CS 4530 Software Engineering

Module 10: Distributed Systems Architectures

Jonathan Bell, Adeel Bhutta, Mitch Wand
Khoury College of Computer Sciences

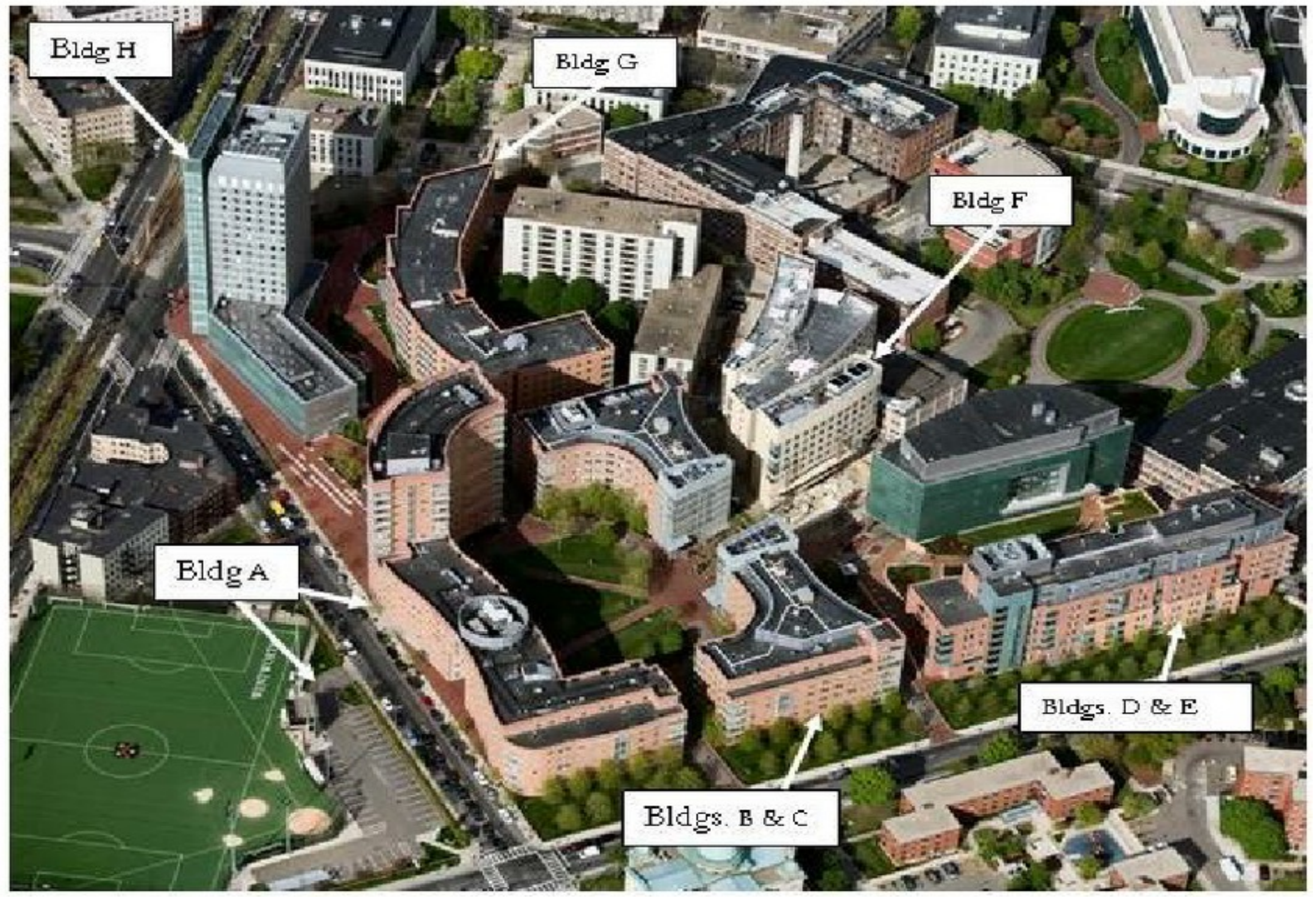
Learning Objectives for this Lesson

By the end of this lesson, you should be able to…

- Recognize common software architectures
- Understand tradeoffs of scalability, performance, and fault tolerance between these architectures
- Describe what makes web services RESTful, and implement a REST API

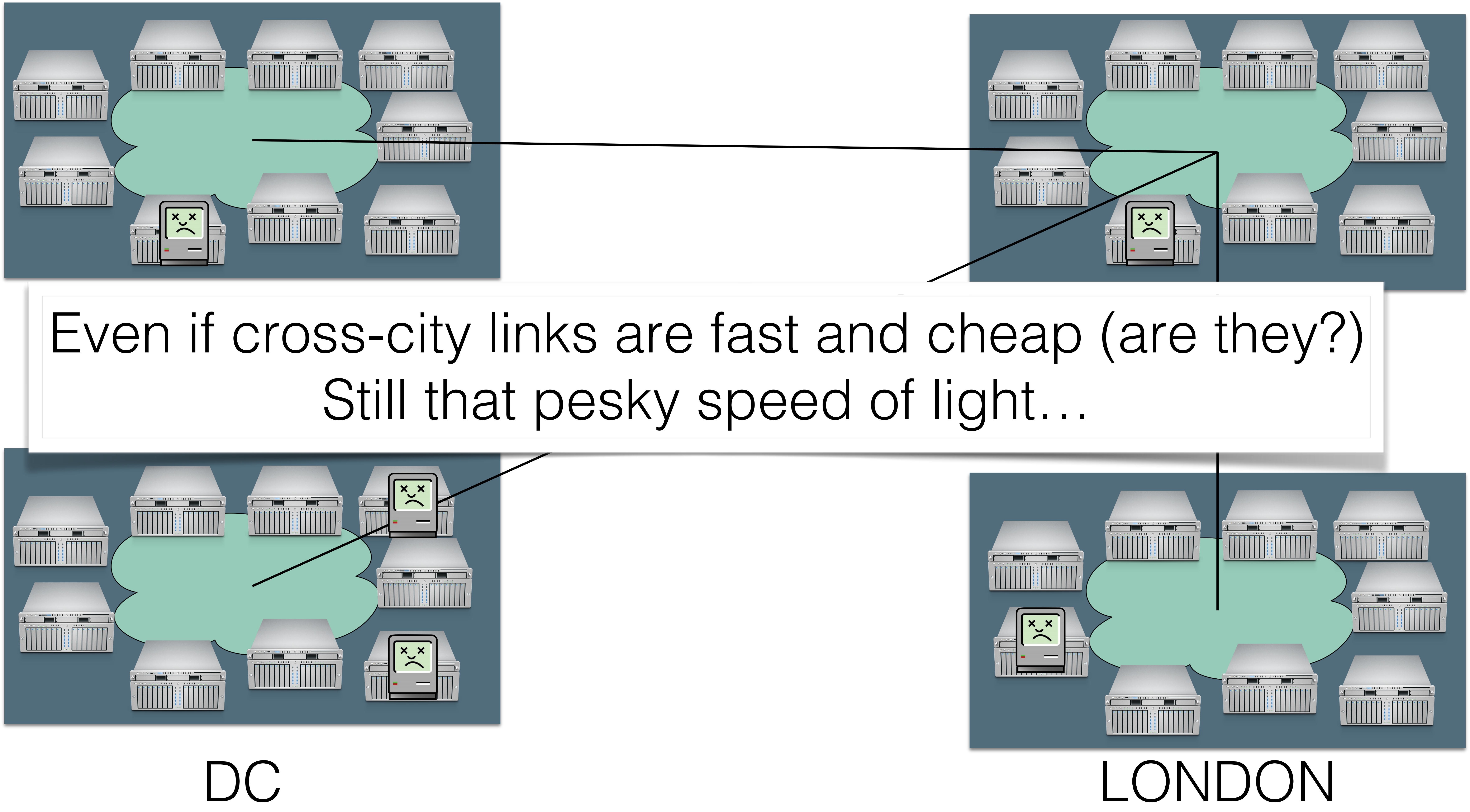
Distributed Software Architectures

- Goal: abstract details away into reusable components
- Enables exploration of design alternatives
- Allows for analysis of high-level design before implementation
- Match system requirements to quality attributes of common architectural patterns



Review: Distributed Systems Must Compromise

Constraints: number of nodes, network links



Replicated Systems Must Compromise

Consistency or availability?

Consistent:

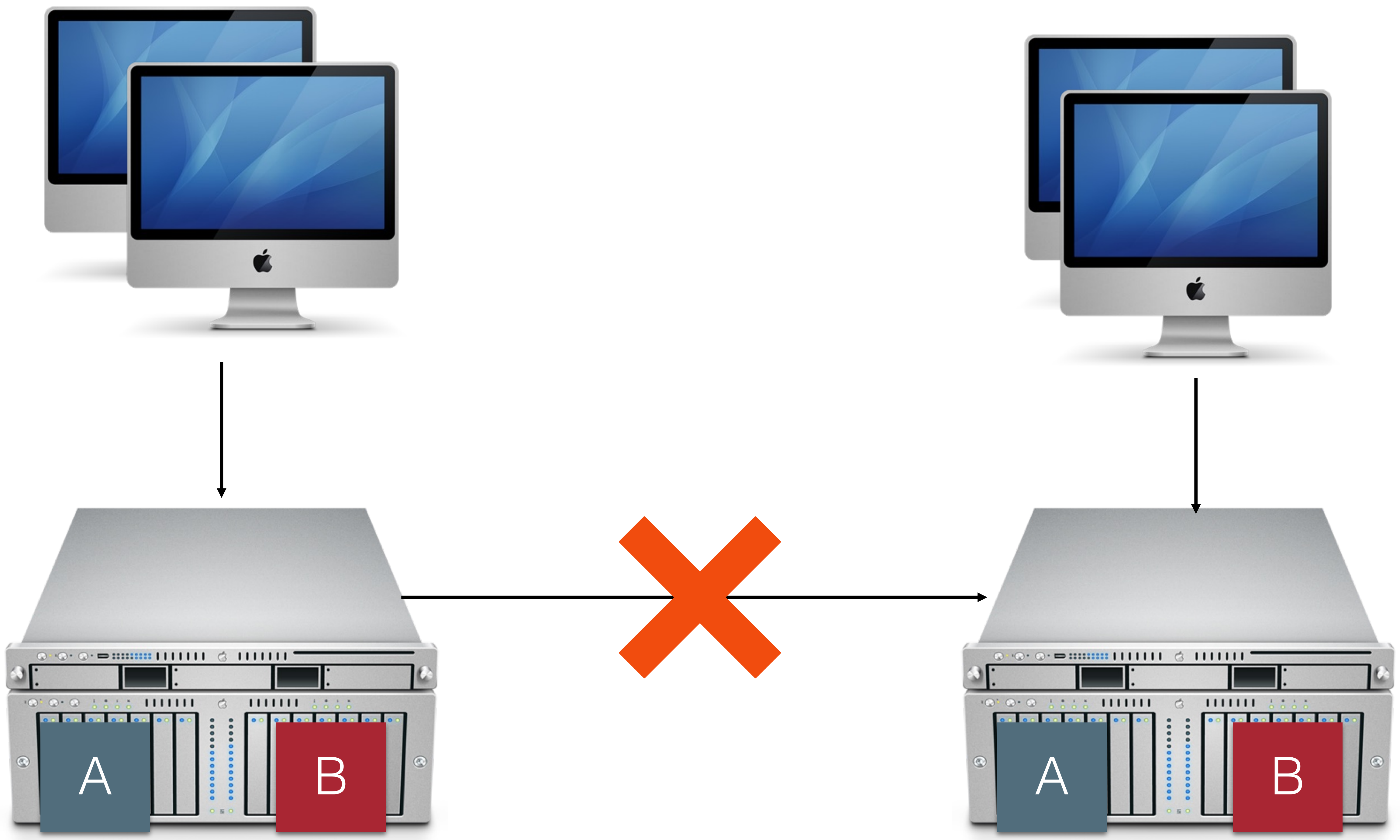
Maintain that "single server" behavior - all clients see the same values *regardless* of failures

At least one server can't safely respond in case of failure

Available:

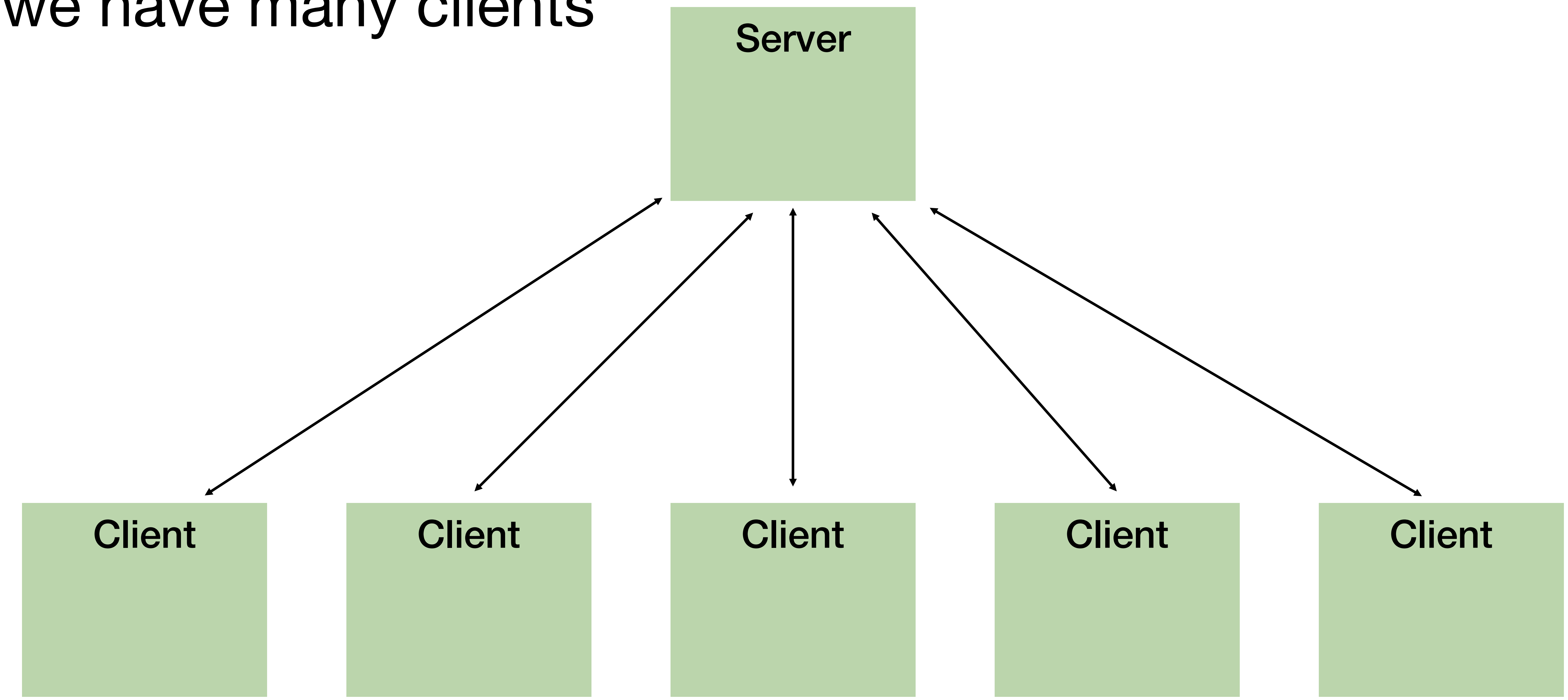
Different servers might diverge

Ignores network failures, as long as client can reach server, still offer a response



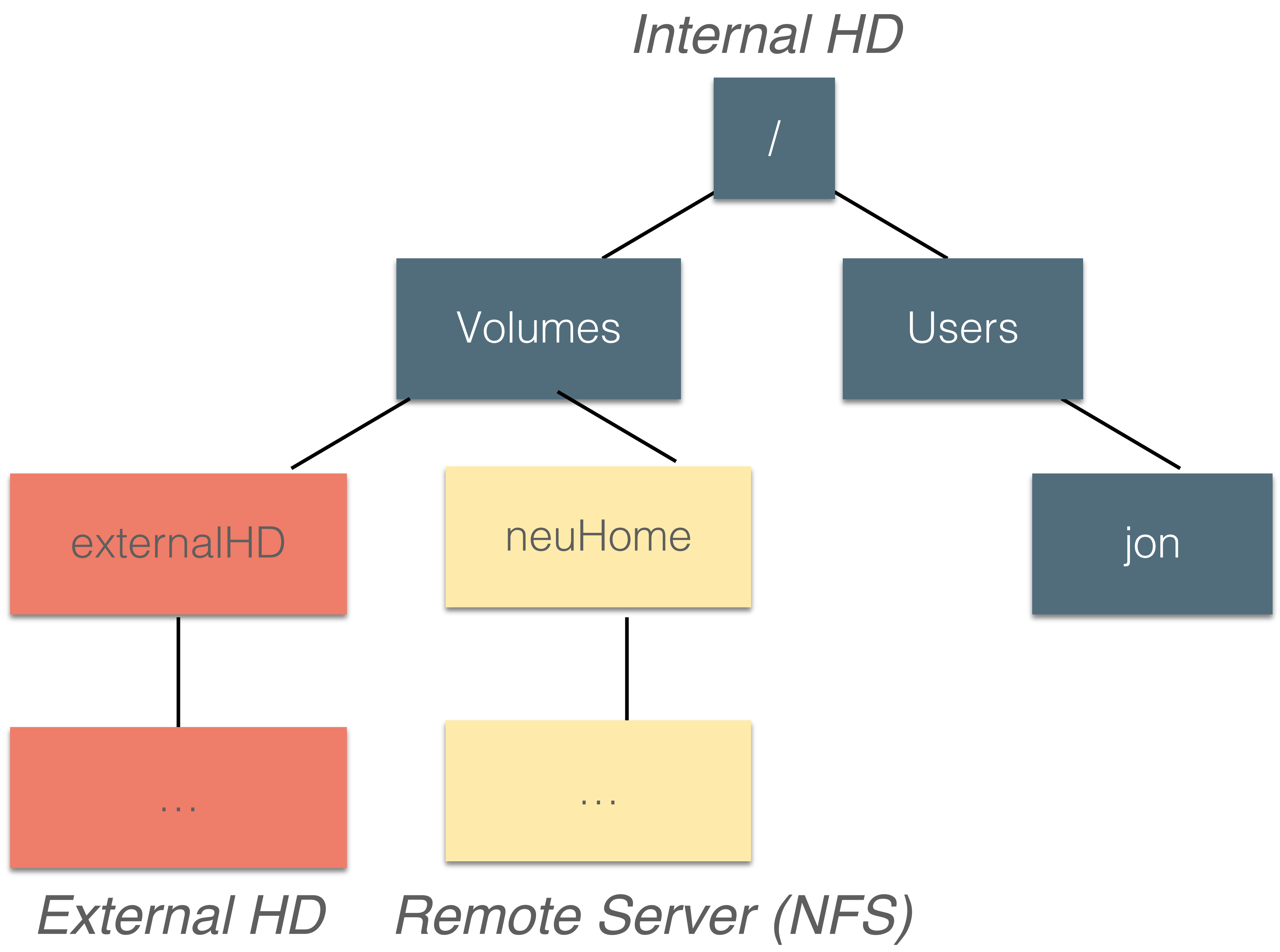
The Monolith Architecture Relies on a Single Server

- Simplest answer to consistency problem: have only one server, one source of truth
- Still “distributed” in that we have many clients
- Sacrifices:
  - Scalability
  - Performance
  - Fault tolerance



NFS is the Network File System

- In a UNIX (POSIX-compliant) operating system, files are stored in a tree from “/”
- “Mount” multiple filesystems to access them locally
- Filesystems could be directly attached to this computer, or shared by a remote server
- NFS is a distributed file system: multiple clients can read/write the same files
- Created in 1984, still widely used



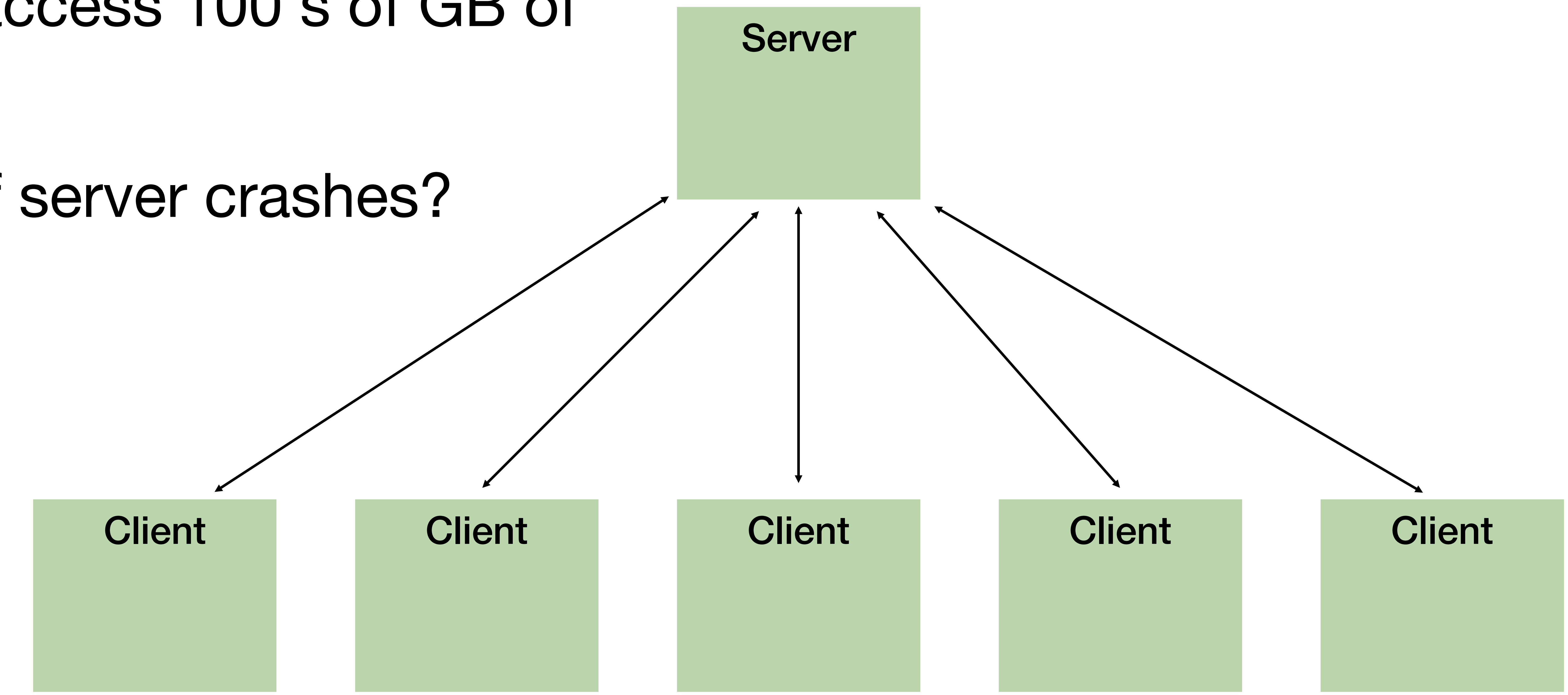
NFS is a Monolithic Shared Filesystem

- 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

Monolithic Architectures Struggle to Scale

Challenges with NFS

- Scalability - How to go from 10 to 100 to 1,000 clients?
- Performance - How to access 100's of GB of data concurrently?
- Fault tolerance - What if server crashes?

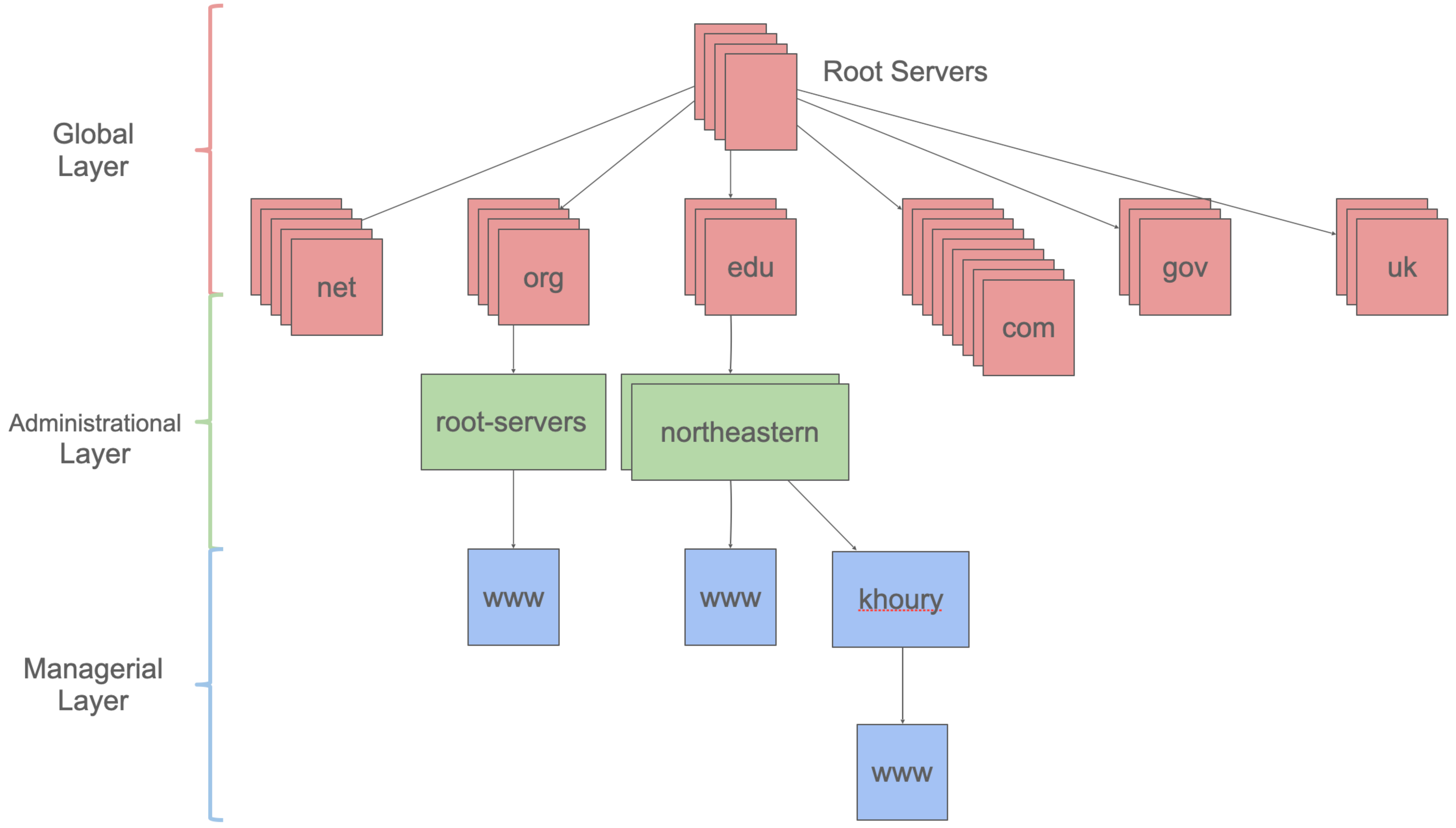


Replication Alone is Not The Answer

- Constraints:
  - Latency: Speed of light (~1ns/ft)
  - Throughput: Long-distance links between servers are relatively low throughput (10’s of Gbps, compare to 100’s of Gbps within a single server)
- Tradeoffs for replication, particularly over long distances:
  - Replication will *add* latency, not reduce it
  - Usually not enough bandwidth to maintain replication of all data across all nodes

Tiered Architectures Partition Responsibility

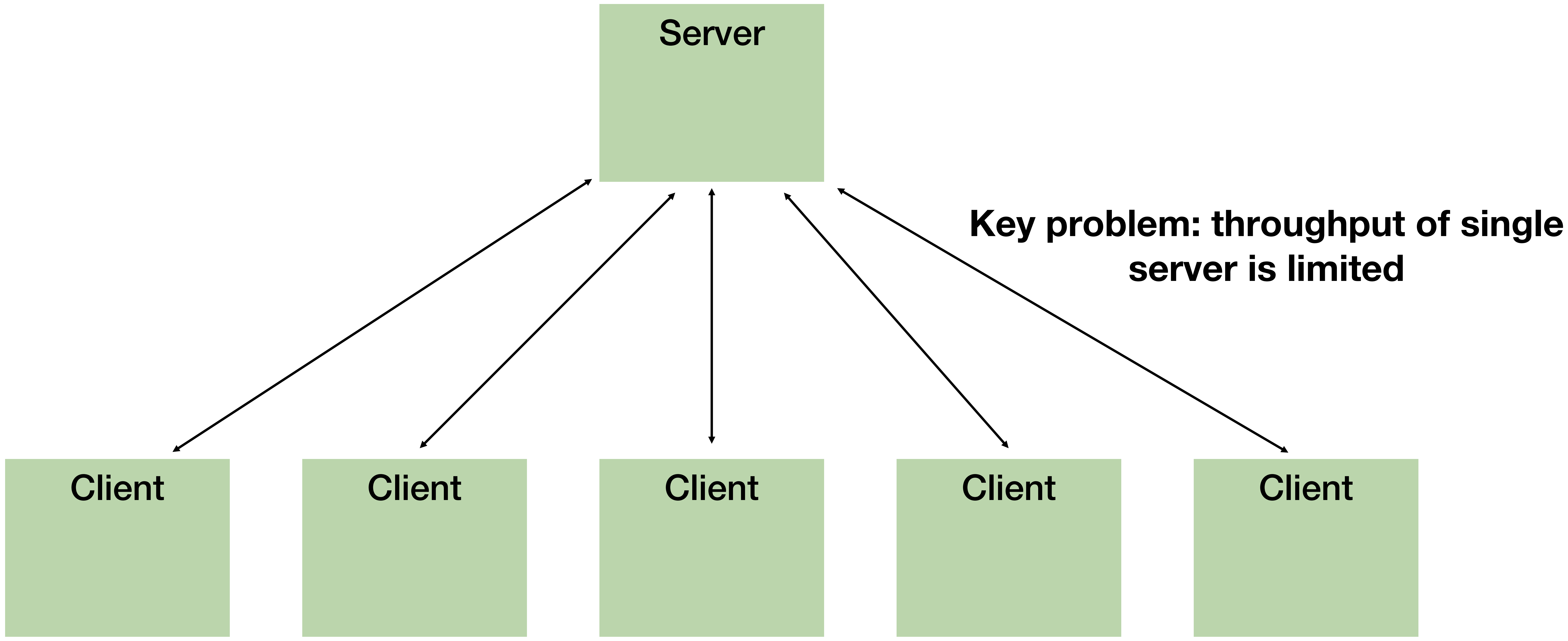
- Key idea: Partition the system into distinct tiers based on responsibilities
- Each tier scales independently of the others - .com need not know about .org
- Satisfying a single request may require multiple tiers
- DNS is a tiered architecture
  - Example: scale .com differently from .gov



Design Tiers Considering the Structure of Data

Example: GFS (Google File System, c 2010)

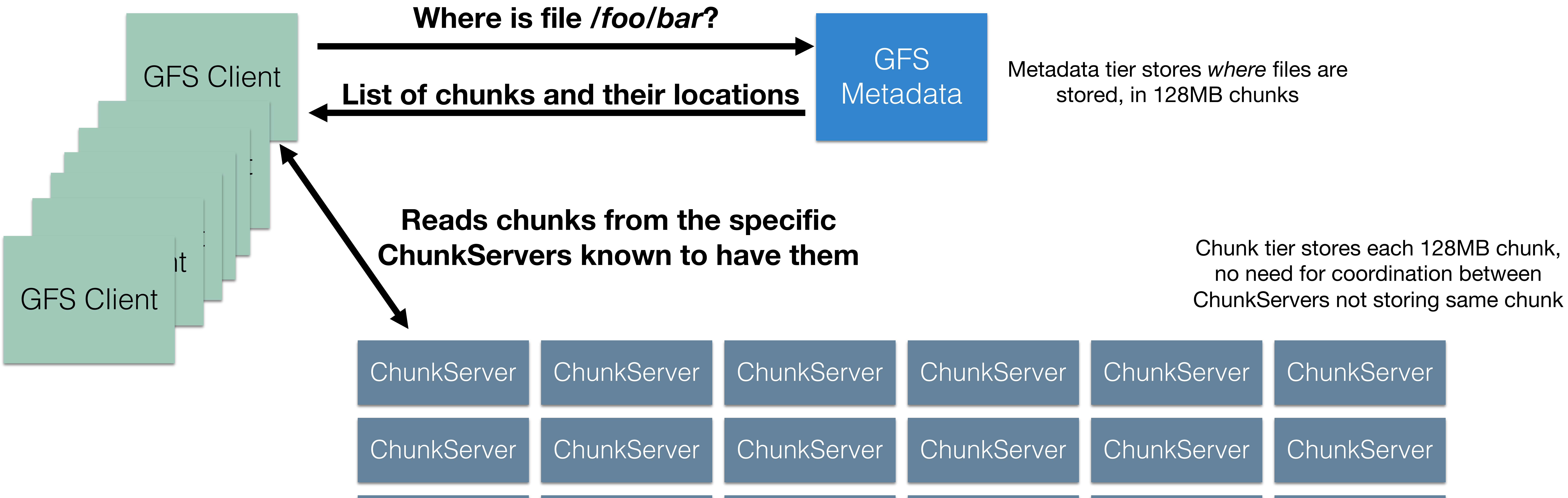
- Stated requirements: “**High sustained bandwidth is more important than low latency.** Most of our target applications place a premium on **processing data in bulk at a high rate**, while **few have stringent response time requirements for an individual read or write.**”



GFS Tiers Filesystem Metadata and File Chunks

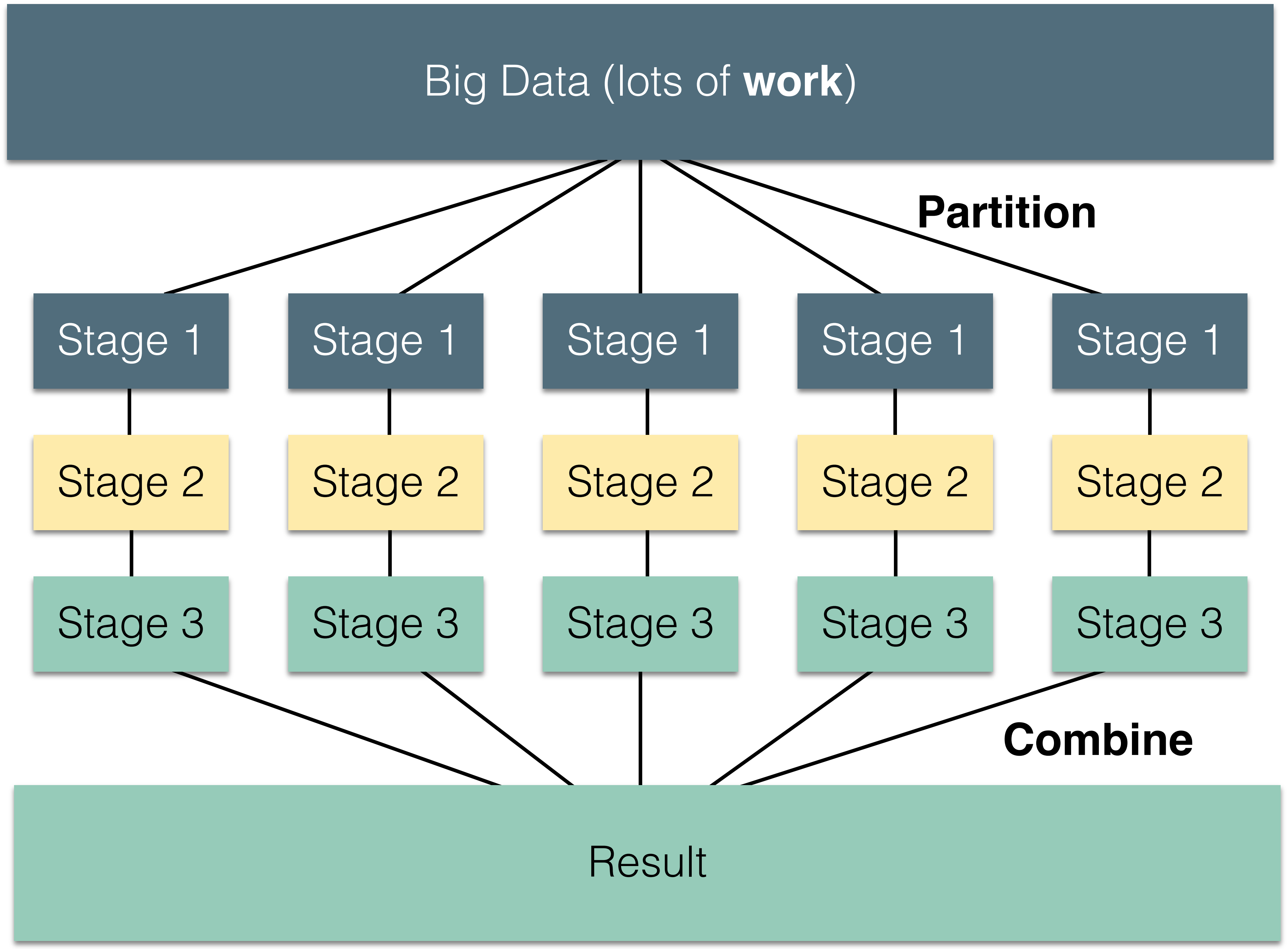
Example: GFS (Google File System, c 2010)

- 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.**"



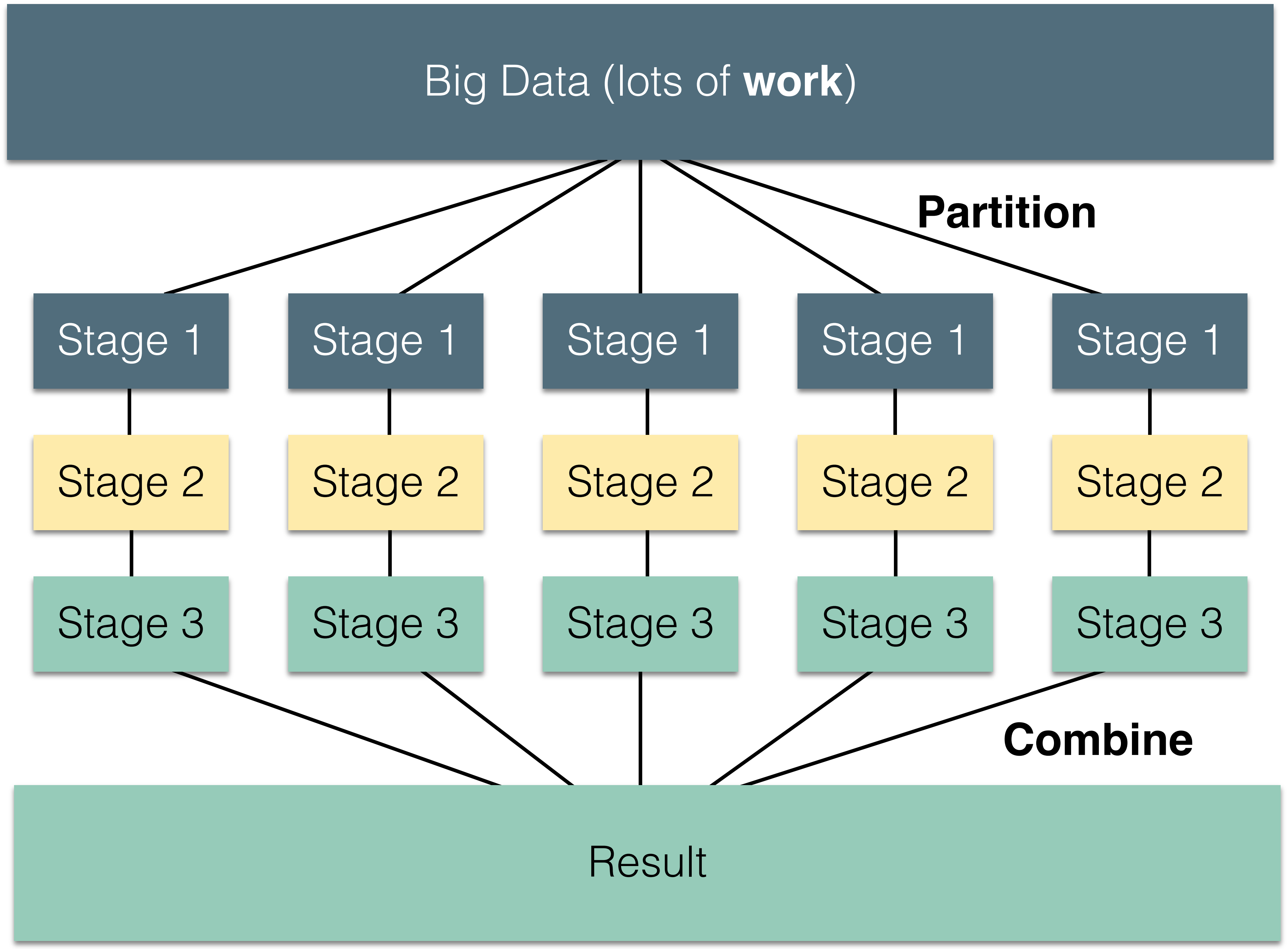
Pipeline Architectures

- The pieces correspond to stages in the transformation of data in the system
- Good for complex straight-line processes where multiple stages applied to different data, concurrently
- Each stage in the pipeline takes an input, produces an output: otherwise *stateless*
- Example: Map/Reduce splits data, filters it through stages, then combines
- Pipeline architecture allows flexibility in mapping stages to physical servers



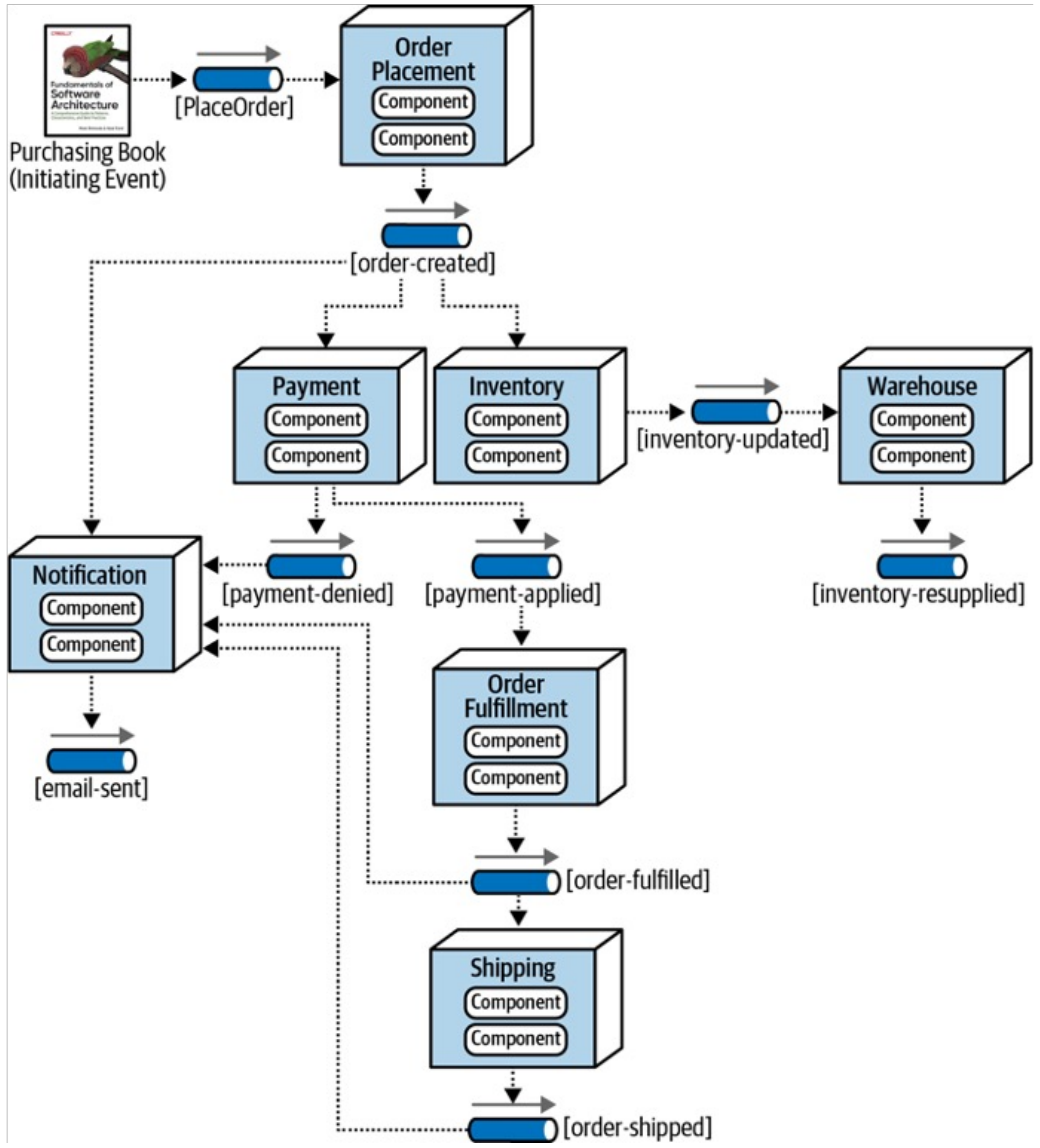
Pipeline Architectures

- Scalability/Performance:
  - Add more machines to process more data in parallel
  - Limited by bandwidth to transfer inputs/outputs between stages
- Fault tolerance: Each stage in pipeline is stateless. If one fails, it can be repeated elsewhere.



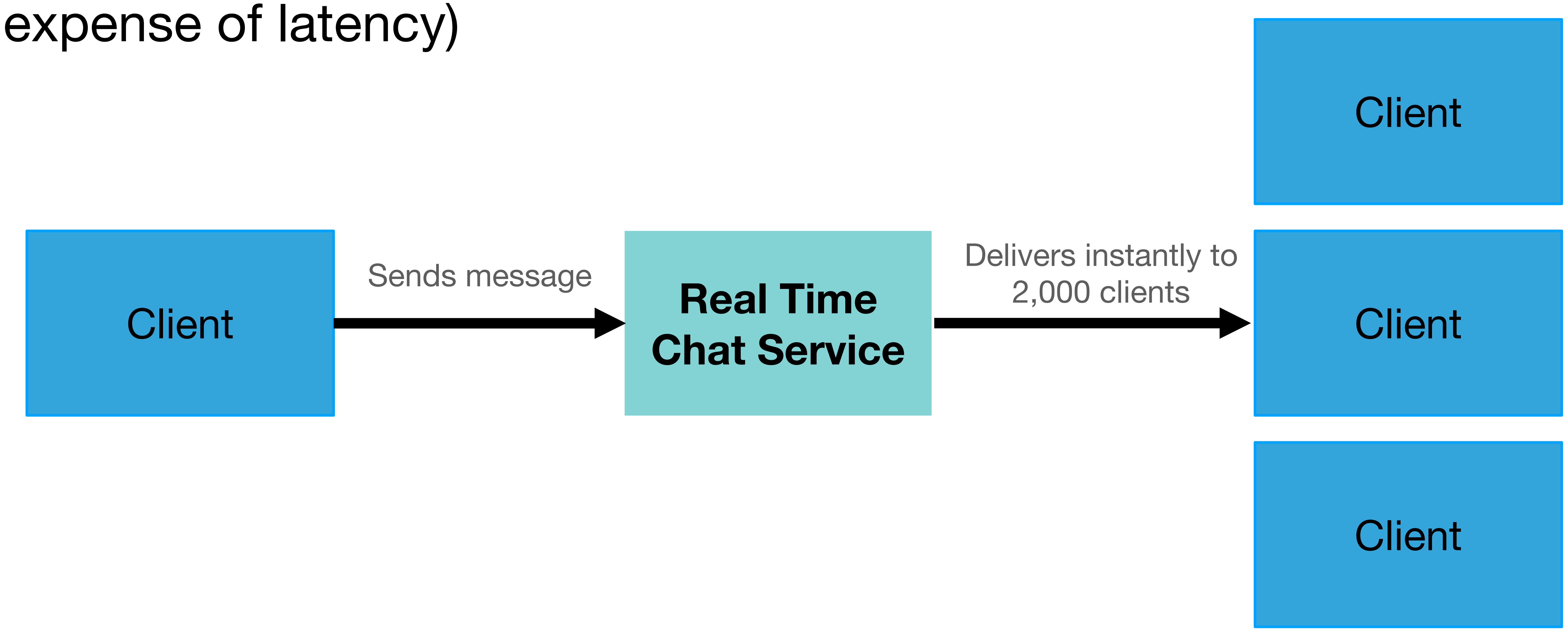
Event-Driven Architectures

- Metaphor: a bunch of bureaucrats shuffling papers
- Components correspond to stages in the flow of data through the system (not necessarily a straight-line flow)
- Very useful for *composing* other services (bureaucrats)
- Each processing unit has an in-box and one or more out-boxes
- Each unit takes a task from its inbox, processes it, and puts the results in one or more outboxes.
- Stages are typically connected by asynchronous message queues.



Event Driven Architecture: 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)

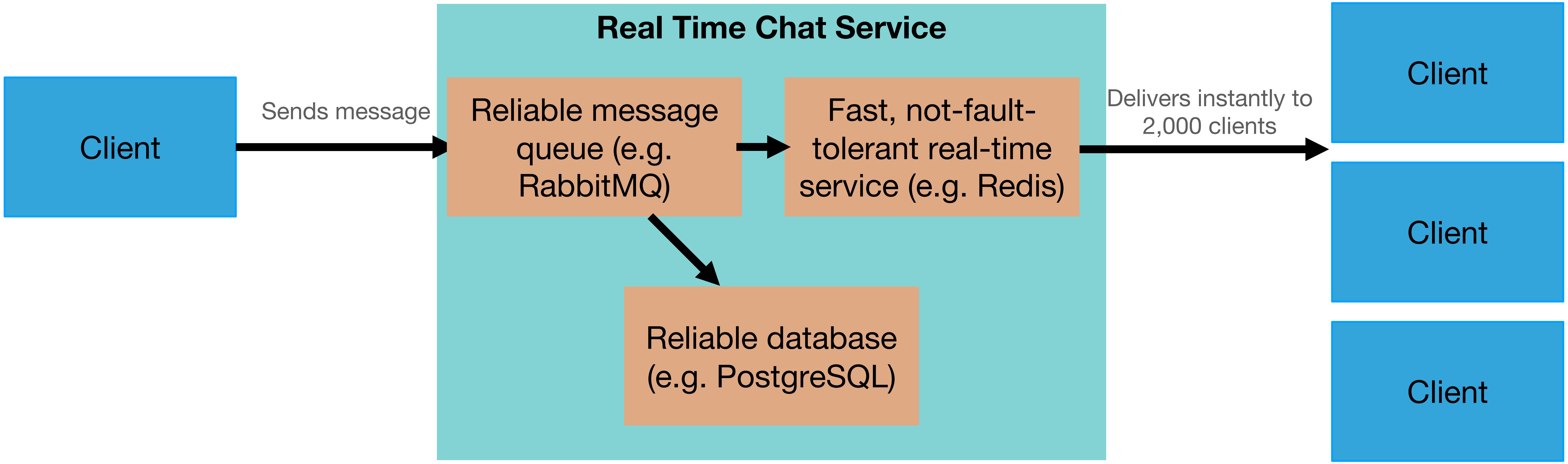


Event Driven Architecture: Reliable Real-Time Chat

- 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**.”
- Responsibilities/processing units:
  - “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

Event Driven Architecture: Reliable Real-Time Chat

- “Real time” component optimizes for speed and availability sacrificing fault-tolerance
- “Persistence” component optimizes for fault-tolerance, sacrificing speed and availability
- Reliable message queue buffers new chat messages



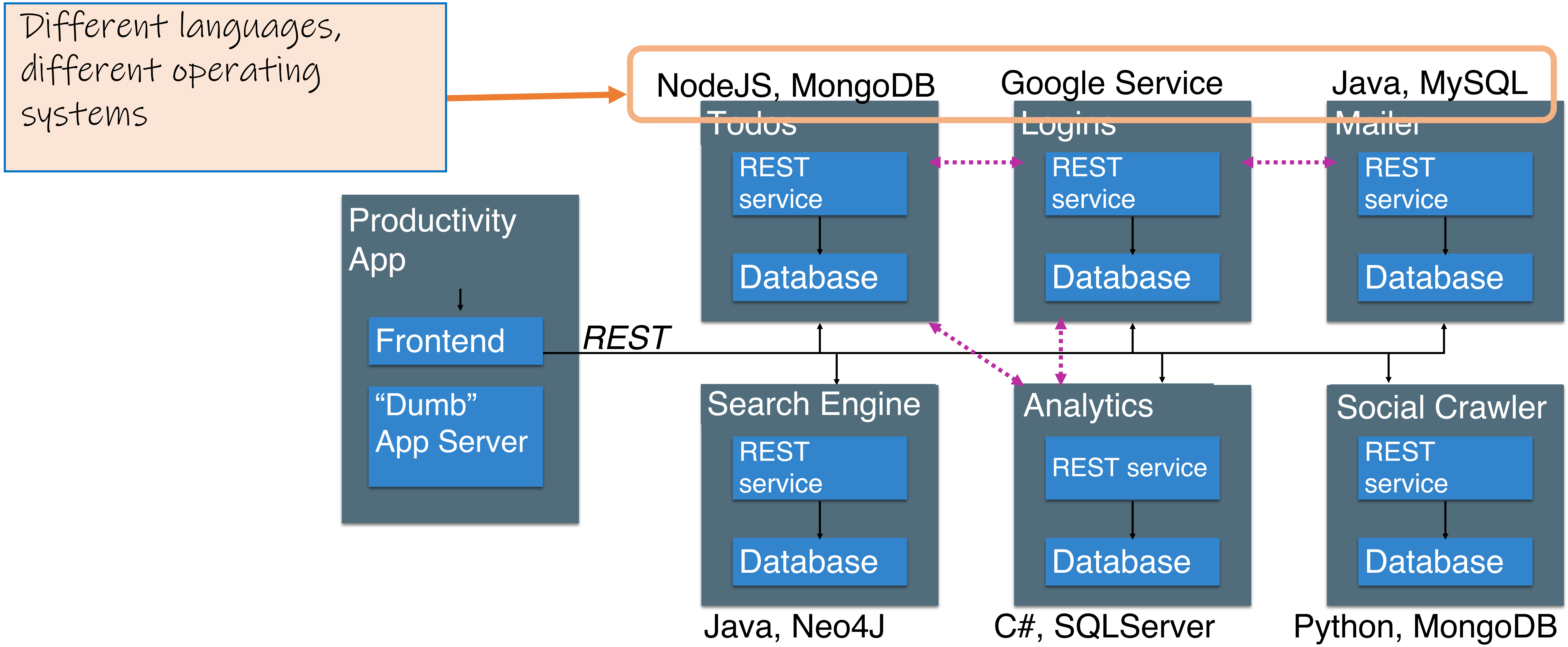
Event-Driven Architecture Tradeoffs

- Scalability:
  - Scale each processing unit separately
  - Add more processing units at a marginal cost
- Performance:
  - Message queue usually very high-throughput, relies on event processors to pick up and process messages or queue can overflow
- Fault tolerance:
  - Message queue can implement a buffer to ensure fault tolerance

Microservice Architectures

- Organize implementation around components (responsibilities)
- 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 well-defined protocol like REST

Microservices: Schematic Example

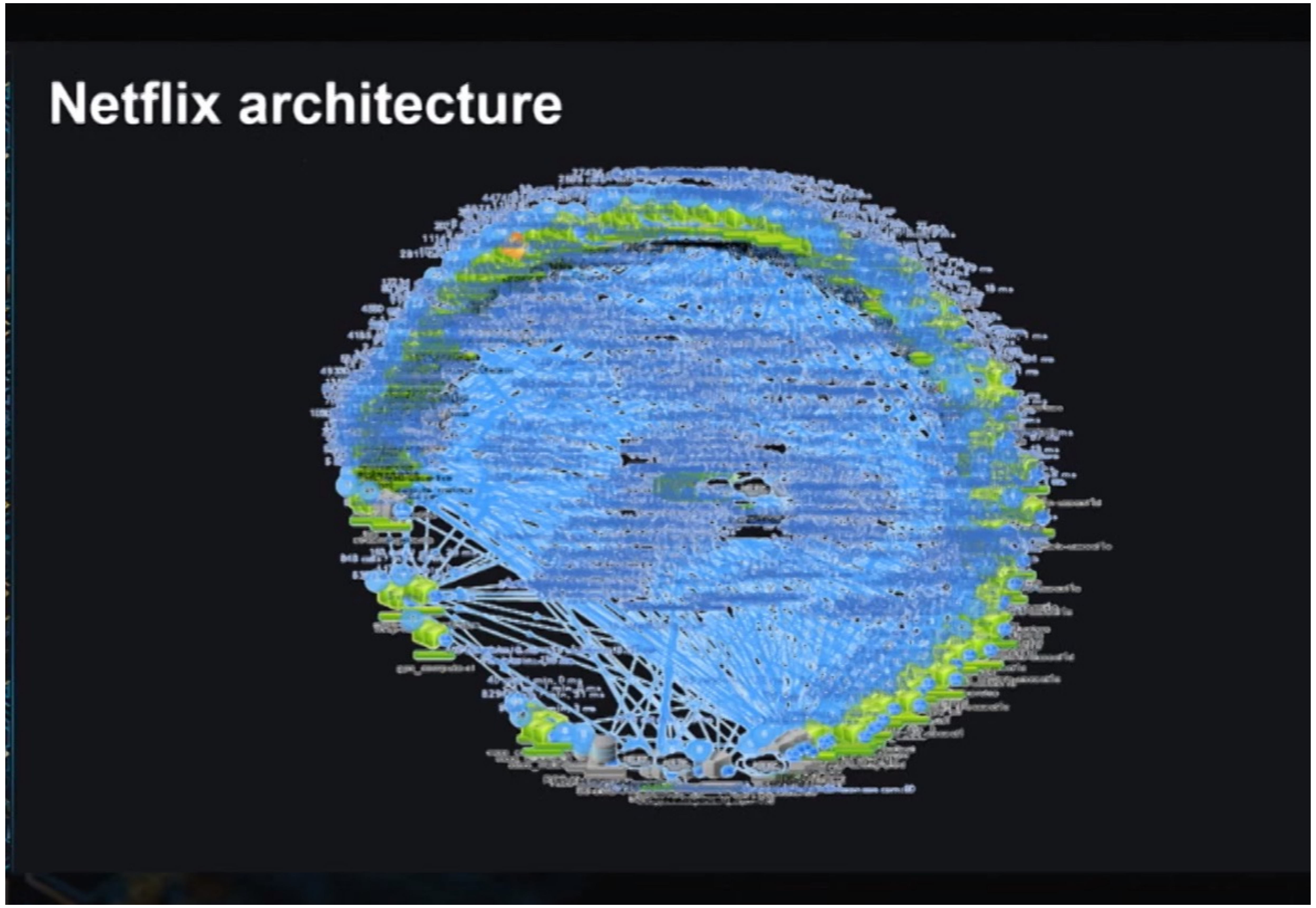


Microservice Advantages and Disadvantages

- Advantages
  - services may scale differently, so can be implemented on hardware appropriate for each (how much cpu, memory, disk, etc?). Ditto for software (OS, implementation language, etc.)
  - services are independent (yay for interfaces!) so can be developed and deployed independently
- Disadvantages
  - service discovery?
  - should services have some organization, or are they all equals?
  - overall system complexity

Microservices are (a) highly scalable and (b) trendy

- Microservices at Netflix:
  - 100s of microservices
  - 1000s of daily production changes
  - 10,000s of instances
  - BUT:
  - only 10s of operations engineers

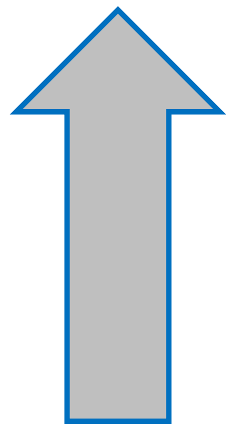


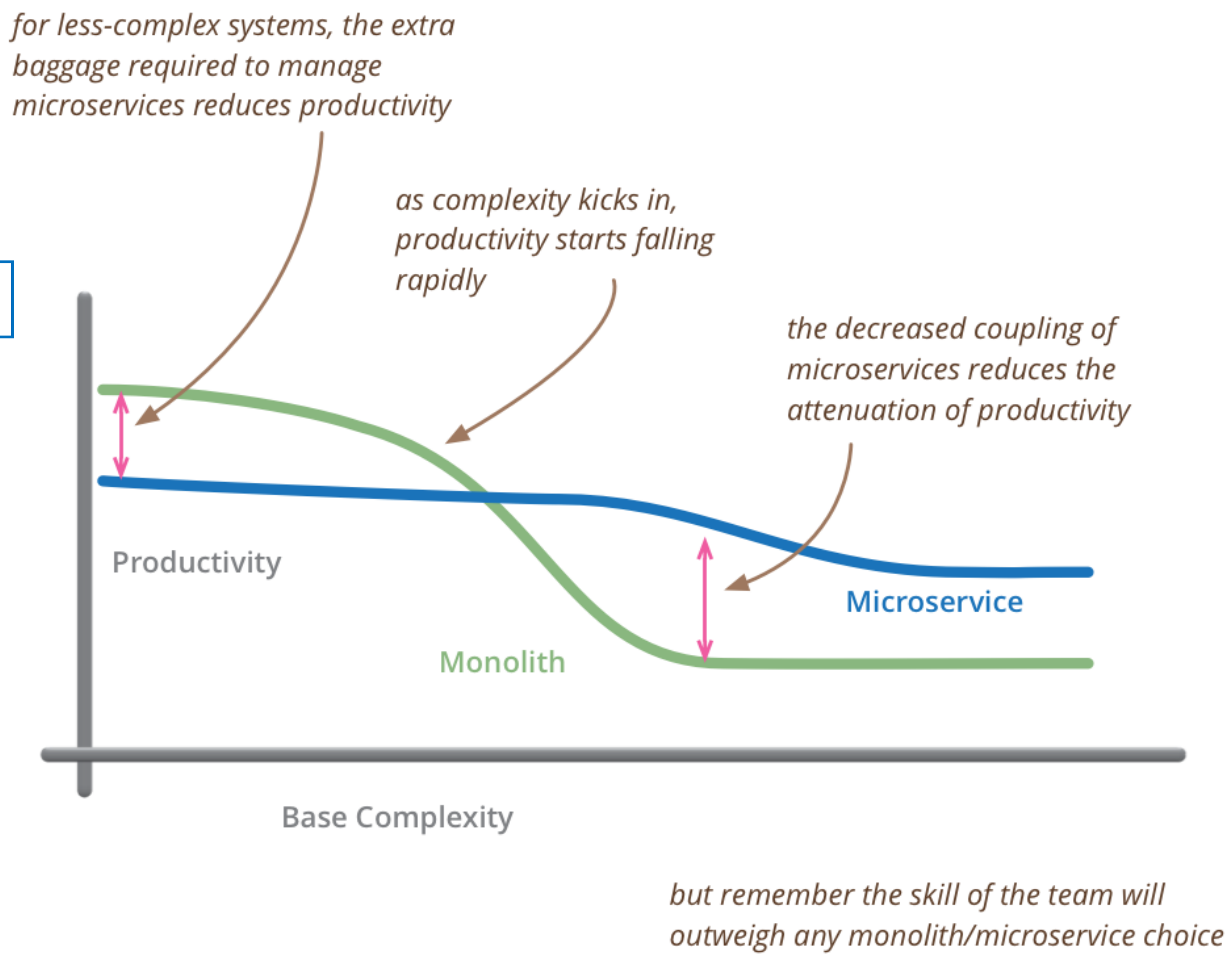
https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b

Microservices vs Monoliths

Martin Fowler's Microservices Guide - https://martinfowler.com/microservices/

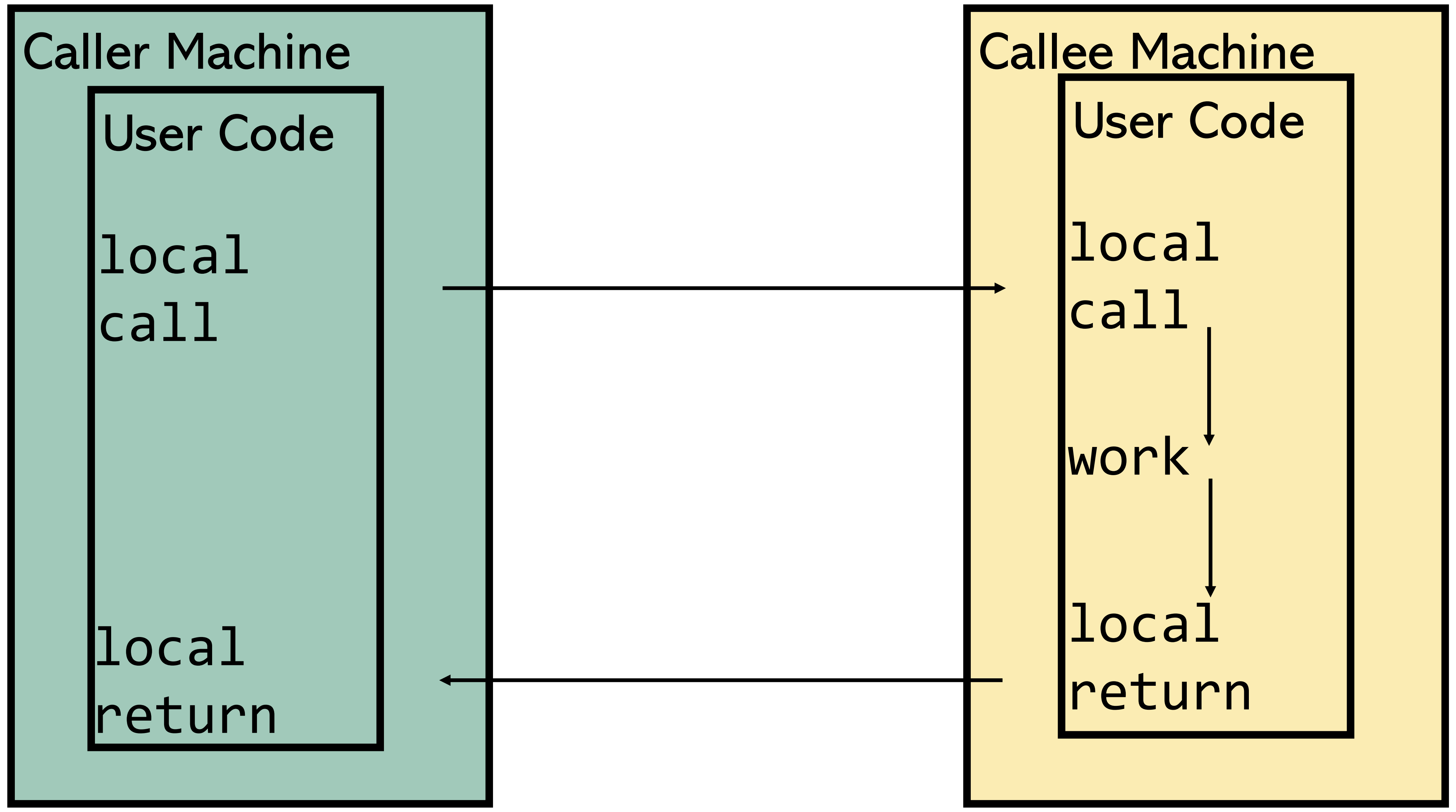
higher is better





How Do Components/Services Communicate?

- Ideally, a magic abstraction: remote procedure call (RPC) should make the separation transparent
- There are many variations of RPC
- CORBA, RMI, SOAP, and more
- The most common form of RPC today is called REST

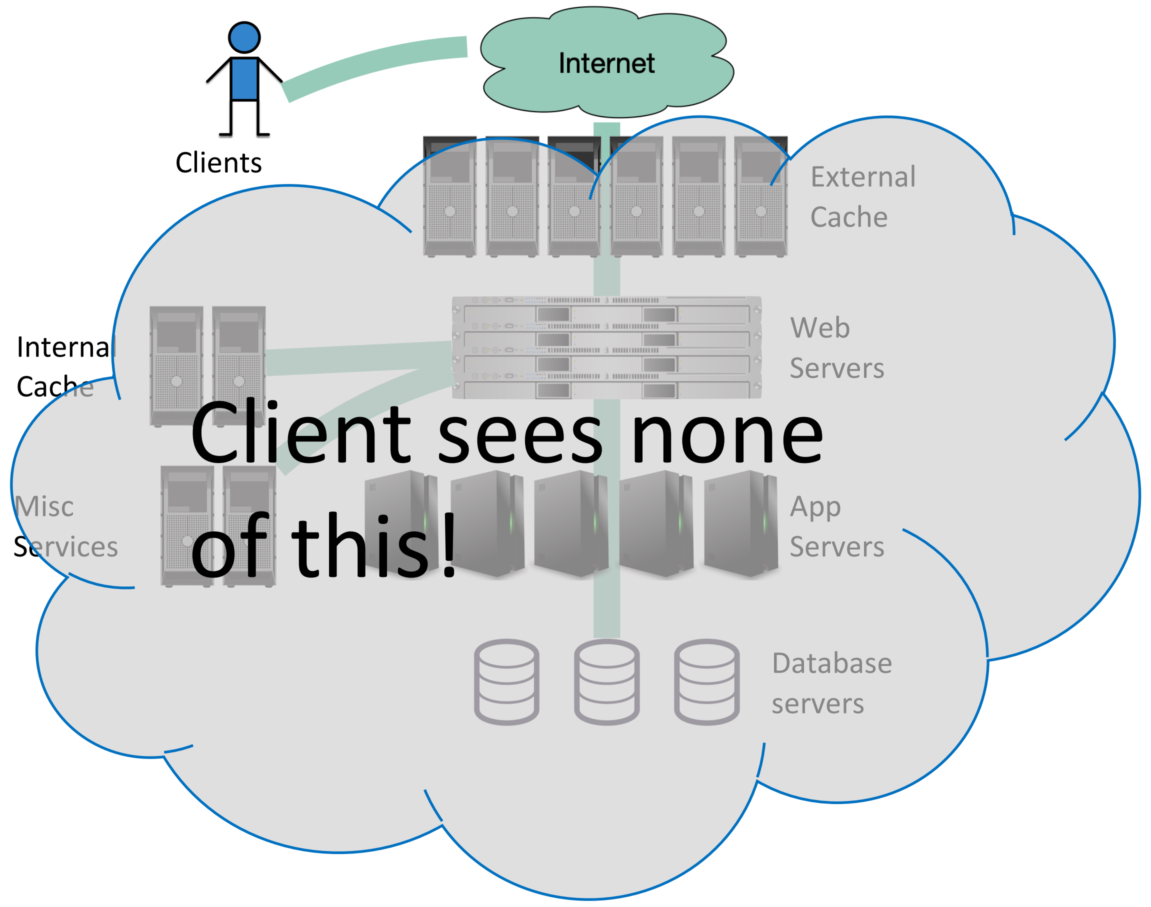


REST: Representational State Transfer

- Defined by Roy Fielding in his 2000 Ph.D. dissertation
- “Throughout the HTTP standardization process, I was called on to defend the design choices of the Web. That is an extremely difficult thing to do... I had comments from well over 500 developers, many of whom were distinguished engineers with decades of experience. That process honed my model down to a core set of principles, properties, and constraints that are now called REST.”
- Not just a transport protocol, not a protocol definition language: a design philosophy
- Interfaces that follow REST principles are called RESTful

REST Principles

- Single Server - As far as the client knows, there's just one
- Stateless - Each request contains enough information that a different server could process it (if there were multiple…)
- Uniform Cacheability - Each request is identified as cacheable or not.
- Uniform Interface - Standard way to specify interface



Nouns are represented as URIs

- In a RESTful system, the server is visualized as a store of resources (nouns), each of which has some data associated with it.
- URIs represent these resources
- Examples:
  - /cities/losangeles
  - /transcripts/00345/graduate (student 00345 has several transcripts in the system; this is the graduate one)
- Anti-examples:
  - /getCity/losangeles
  - /getCitybyID/50654
  - /Cities.php?id=50654

We prefer plural nouns for toplevel resources, as you see here.

Useful heuristic: if you were keeping this data in a bunch of files, what would the directory structure look like? But you don't have to actually keep the data in that way.

Verbs are represented as http methods

- In REST, there are four things you can do with a resource
- POST: requests the server to create a resource
  - there are several ways in which the value for the new resource can be transmitted (more In a minute)
- GET: requests the server to respond with a representation of the resource
- PUT: requests the server to replace the value of the resource by the given value
- DELETE: requests the server to delete the resource

You say you want parameters?

There are at least 3 ways to associate parameters with a request:

- path parameters. These specify portions of the path to the resource. For example, your REST protocol might allow a path like

```
/transcripts/00345/graduate
```

- query parameters. These are part of the URI and are typically used as search items. For example, your REST protocol might allow a path like

```
/transcripts/graduate?lastname=covey&firstname=avery
```

- body parameters. You can put additional parameters or information in the body, using any coding that you like.

Example interface #1: a todo-list manager

- Resource: /todos
  - GET /todos - get list all of my todo items
  - POST /todos - create a new todo item (data in body)
- Resource: /todos/:todoItemID
  - :todoItemID is a path parameter
  - GET /todos/:todoItemID - fetch a single item by id
  - PUT /todos/:todoItemID - update a single item (new data in body)
  - DELETE /todos/:todoItemID - delete a single item

Example Interface #2: a database of transcripts

```
POST /transcripts
 -- adds a new student to the database,
 -- returns an ID for this student.
 -- requires a body parameter 'name', url-encoded (eg name=avery)
 -- Multiple students may have the same name.
GET  /transcripts/:ID
 -- returns transcript for student with given ID.  Fails if no such student
DELETE /transcripts/:ID
 -- deletes transcript for student with the given ID, fails if no such student
POST /transcripts/:studentID/:courseNumber
 -- adds an entry in this student's transcript with given name and course.
 -- Requires a body parameter 'grade', url-encoded
 -- Fails if there is already an entry for this course in the student's transcript
GET  /transcripts/:studentID/:courseNumber
 -- returns the student's grade in the specified course.
 -- Fails if student or course is missing.
GET  /studentids?name=string
 -- returns list of IDs for student with the given name
```

Remember the heuristic: if you were keeping this data in a bunch of files, what would the directory structure look like?

Didn't seem to fit the model, sorry ☹️

Specify REST APIs using OpenAPI

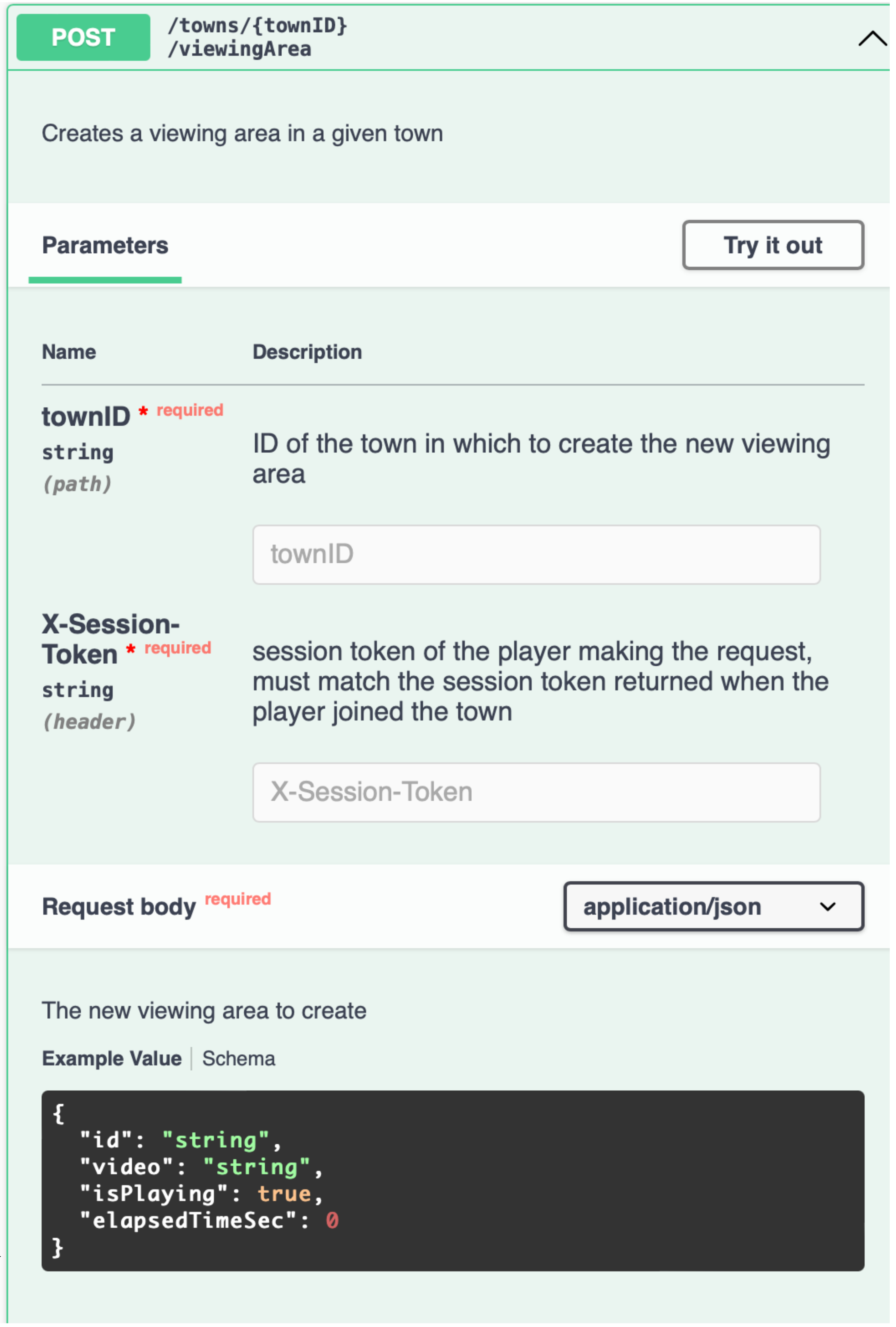
- The specification of the transcript API on the last slide is RESTful, but is not machine-readable
- A machine-readable specification is useful for:
  - Automatically generating client and server boilerplate, documentation, examples
  - Tracking how an API evolves over time
  - Ensuring that there are no misunderstandings

```
/towns/{townID}/viewingArea:
post:
  operationId: CreateViewingArea
responses:
  '204':
description: No content
'400':
description: Invalid values specified
content:
  application/json:
schema:
  $ref: '#/components/schemas/InvalidParametersError'
description: Creates a viewing area in a given town
tags:
  - towns
security: []
parameters:
  - description: ID of the town in which to create the new viewing area
in: path
name: townID
required: true
schema:
  type: string
  - description: |-
  session token of the player making the request, must
match the session token returned when the player joined the town
in: header
name: X-Session-Token
required: true
schema:
  type: string
requestBody:
  description: The new viewing area to create
required: true
content:
  application/json:
schema:
  $ref: '#/components/schemas/ViewingArea'
description: The new viewing area to create
```

TSOA Auto-Generates OpenAPI Specifications from TypeScript

```typescript
@Route('towns')
export class TownsController extends Controller {


/**
 * Creates a viewing area in a given town
 *
 * @param townID ID of the town in which to create the new viewing area
 * @param sessionToken session token of the player making the request, must
 *        match the session token returned when the player joined the town
 * @param requestBody The new viewing area to create
 *
 * @throws InvalidParametersError if the session token is not valid, or if the
 *          viewing area could not be created
 */
@Post('{townID}/viewingArea')
@Response<InvalidParametersError>(400, 'Invalid values specified')
public async createViewingArea(
  @Path() townID: string,
  @Header('X-Session-Token') sessionToken: string,
  @Body() requestBody: ViewingArea,
)
```

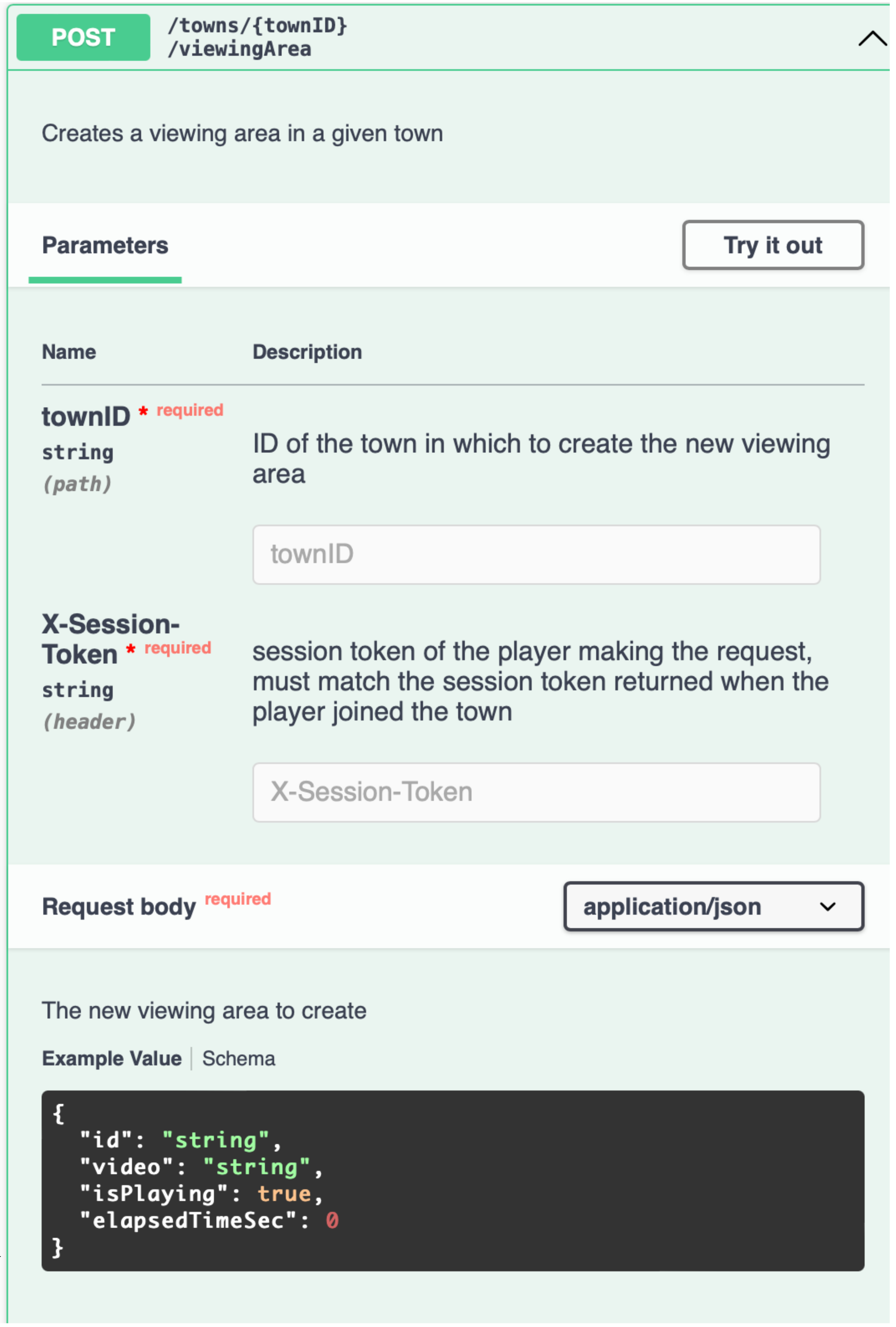
Open API Specification



TSOA Auto-Generates OpenAPI Specifications from TypeScript

```
@Route('towns')
export class TownsController extends Controller {


/**
 * Creates a viewing area in a given town
 *
 * @param townID ID of the town in which to create the new viewing area
 * @param sessionToken session token of the player making the request, must
 *        match the session token returned when the player joined the town
 * @param requestBody The new viewing area to create
 *
 * @throws InvalidParametersError if the session token is not valid, or if the
 *          viewing area could not be created
 */
@Post('{townID}/viewingArea')
@Response<InvalidParametersError>(400, 'Invalid values specified')
public async createViewingArea(
  @Path() townID: string,
  @Header('X-Session-Token') sessionToken: string,
  @Body() requestBody: ViewingArea,
)
```

Open API Specification



Converting JavaScript Errors to HTTP Errors

- Under the hood, we use the popular express web server for NodeJS
- Express uses a pipeline architecture for processing requests
- This code snippet runs after our controller, inspects any error that might be thrown, and returns an HTTP error of 400, 422 or 500
- Example to return a 400 error:

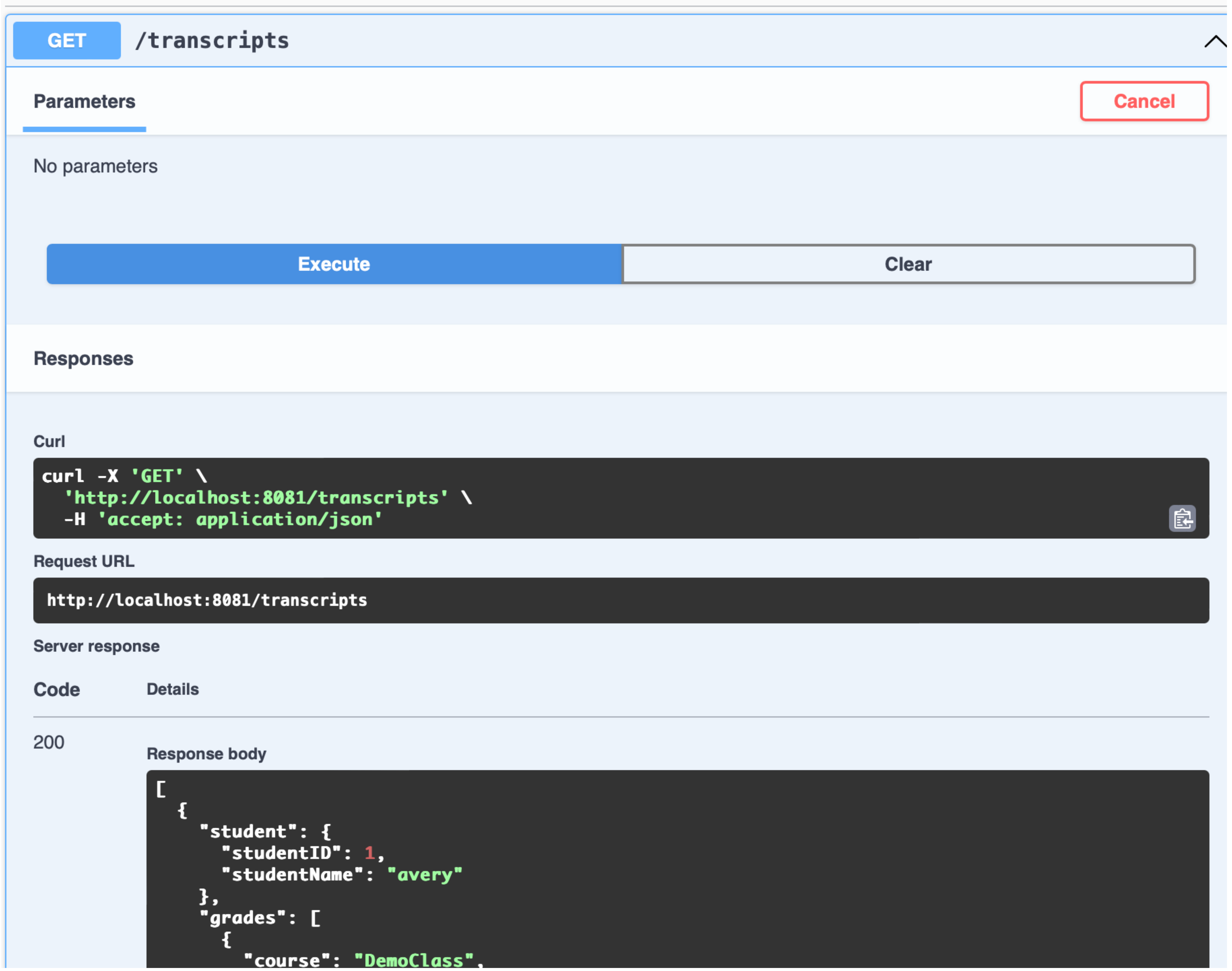
```
throw new InvalidParametersError('Some message')
```

```
//server.ts

app.use(
  (
    err: unknown, _req: Express.Request, res: Express.Response,
    next: Express.NextFunction,
  ): Express.Response | void => {
    if (err instanceof ValidateError) {
      return res.status(422).json({
        message: 'Validation Failed',
        details: err?.fields,
      });
    }
    if(err instanceof InvalidParametersError){
      return res.status(400).json({
        message: 'Invalid parameters',
        details: err?.message
      })
    }
    if (err instanceof Error) {
      console.trace(err);
      return res.status(500).json({
        message: 'Internal Server Error',
      });
    }

    return next();
  },
)
```

Activity: Build the Transcript REST API

```
@Route('transcripts')
export class TranscriptsController extends
Controller {

    @Get()
    public getAll() {
        return db.getAll();
    }
```

Open API Specification



Review: Learning Objectives for this Lesson

By the end of this lesson, you should be able to…

- Recognize common software architectures
- Understand tradeoffs of scalability, performance, and fault tolerance between these architectures
- Describe what makes web services RESTful, and implement a REST API