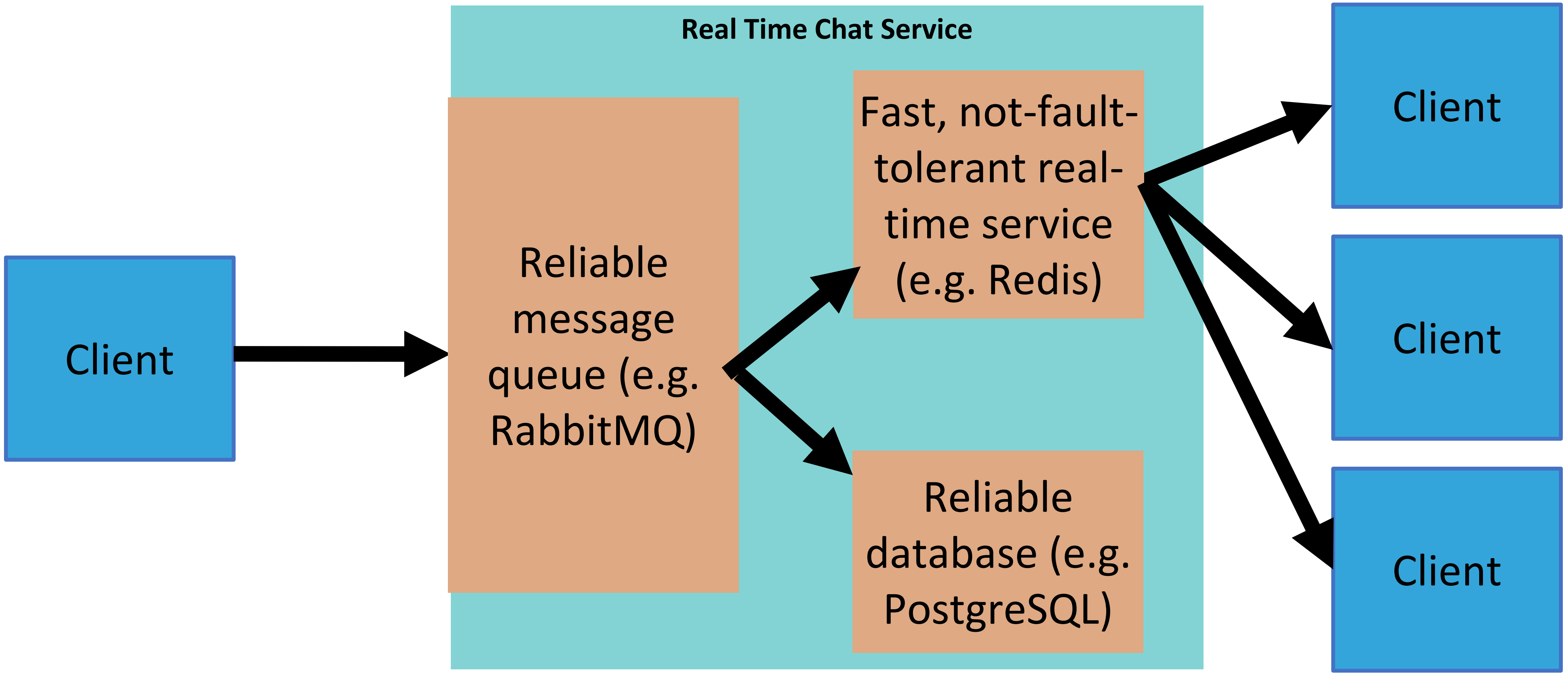
Case Study 4: Reliable Real-Time Chat

- Requirements: “Must support real-time text chat for 2,000 users exchanging messages. Must have **best-effort delivery in real-time**, and **guarantee that all messages acknowledged are preserved**.”
- Challenge: Real-time “best-effort” delivery has conflicting requirements (low latency at expense of fault tolerance) with guaranteeing all messages are eventually delivered (fault tolerance at expense of latency)

A reliable real-time chat could use separate processing units for each requirement.

- Requirements: “Must support real-time text chat for 2,000 users exchanging messages. Must have **best-effort delivery in real-time**, and **guarantee that all messages acknowledged are preserved**.”
- Allocate separate processing units for these requirements:
  - “Real time” component optimizes for speed and availability (sacrificing fault-tolerance)
  - “Persistence” component optimizes for fault-tolerance, sacrificing speed and availability
- Event queue service receives events, dispatches to both processing units and is fault tolerant

Block diagram for a real-time chat service


Real Time Chat Service
Client
Reliable message queue (e.g. RabbitMQ)
Fast, not-fault-tolerant real-time service (e.g. Redis)
Reliable database (e.g. PostgreSQL)
Client
Client
Client

Learning Goals for this Lesson

- By the end of this lesson you should be able to:
  - Briefly describe several typical examples of distributed systems
  - Briefly describe how each of them deals with scalability, fault tolerance, etc.