Distributed Sys & Course Wrap-up

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

Ash Fujiyama
CV Kunjeti
Felix Sun
Heyi Liu
Kevin Bernat
Lang Qin
Sean Chuang
Serena Chen
Yuna Shao

Logistics

- Project released
  - Due May 1st at midnight, please get started if you haven't already
  - Autograder to be posted soon
  - NOTE: part of it is manually checked, not auto-graded

- HW4
  - Due this Friday
  - Autograder posted

- Last Checkin to be released soon
  - Due May1st at midnight (late deadline over reading days)
  - (Post Semester Survey)

Poll Everywhere

pollev.com/tqm

- Any questions? (On anything)
  - This is the chance for catchup questions, same at the beginning of next lecture.

Lecture Outline

- **Intro to Distributed Systems**
- Course wrap-up

What are distributed systems?

- A group of computers communicating over the network by sending messages, which interact to accomplish some common task
  - There is no shared state (e.g. memory)
  - Individual computers (nodes) can fail
  - The network itself can fail (Drop messages, corrupt messages, delay messages, etc.)

Why do we care?

- They are a really interesting problem to work with
- Most applications we interact with are distributed systems

Distributed Systems Concerns

- How do we make it so that the computers work together:
  - Correctly
  - Consistent
  - Efficiently
  - At (huge) scale
  - High availability

- Despite issues with the network

- Despite some computers crashing

- Despite some computers being compromised

Distributed Systems: Pessimistic View

- Considered a very hard topic
  - Involves many of the topics covered in this course and more
  - CIS 5050 spends ~8 lectures covering things already introduced here. (out of 25 lectures)

- “The most thought per line of code out of any course”
  - Hal Perkins Circa 2019

- “A distributed system is one where you can’t get your work done because some machine you’ve never heard of is broken.”
  - Leslie Lamport, circa 1990

Distributed Systems Topics

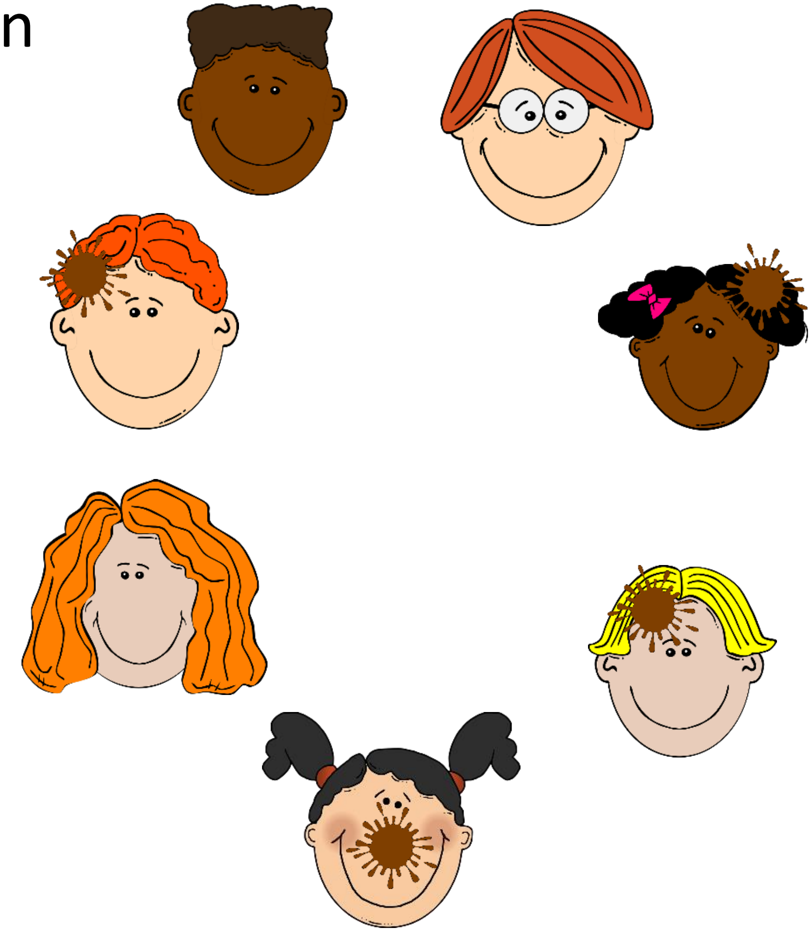
- Concurrency on a single node
  - Threads, processes, pipes, locks, etc.
- Networking
  - HTTP, DNS, TCP, Sockets, etc.
- Synchronization across network nodes
  - Common Knowledge, Clocks, coordination, leader elections, etc.
- Fault Tolerance & Robustness
  - Byzantine fault tolerance, ACID, etc.

Distributed Systems Topics

- Concurrency on a single node
  - Threads, processes, pipes, locks, etc.
- Networking
  - HTTP, DNS, TCP, Sockets, etc.
- **Synchronization across network nodes**
  - **Common Knowledge, Clocks, coordination, leader elections, etc.**
- **Fault Tolerance & Robustness**
  - **Byzantine fault tolerance, ACID, etc.**

Muddy Foreheads

- Assume the following situation
  - There are n children, k get mud on their foreheads
  - Children sit in circle.
  - Teacher announces, "Someone has mud on their forehead
  - Teacher repeatedly asks "Raise your hand if you know you have mud on your forehead."
  - What happens?



Muddy Foreheads

- Assume the following situation
  - There are n children, k get mud on their foreheads
  - Children sit in circle.
  - Teacher announces, "Someone has mud on their forehead
  - Teacher repeatedly asks "Raise your hand if you know you have mud on your forehead."
  - What happens?
    - The answer is not "no one raises their hand"

The Muddy Forehead "Paradox"

- If k > 1, the teacher didn’t say anything anyone didn’t already know!

- Yet the information is crucial to let the children solve the problem

Common Knowledge

- There’s a difference between what you know and what you know others know

- And what others know you know

- And what others know you know about what you know

- And what you know others know you know about what they know

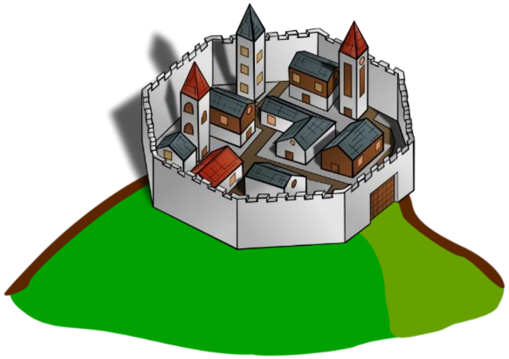
Muddy Forehead Alteration

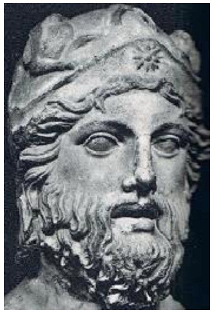
- What if the teacher pulled each student aside individually and told them “at least one student has mud on their forehead”?

  - Would our solution still work?

Generals Problem

- Two generals, on opposite sides of a city on a hill.
- If they attack simultaneously, they will be victorious. If one attacks without the other, they will both be defeated.
- Can communicate by messenger. Messengers can get lost or be captured.
- How do they ensure they can take the city?





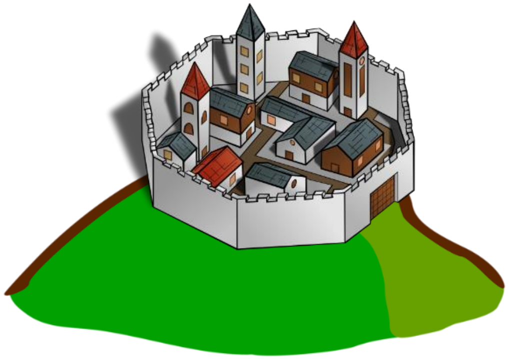
Coordinated Attack

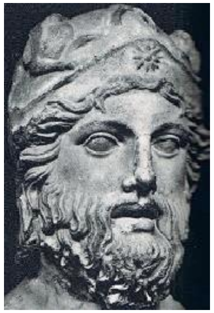
- **Answer**: There does not exist a protocol to decide when and whether to attack.
- **Proof by contradiction**. Assume a protocol exists. Let the minimum number of messages received in any terminating execution be n. Consider the last message received in one such execution.
- The sender's decision to attack does not depend on whether or not the message is received; sender must attack. Since the sender attacks, the receiver must also attack when the message is not received.
- Therefore, the last message is irrelevant, and there exists an execution with n-1 message deliveries. n was the minimum! Contradiction.

Generals Problem

- To coordinate an attack, the problem requires common knowledge

- With the messengers, common knowledge is never reached.





- What happens when we add more generals?

- What happens when some of the generals are malicious?

Example: RPC

- Remote Procedure Call: When a program is able to invoke a function on another computers address space, and then get the results.

- Usually done as a form of “Message Passing”
  - Client calls a function that sends a “message” over the network
  - A server receives the message, executes the function, and sends the response back

- Even in this simple, example, issues can arise

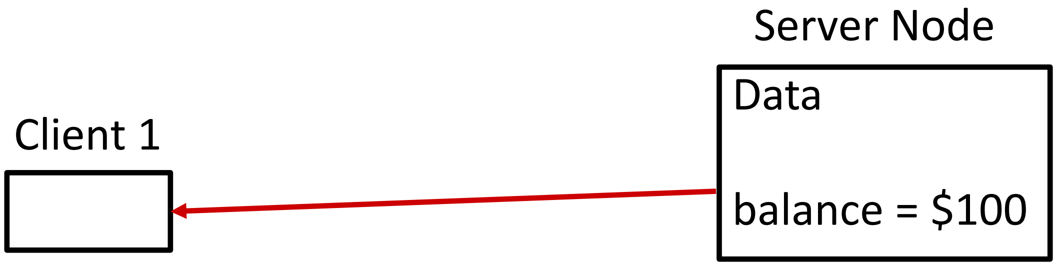
Example: RPC

- Consider: Client wants to read their current Bank Account Balance
  - Client may call a function like get_balance()

Server Node
Data
balance = $100
Client 1

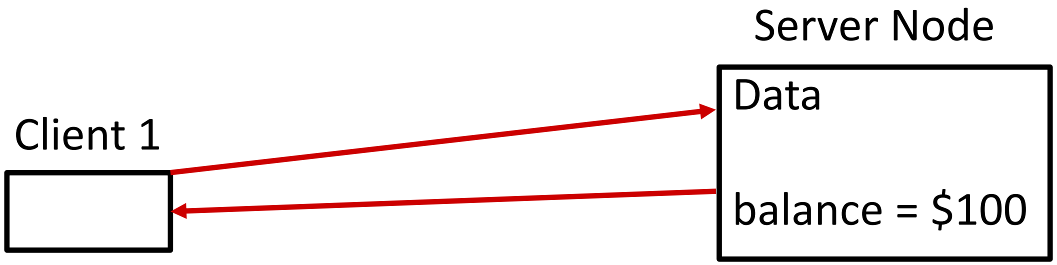
Example: RPC

- Consider: Client wants to read their current Bank Account Balance
  - Client may call a function like get_balance()
  - get_balance() will reach out to the server across the network



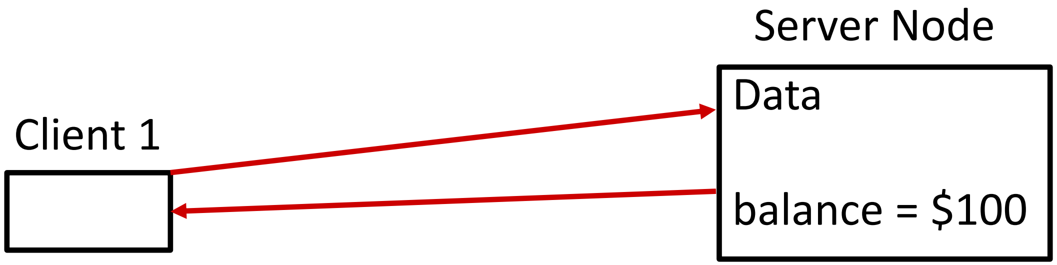
Example: RPC

- Consider: Client wants to read their current Bank Account Balance
  - Client may call a function like get_balance()
  - get_balance() will reach out to the server across the network
  - Server processes the request, and sends it back



Example: RPC

- Consider: Client wants to read their current Bank Account Balance
  - Client may call a function like get_balance()
  - get_balance() will reach out to the server across the network
  - Server processes the request, and sends it back
  - Client returns from the function "get_balance()"



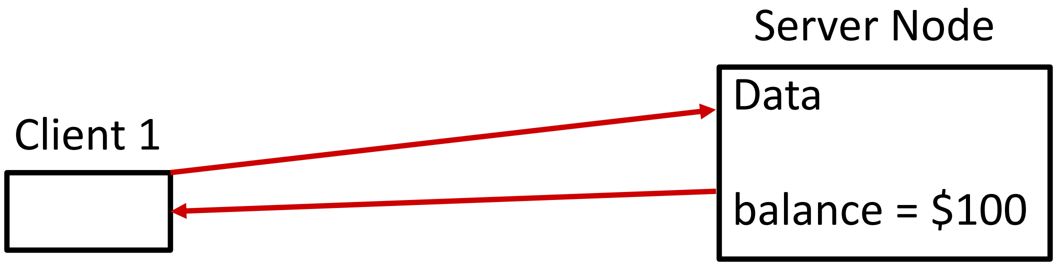
Client was blocked while waiting for the server to respond.

Program that called **`get_balance()`** probably doesn't need to know much about the network messaging

Blank Slide

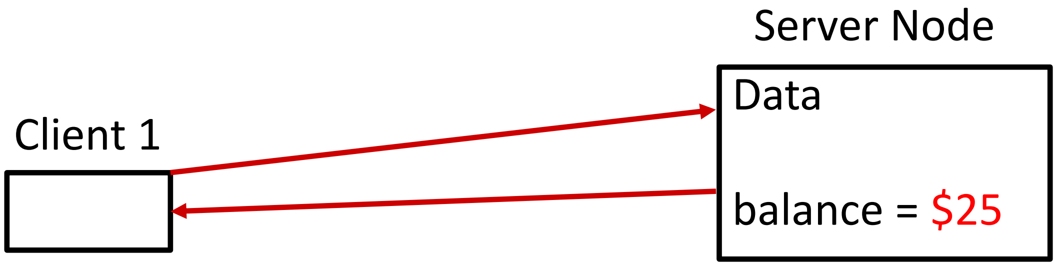
Example: RPC Transaction

- Consider: Client wants to withdraw $75 from their bank account
  - Client may call a function like withdraw(75)
  - withdraw() will reach out to the server across the network



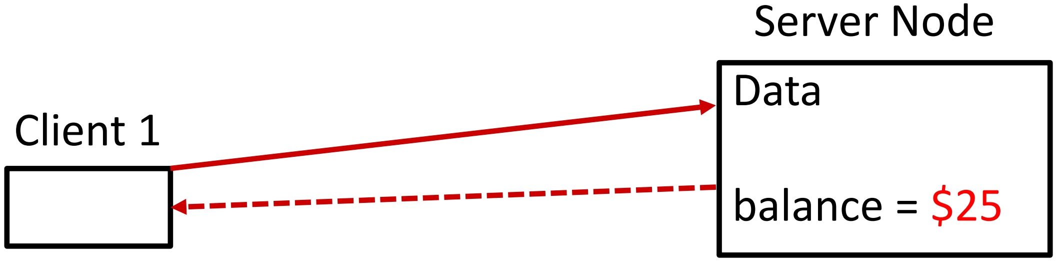
Example: RPC Transaction

- Consider: Client wants to withdraw $75 from their bank account
  - Client may call a function like withdraw(75)
  - withdraw() will reach out to the server across the network
  - Server processes the request, and sends it back



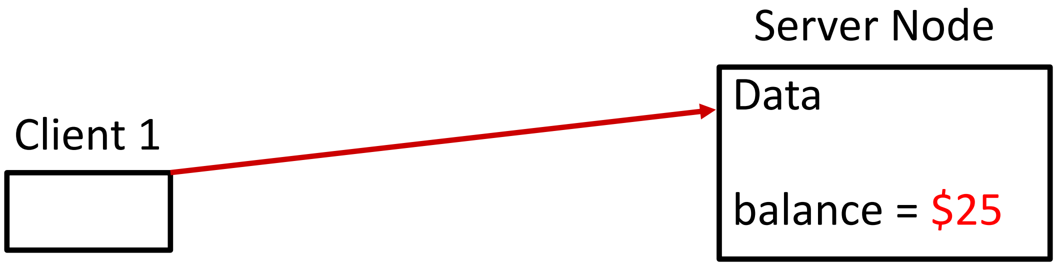
Example: RPC Transaction

- Consider: Client wants to withdraw $75 from their bank account
  - Client may call a function like withdraw(75)
  - withdraw() will reach out to the server across the network
  - Server processes the request, and sends it back
    - … But what if the connection is dropped before client receives response!



Example: RPC Transaction

- Server processes the withdraw request, and sends it back
  - … But what if the connection is dropped before client receives response!
- Let’s say connection is re-established and client resends “withdraw(75)”…



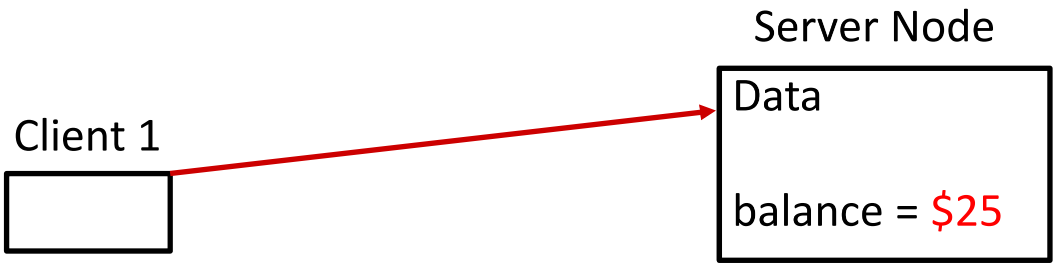
Question: Does TCP Solve This?

- If we were using TCP, is this situation even possible?
  - TCP: provides an abstraction of a reliable stream of bytes.
  - TCP: each packet is acknowledged between user and receiver and automatically resent.

- Yes: this can still happen.
  - TCP Ensures that packets are sent in a specific order and are acknowledged before it is “successfully written”.
  - Does not ensure that the network (or server itself) goes down
  - Does not ensure that the function we want to execute on the server worked or whether it actually happened.

Example: RPC Transaction

- Server processes the withdraw request, and sends it back
  - … But what if the connection is dropped before client receives response!
- Let's say connection is re-established and client resends "withdraw(75)"…
  - How does the server know if this is the same request as last time, or another request to withdraw $75
  - How does the server know what the client is "intending"



Terminology

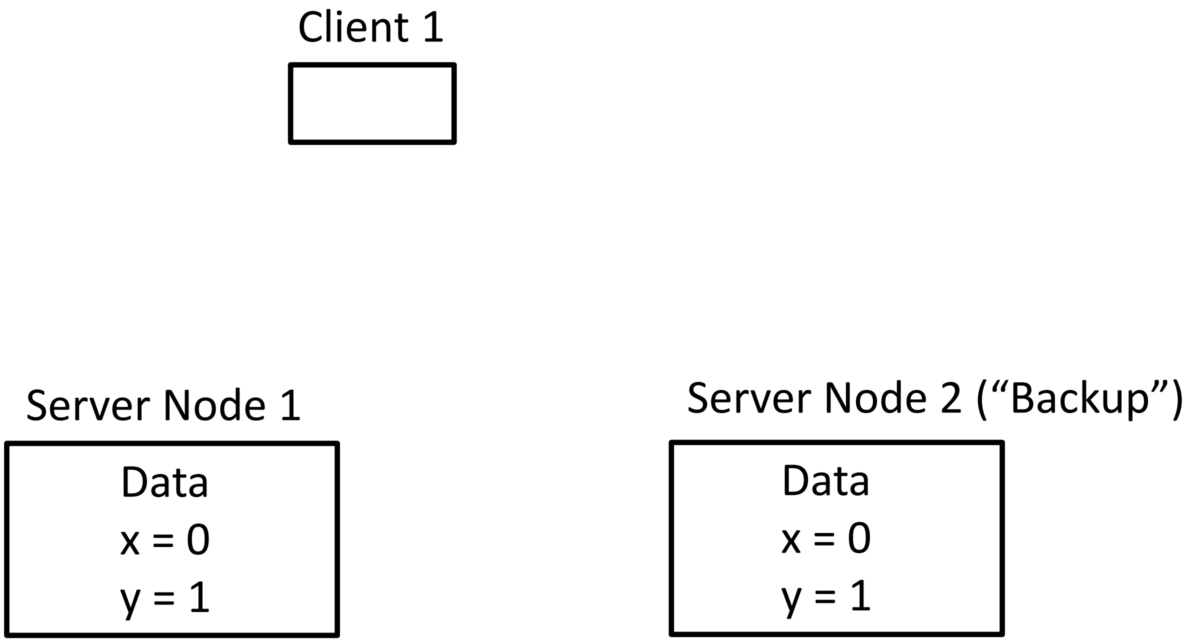
- Exactly Once:
  - Hardest to guarantee
  - That something happens and it only happens exactly one time.
  - Requires that the clients have an ID and each request has an ID number.
  - Servers must also keep a history of previously processed requests and their ID number so that the server can respond to duplicate/old requests.

Terminology

- At Most Once:
  - That a request is executed at most once (e.g. 0 times or 1 time)
  - Usually means the client sends the request once and only once.
  - Usable in some cases, but sometimes we need to guarantee that something happened.
- At Least Once:
  - That the thing is executed at least one time.
  - This is fine for things like "Reading a value" or "setting" a value Other operations may get different results if done multiple times (Like our transaction)
- Exactly Once:

Blank Slide

Example: Consistent State



Client 1

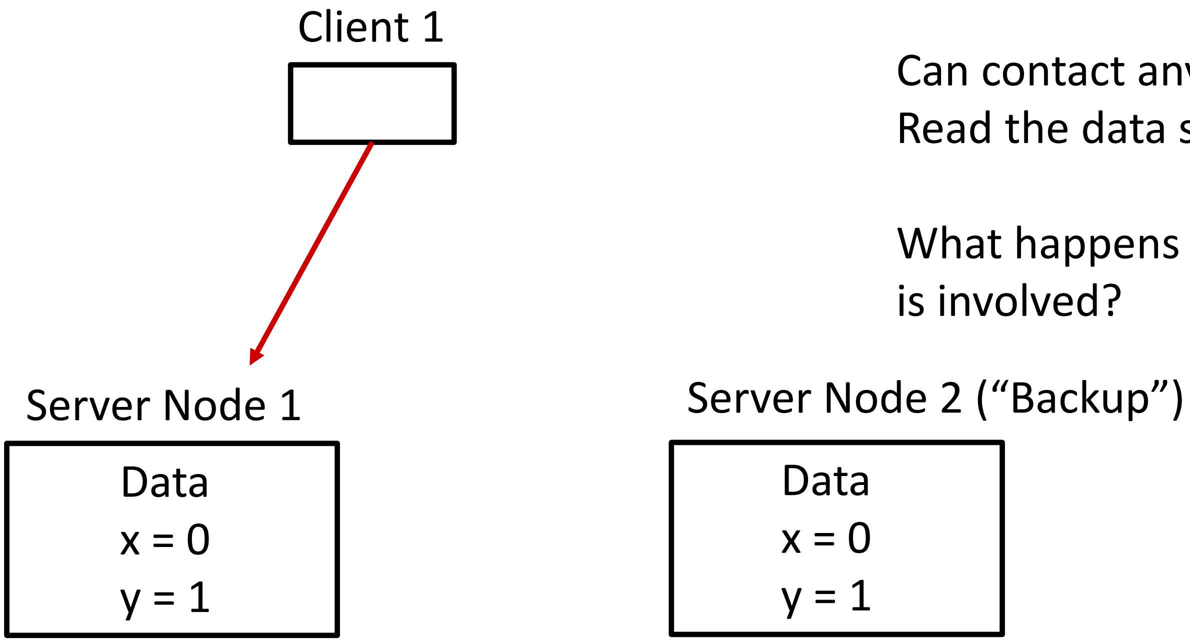
Server Node 1

Data
x = 0
y = 1

Server Node 2 (“Backup”)

Data
x = 0
y = 1

Example: Consistent State



Client 1

Server Node 1

Data
x = 0
y = 1

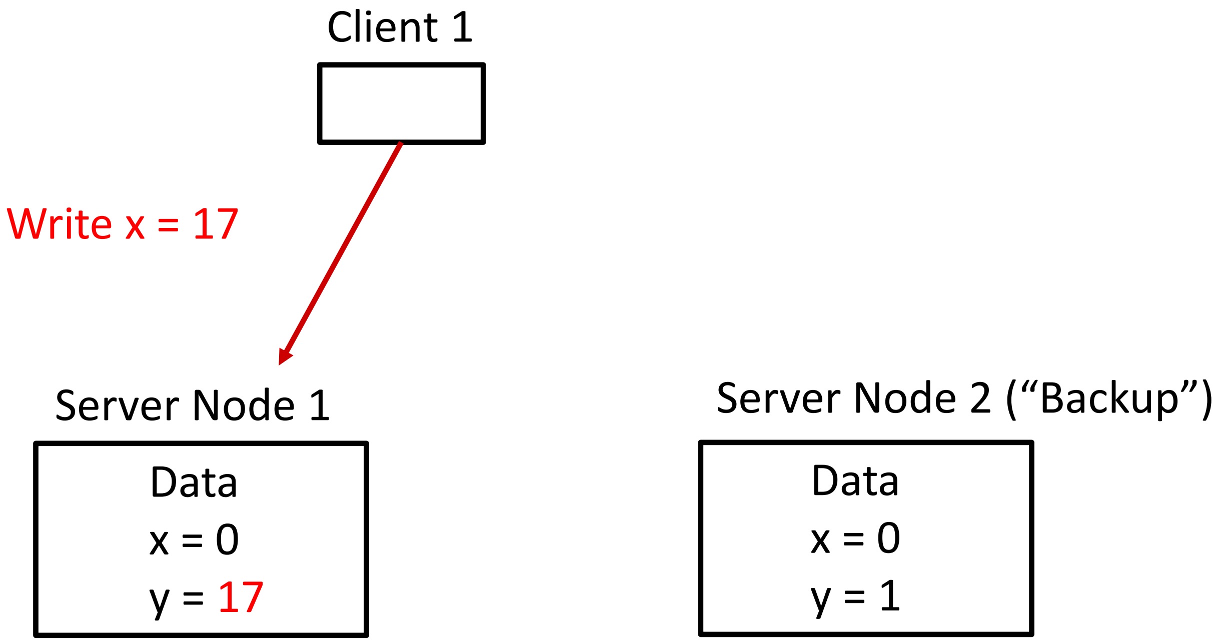
Server Node 2 (“Backup”)

Data
x = 0
y = 1

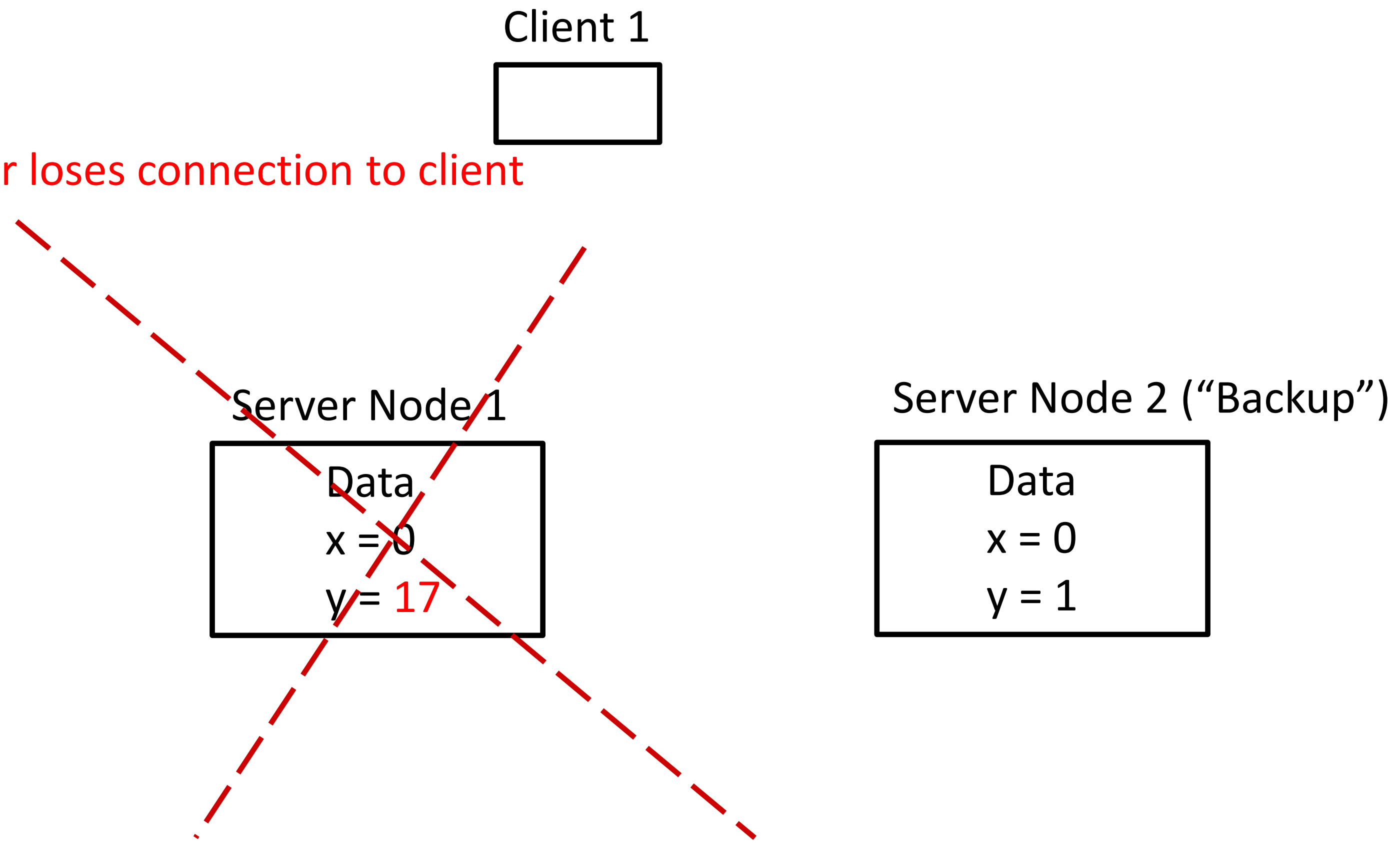
Can contact any node to
Read the data stored

What happens when writing
is involved?

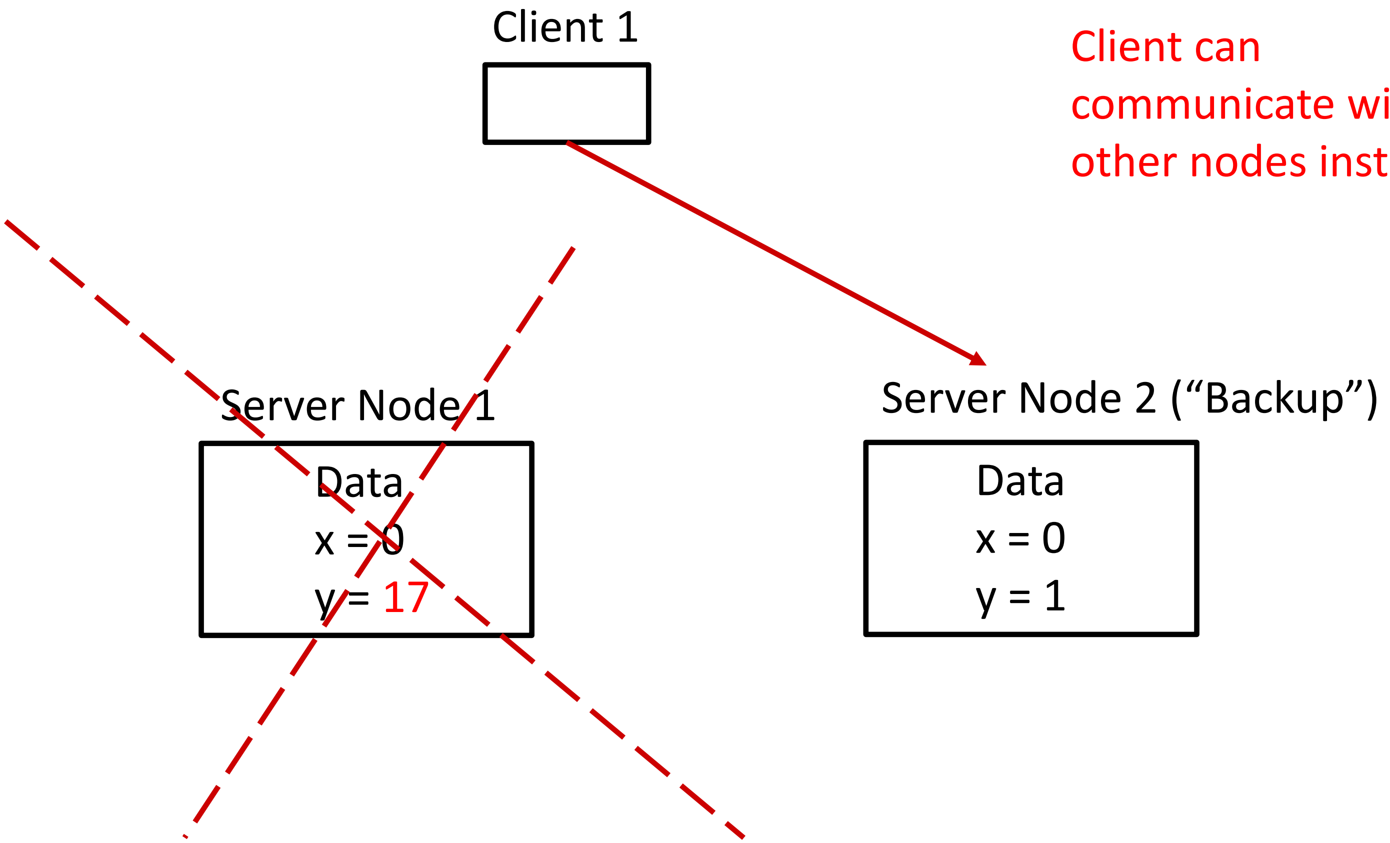
Example: Consistent State



Example: Consistent State

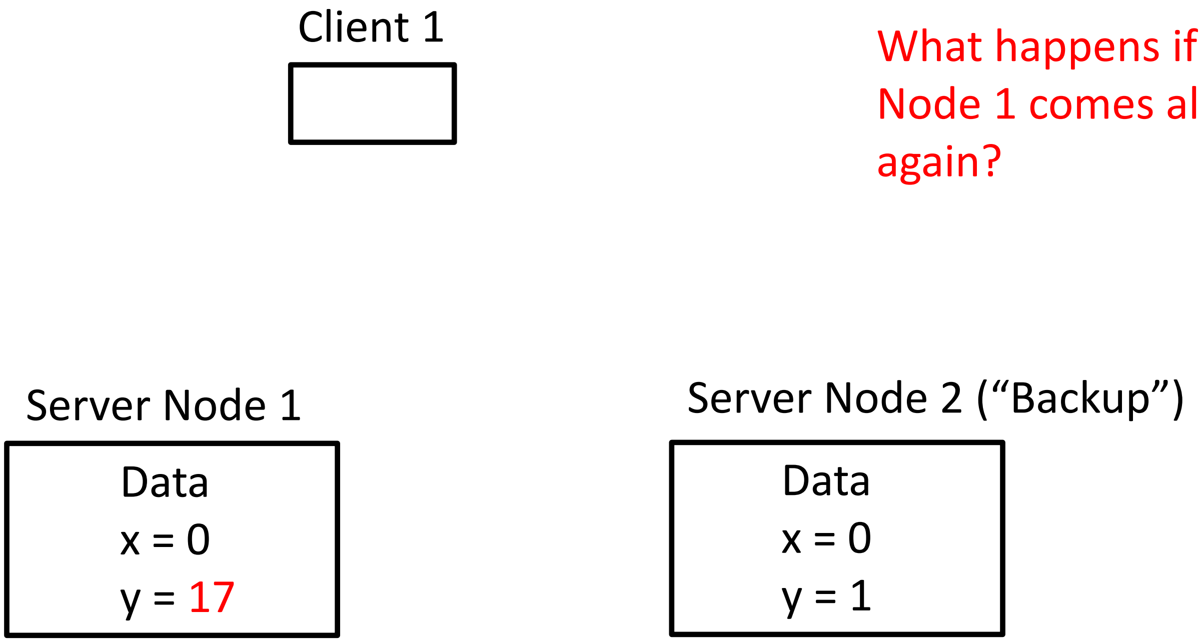


Example: Consistent State



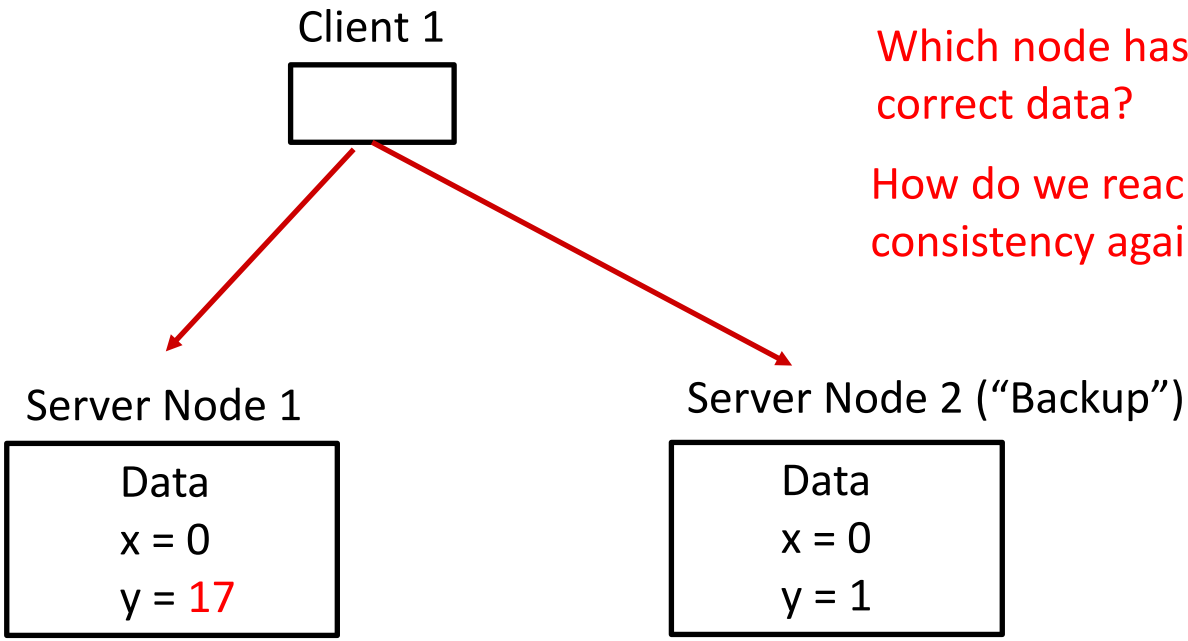
Client can communicate with other nodes instead

Example: Consistent State



What happens if Node 1 comes alive again?

Example: Consistent State



Which node has the correct data?

How do we reach consistency again?

PAXOS

- No deterministic fault-tolerant consensus protocol can guarantee progress in an asynchronous network.
- PAXOS is a protocol for solving consensus while being **resistant** to **unreliable** or **failable** processors in the system
  - Unreliable and failable could mean just that
    - the system crashes
    - packet (messages) are being sent and received inconsistently
    - Becomes malicious and behaves incorrectly “on purpose”
    - And in paxos, could possibly recover from any of these
- Paxos guarantees consistency, and the conditions that could prevent it from making progress are difficult to provoke.

Real Life Equivalents

- While what we went over aren’t “real” examples, these concepts apply to distributed systems.

- If a bank or database runs on a collection of nodes. How do we agree on whether a transaction occurred?
  - How do we ensure that the transaction went through and won’t get “lost” due to faults?

- What if data was split across different nodes and multiple clients needed data from multiple nodes at the same time?

Lecture Outline

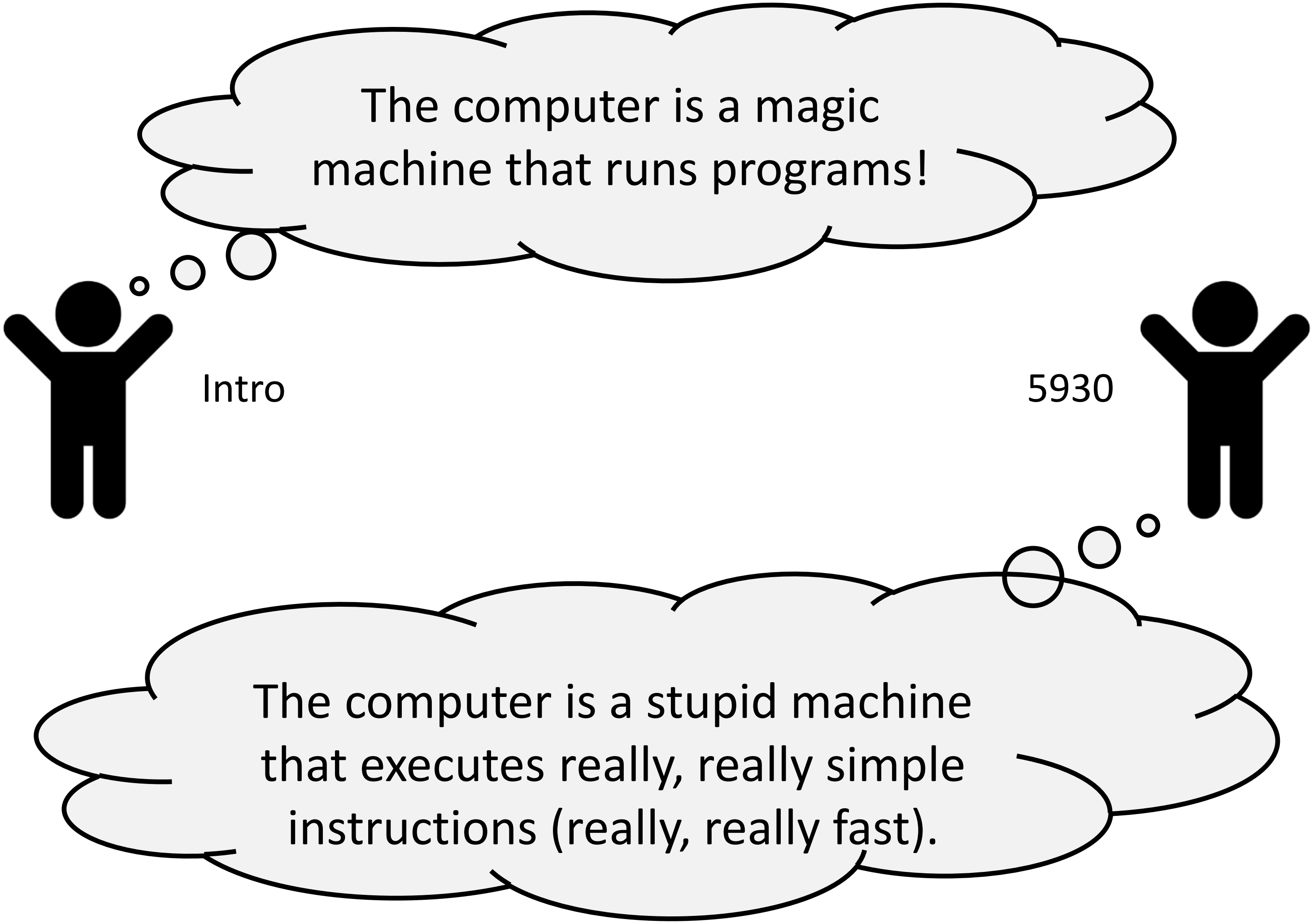
- Intro to Distributed Systems
- **Course wrap-up**

What have we been up to for the last 14 weeks?

- Ideally, you would have “learned” everything in this course, but we’ll use red stars ☆ today to highlight the ideas that we hope stick with you beyond this course

Course Goals

- Explore the gap between:



The computer is a magic machine that runs programs!

Intro

5930

The computer is a stupid machine that executes really, really simple instructions (really, really fast).

Systems Programming: The Why

- The programming skills, engineering discipline, and knowledge you need to build a system

  1) Understanding the "layer below" makes you a better programmer at the layer above
  2) Gain experience with working with and designing more complex "systems"
  3) Learning how to handle the unique challenges of low-level programming allows you to work directly with the countless "systems" that take advantage of it

So What is a System?

- "A **system** is a group of interacting or interrelated entities that form a unified whole. A system is delineated by its spatial and temporal boundaries, surrounded and influenced by its environment, described by its structure and purpose and expressed in its functioning."
  - https://en.wikipedia.org/wiki/System
  - Still vague, maybe still confusing

- But hopefully you have a better idea of what a system in CS is now
  - What kinds of systems have we seen…?

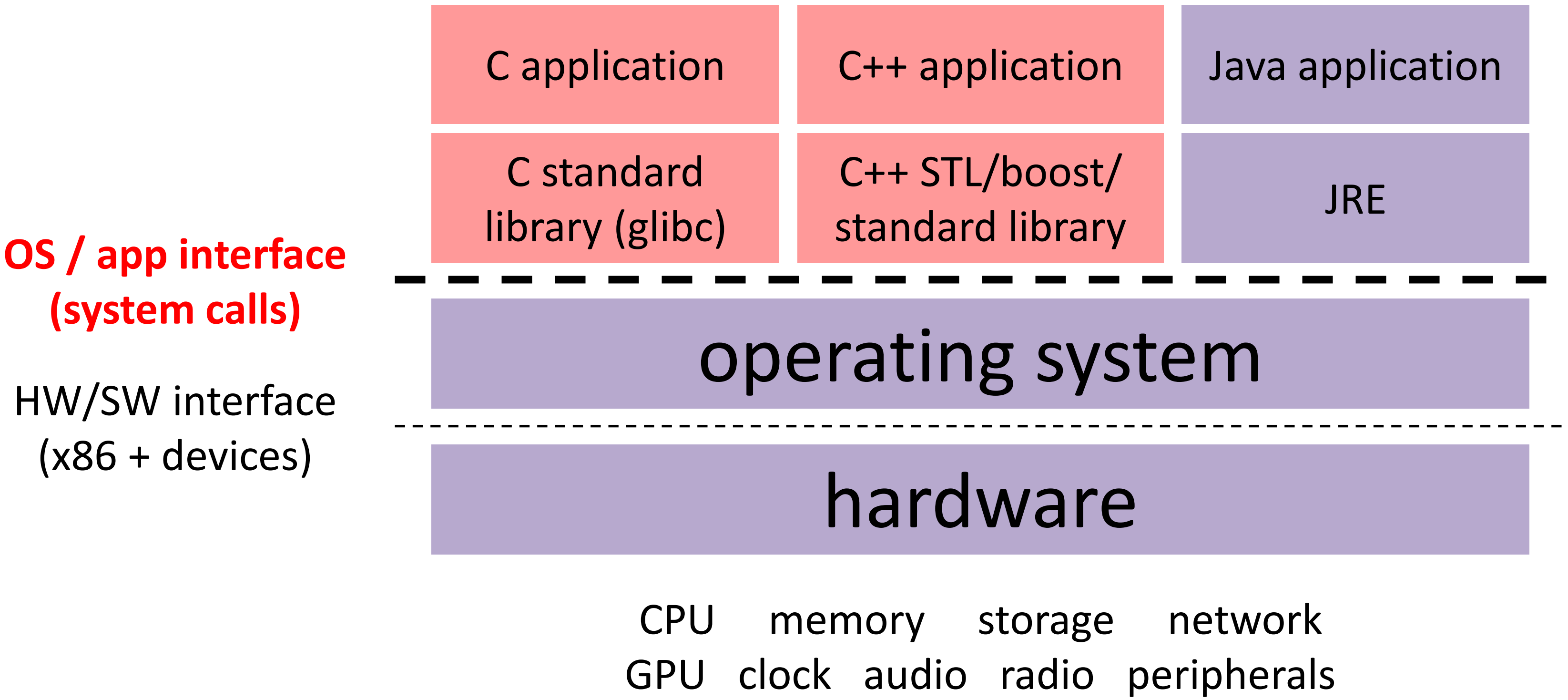
Software System

- Writing complex software systems is *difficult*!
  - Modularization and encapsulation of code
  - Resource management
  - Documentation and specification are critical
  - Robustness and error handling
  - Must be user-friendly and maintained (not write-once, read-never)

- **Discipline:** cultivate good habits, encourage clean code
  - Coding style conventions
  - Unit testing, code coverage testing, regression testing
  - Documentation (code comments, design docs)

The Computer as a System

- Modern computer systems are increasingly complex!
  - Networking, threads, processes, pipes, files
  - Buffered vs. unbuffered I/O, blocking calls, latency


C application
C++ application
Java application
C standard library (glibc)
C++ STL/boost/ standard library
JRE
OS / app interface (system calls)
operating system
HW/SW interface (x86 + devices)
hardware
CPU memory storage network
GPU clock audio radio peripherals

A Network as a System

- A networked system relies heavily on its connectivity
  - Depends on materials, physical distance, network topology, protocols
- Conceptual abstraction layers
  - Physical, data link, network, transport, session, presentation, application
  - Layered *protocol* model
    - We focused on IP (network), TCP (transport), and HTTP (application)
- Network addressing
  - MAC addresses, IP addresses (IPv4/IPv6), DNS (name servers)
- Routing
  - Layered packet payloads, security, and reliability

Systems Programming: The What

- The programming skills, engineering discipline, and knowledge you need to build a system
  - **Programming:** C & C++
  - **Discipline:** design, testing, debugging, performance analysis
  - **Knowledge:** long list of interesting topics
    - Concurrency, OS interfaces and semantics, techniques for consistent data management, distributed systems algorithms, …
    - Most important: a deep understanding of the "layer below"

Main Topics

- C
  - Low-level programming language
- C++
  - The 800-lb gorilla of programming languages
  - “better C” + classes + STL + smart pointers + …
- Memory management
- System interfaces and services
- Networking basics – TCP/IP, sockets, …
- Concurrency basics – POSIX threads, synchronization
- Multi-processing Basics – Fork, Pipe, Exec

Topic Theme: Abstraction

- C: `void*` as a generic data type
- C++: hide execution complexity
  - *e.g.*, operator overloading, dispatch, containers & algorithms
- C++: templates to generalize code
- OS: abstract away details of interacting with system resources via system call interface
- Networking: 7-layer OSI model hides details of lower layers
  - *e.g.*, DNS abtracts away IP addresses, IP addresses abstract away MAC addresses

Topic Theme: Using Memory

- Variables, scope, and lifetime
  - *Static*, *automatic*, and *dynamic* allocation / lifetime
  - C++ objects and destructors; C++ containers and copying
- Pointers and associated operators (`&`, `*`, `->`, `[]`)
  - Can be used to link data or fake "call-by-reference"
- Dynamic memory allocation
  - **`malloc`**/**`free`** (C), **`new`**/**`delete`** (C++), smart pointers (C++)
  - Who is responsible? Who owns the data? What happens when (not if) you mess this up? (dangling pointers, memory leaks, …)
- Tools
  - Debuggers (`gdb`), monitors (`valgrind`)
  - Most important tool: thinking!

Topic Theme: Data Passing

- C: output parameters
- C++: Copy constructors, and copy vs move semantics
- Threads: return values or shared memory/resources
  - Leads to synchronization concerns
- I/O to send and receive data from outside of your program (*e.g.*, disk/files, network, streams)
  - Linux/POSIX treats all I/O similarly
  - Takes a LONG time relative to other operations
  - Blocking vs. polling
- Buffers can be used to temporarily hold passed data
  - Buffering can be used to reduce costly I/O accesses, depending on access pattern. Similar thing for caches.

Topic Theme: Concurrency

- Processes
  - Exec
  - Process Groups
    - Terminal Control
  - IPC
    - Pipe
    - Signals
- Threads
  - Synchronization
    - mutex
    - Condition variables
  - Deadlock
- Concurrency vs parallelism

MISSING Topic Theme: Society

- One flaw (among others) of this course is how we don’t talk about how this relates to the rest of the world
  - These systems we build do not have to necessarily be “evil”, but can often be used in those ways
  - We need to work and communicate with other people, even in CS.

- Actions:
  - Take Algorithmic Justice (CIS 7000) with Danaë Metaxa
  - Join a community of people working on things that matter to you, (Unions or other organizations)
  - Join me as a TA for 2400 or 5950 next year. We will try to integrate ethics into those courses (still working out details).

Congratulations!

- Look how much we learned!

- Lots of effort and work, but lots of useful takeaways:
  - Debugging practice
  - Reading documentation
  - Tools (gdb, `valgrind`, `helgrind`)
  - C and C++ familiarity, including multithreaded and networked code

- Go forth and build cool systems!

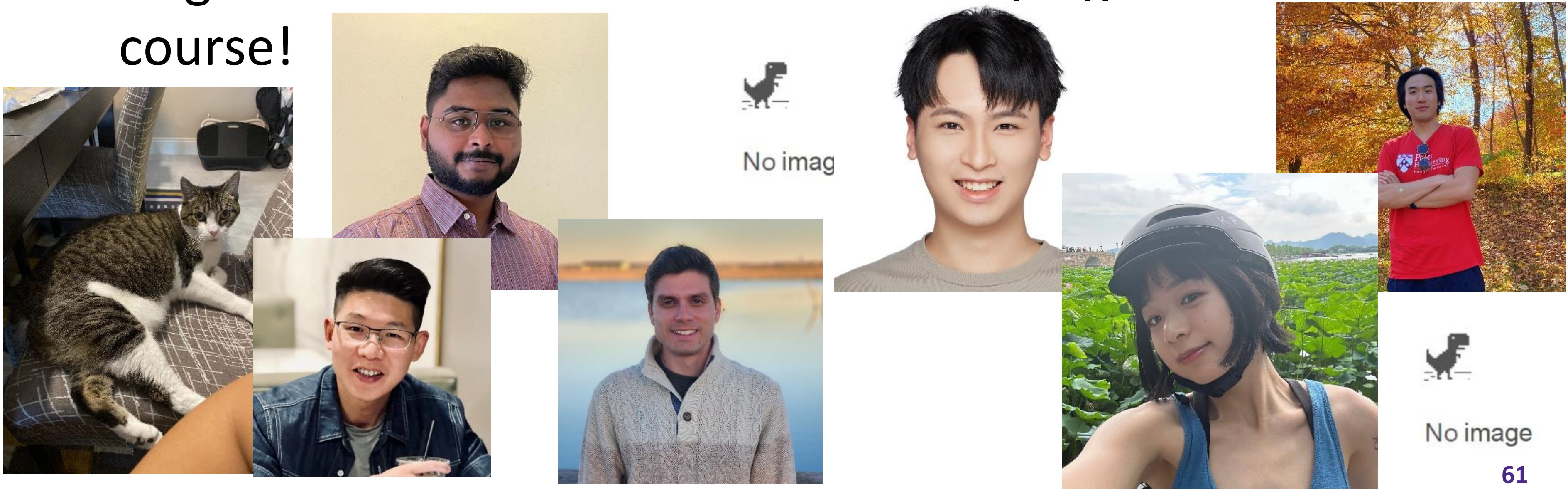
Future Courses

- Systems Courses
  - CIS 3410 Compilers (May have a grad version in the future)
  - CIS 5050: Software Systems
  - CIS 5480: Operating Systems Design and Implementation
  - CIS 5530: Networked Systems
  - CIS 5550 Internet and Web Systems
  - CIS 5500: Database and Information Systems
  - CIS 5470: Software Analysis
- Otherwise related courses
  - CIS 5600 Interactive Computer Graphics
  - CIS 5650 GPU Programming and Architecture
  - CIS 5570 Programming for the Web

Thanks for a great semester!

- Special thanks to all the instructors before me (Both at UPenn and UW) who have influenced me to make the course what it is

- Huge thanks to the course TA's for helping with the course!



Thanks for a great semester!

- Thanks to you!
  - It has been another tough semester. Still not completely out of the pandemic, Zoom fatigue, faltering motivation, etc
  - Relatively "new" version of the course. Many of the assignments and infrastructure are recently developed.
  - You've made it through so far, be proud that you've made it and what you've accomplished!

- **Please take care of yourselves, your friends, and your community**