Sockets & Distributed Systems Intro

Computer Operating Systems, Spring 2025

Instructors: Joel Ramirez Travis McGaha

Head TAs: Ash Fujiyama Emily Shen Maya Huizar

TAs:

Ahmed Abdellah	Bo Sun	Joy Liu	Susan Zhang	Zihao Zhou
Akash Kaukuntla	Connor Cummings	Khush Gupta	Vedansh Goenka	
Alexander Cho	Eric Zou	Kyrie Dowling	Vivi Li	
Alicia Sun	Haoyun Qin	Rafael Sakamoto	Yousef AlRabiah	
August Fu	Jonathan Hong	Sarah Zhang	Yu Cao	

Poll Everywhere

pollev.com/tqm

- What kind of lectures do you like?
  - What makes a lecture good for you?
  - (I know this is a biased sample but I still want to ask)

Administrivia

- PennOS
  - PennOS Final Demo posted
    - You will have to demo FAT here even if you pass the autograder.
  - Integration can be a big pain, make sure you allocate enough time to it!
  - Can use 1 late token for free now to submit by EOD Sunday
  - We will ask you short answer questions during the demo to check that you actually understand your code.
    - You will be able to choose the category of question.
    - More details on Ed

Administrivia

- Some notes:
  - **NEW**: If you are having issues with the scheduler sometimes not suspending a thread: make sure that you do NOT have interrupts **disabled** when you call spthread_create.
    - Or, you can redownload spthread.c from the course website
    - You do not HAVE to do either of these, though this will almost certainly cause issues.
  - DO NOT mmap the entire File System. Only mmap the Allocation Table, the rest of the file system needs to be handled with lseek/write.
    - Do not keep the contents of the file in memory, it should be stored in the file
    - If your PennFat is killed with kill -9, your file contents should still be saved in disk
  - Advice for using gdb to debug
    - **`Handle SIGUSR1 noprint nostop`**
      Makes it so that gdb doesn't report every time SIGUSR1 goes and interrupts you

Administrivia

- Some notes:
  - Reminder, you instead of just doing:

```
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
```

  you may need to do:

```
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
lseek(FAT_FD, offset, SEEK_SET);
write(FAT_FD, contents, size);
```

  - With the description of `setitimer()`, it just says that sigalarm is delivered to the process, not necessarily the calling thread. To make sure siglaram goes to the scheduler, you may want to make it so that all threads (spthread or otherwise) that aren't the scheduler call something like: **`pthread_sigmask(SIG_BLOCK, SIGALARM)`**
    - Which will block SIGALARM in that thread.

Administrivia

- If you are having issues with the scheduler not running you can try running
  - **`strace -e 'trace=!all' ./bin/pennos`**
  - You may have to install strace: **`sudo apt install strace`**
  - This will print out every time a signal is sent to your pennos
  - (Usual fix is the pthread_sigmask thing on the previous slide)

Poll Everywhere

pollev.com/tqm

- Any questions on PennOS?

Lecture Outline

- **Sockets**
  - **High Level Socket**
  - **How different from Files**
- Distributed Systems
  - Muddy Forehead
  - Distributed String
  - Performance

The Sockets API

- Berkeley sockets originated in 4.2BSD Unix (1983)
  - It is the standard API for network programming
    - Available on most OSs
  - Written in C

    Can still use these in C++ code
    You'll see some C-idioms and design practices.
- POSIX Socket API
  - A slight update of the Berkeley sockets API
    - A few functions were deprecated or replaced
    - Better support for multi-threading was added

Socket

- A Socket is an endpoint for a specific connection
  - If we think of a connection like a wire, then it must “plug in” to each end of the connection. Sort of like how you plug a charger into an outlet/wall socket
- A connection is identified by four things:
  - Client IP address
  - Client Port Number
  - Server IP Address
  - Server Port Number
- Going back to our apartment and post office analogy. For real packages we don’t just put an address and apartment number of the destination, we also include the address it came from.

Files and File Descriptors

- Remember `open()`, `read()`, `write()`, and `close()`?
  - POSIX system calls for interacting with files
  - `open()` returns a file descriptor
    - An integer that represents an open file
    - This file descriptor is then passed to `read()`, `write()`, and `close()`
  - Inside the OS, the file descriptor is used to index into a table that keeps track of any OS-level state associated with the file, such as the file position

Parameters to

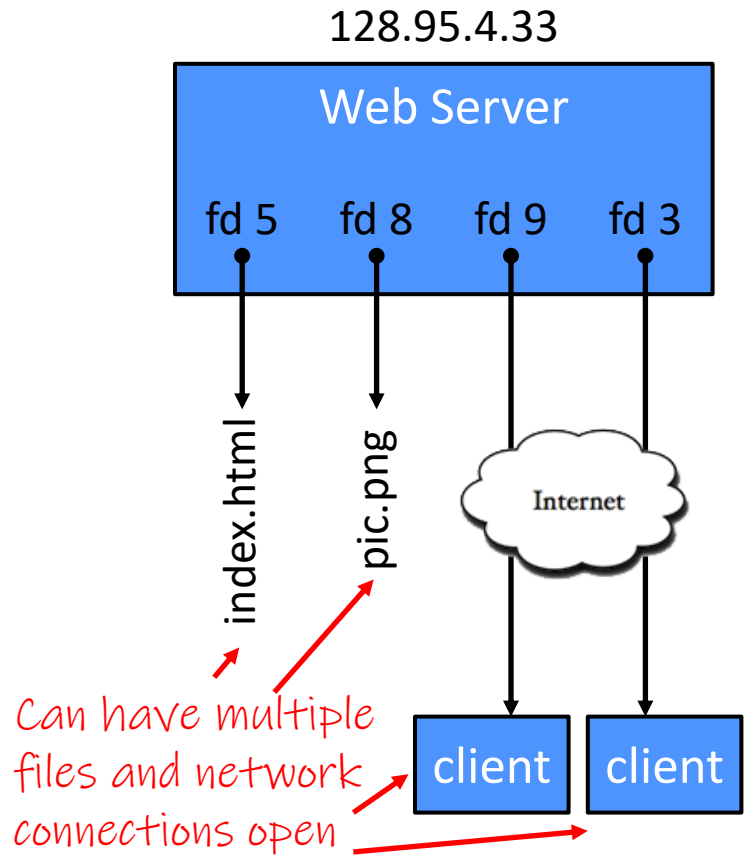
Can't be a pointer, don't want to give address to kernel

Networks and Sockets

- UNIX likes to make *all* I/O look like file I/O
  - You use **`read`**() and **`write`**() to communicate with remote computers over the network!
  - A file descriptor use for network communications is called a socket
  - Just like with files:
    - Your program can have multiple network channels open at once
    - You need to pass a file descriptor to **`read`**() and **`write`**() to let the OS know which network channel to use

In other words, we specify the socket to read/write on

File Descriptor Table



OS's File Descriptor Table for the Process

File Descriptor	Type	Connection
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

0,1,2 always start as stdin, stdout & stderr.

Types of Sockets

- **Stream sockets** *What we will focus on*
  - For connection-oriented, point-to-point, reliable byte streams
    - Using TCP, SCTP, or other stream transports
- **Datagram sockets**
  - For connection-less, one-to-many, unreliable packets
    - Using UDP or other packet transports
- **Raw sockets**
  - For layer-3 communication (raw IP packet manipulation)

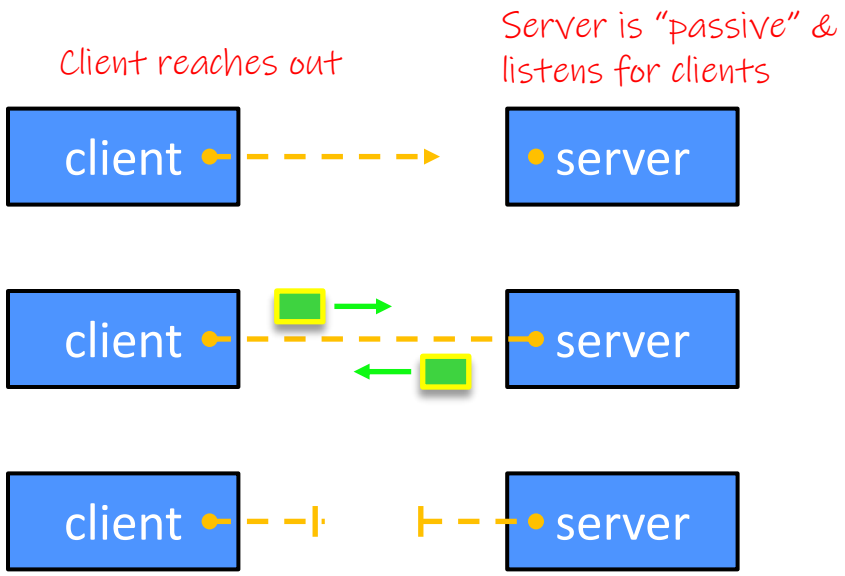
Stream Sockets

- Typically used for client-server communications
  - Client: An application that establishes a connection to a server
  - Server: An application that receives connections from clients
  - Can also be used for other forms of communication like peer-to-peer

1) Establish connection:

2) Communicate:

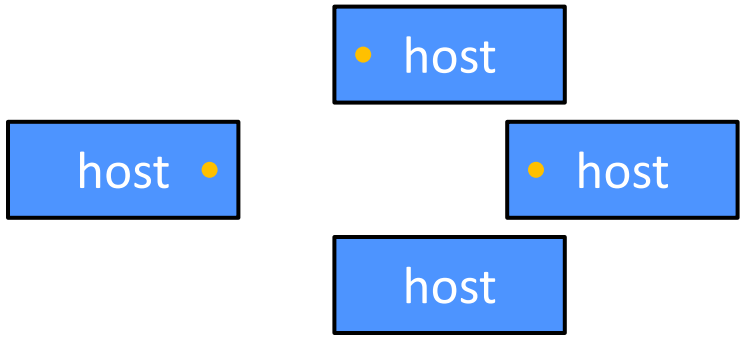
3) Close connection:



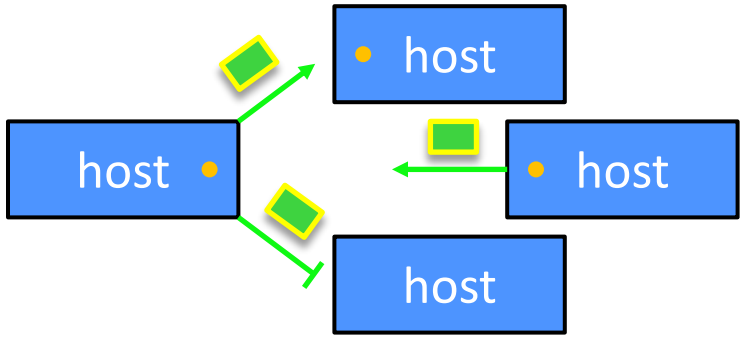
Datagram Sockets

- Often used as a building block
  - No flow control, ordering, or reliability, so used less frequently
  - *e.g.* streaming media applications or DNS lookups

1) Create sockets:



2) Communicate:



Demo

- sendreceive.cpp

- udp_receive.cpp
- udp_send.cpp

Sockets are sort of like files

- When dealing with stream sockets (TCP) Sockets, the TCP part is done for us. We can deal with the stream ABSTRACTION
  - Stream: That the bytes show up in order reliably

pollev.com/tqm

- How do you think a network connection may behave differently from a file?
  - If it helps you can compare a file to reading/writing into a book and reading/writing a socket to texting/messaging a friend.

`read()`

- If there is data that has already been received by the network stack, then read will return immediately with it
  - `read()` might return with *less* data than you asked for

- If there is no data waiting for you, by default `read()` will *block* until something arrives **pollev.com/tqm**
  - How might this cause *deadlock*?
  - Can `read()` return 0? (EOF)

Lecture Outline

- Sockets
  - High Level Socket
  - How different from Files
- **Distributed Systems**
  - **Muddy Forehead**
  - **Distributed String**
  - **Performance**

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 hardware (e.g. memory) other than the network
  - Individual computers (nodes) can fail
  - The network itself can fail (Drop messages, corrupt messages, delay messages, etc.)

Why do we care?

- They are really interesting problem to work with

- Most applications we interact with are distributed systems of some sort:

Why do we care?

- They are really interesting problem to work with
- Distributed systems typically allow a system to scale well. Need more work to be done? Just add a new computer to the system
- Distributed systems can also allow for some amount of “fault tolerance”. If one computer crashes, the rest of the computers will probably keep running.

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

Shared Nothing Architecture

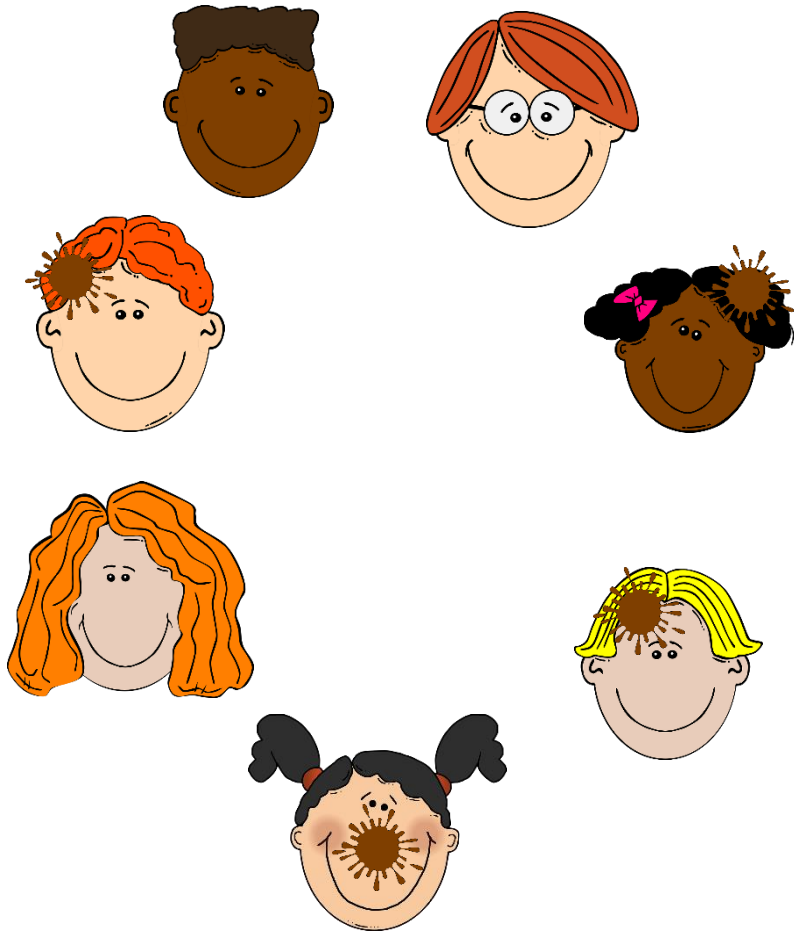
- Consistency and sharing data is hard in a threaded program (as you may see with PennOS, though PennOS is especially weird)

- What about distributed systems?
  - Distributed systems are typically “Shared nothing” meaning that it is a collection of computers communicating over the network
  - There is no shared memory
  - There is no shared disk/storage

- How can we get a cluster (group of machines) to agree on some state?
  - How do the computers in the system reason about each other?

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."
  - Children cannot feel the mud on their head.
  - Children cannot speak
  - 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."
  - Children cannot feel the mud on their head.
  - Children cannot speak
  - What happens?
    - The answer is not "no one raises their hand"

Muddy Foreheads

- **Answer**: On the kth round, all of the muddy children know they have mud on their forehead, raise their hands

- "**Proof" by induction on k**. When k=1, the muddy child knows no other child is muddy, must be muddy themself. When k=2, on the first round, both muddy children see each other, cannot conclude they themselves are muddy. But after neither raises their hand, they realize there must be two muddy children, raise their hand.

- In general, when k>1, after round k-1, if there were k-1 muddy foreheads, all of those children would have raised their hands (by induction). Therefore, each muddy child knows they're muddy and raises their hand on the kth round

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?

Muddy Forehead Takeaways

- Why did we talk about this?
  - **Cause I think its fun ☺**
  - Helps us think about how there are many concurrent “rational agents” all working “as equals”
    - There isn’t a head coordinator amongst the children, they are all equals
  - The children don’t have a shared state (memory or disk). In fact, they barely have a working "network"! (All they can do is see the other children)

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

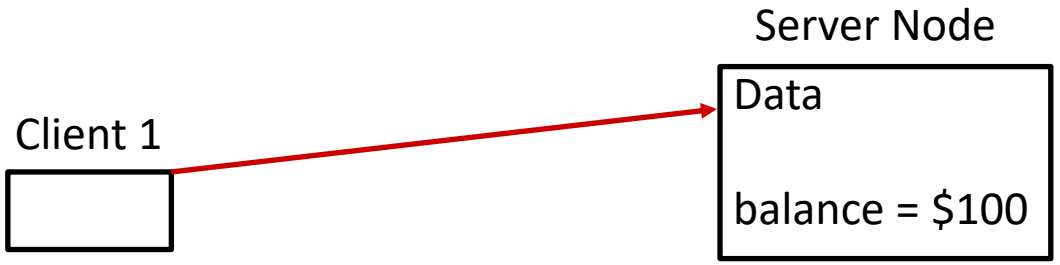
- Even in this simple, example, issues can arise

Example: RPC

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

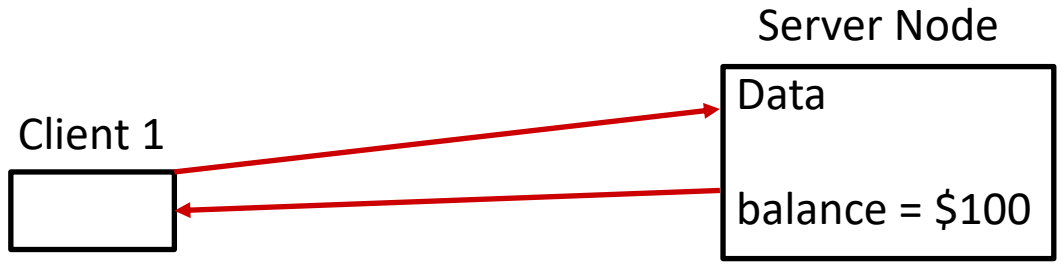
Example: RPC

- 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



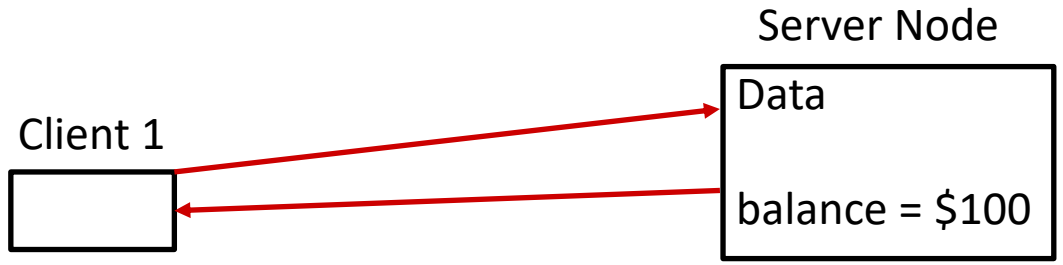
Example: RPC

- 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()"



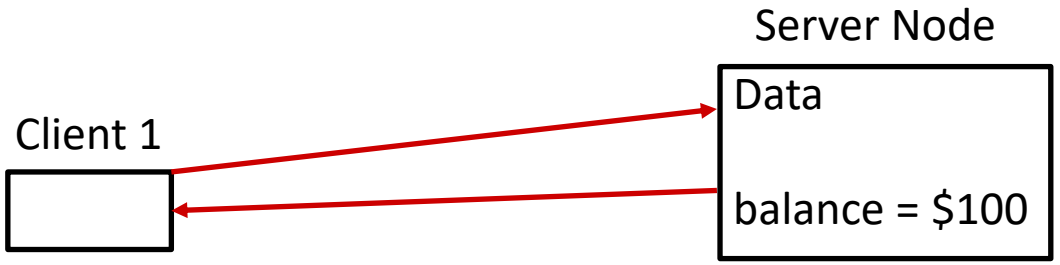
Client was blocked while waiting for the server to respond.

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

Blank Slide

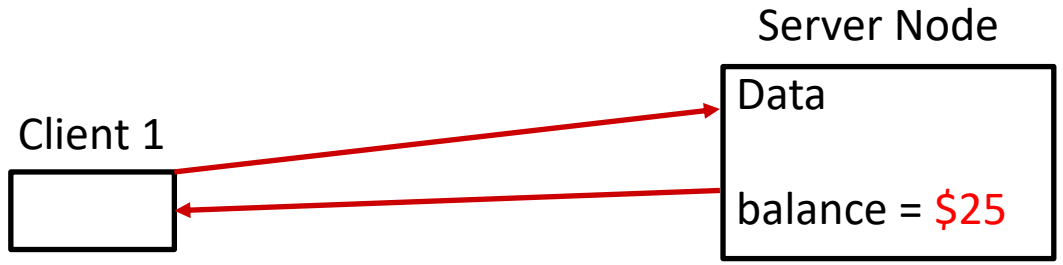
Example: RPC Transaction

- 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



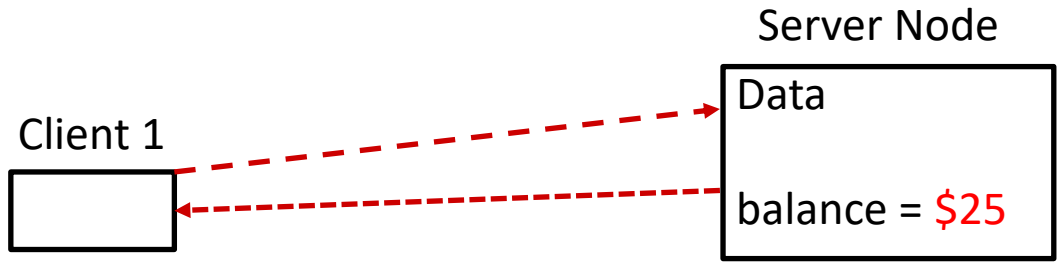
Example: RPC Transaction

- 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



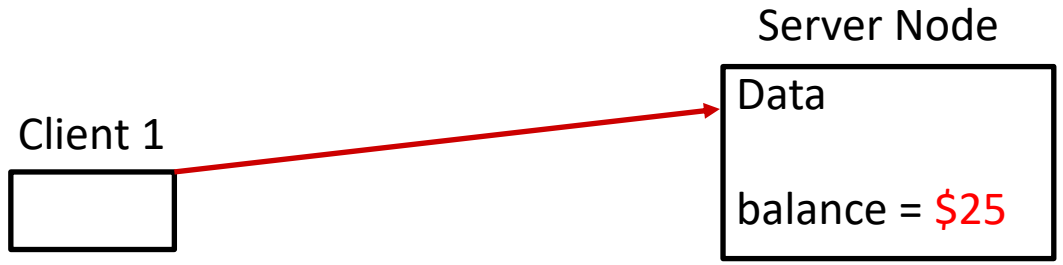
Example: RPC Transaction

- 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!



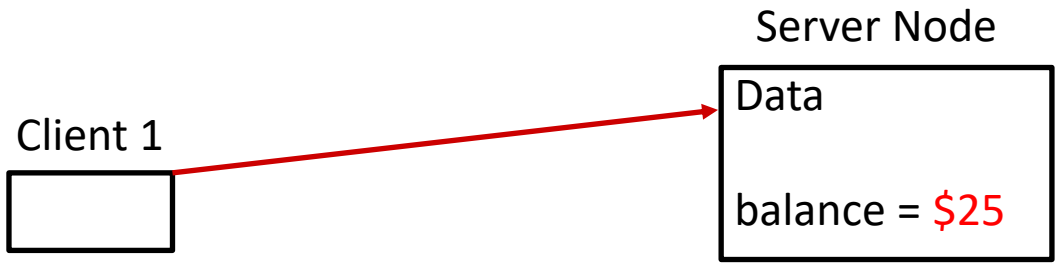
Example: RPC Transaction

- 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)"…



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"
    - Is this a new request? Is this a repeat request?



Fix?

- How do we ensure that each transaction is done exactly once?
  - Thoughts?

Fix?

- How do we ensure that each transaction is done exactly once?
  - Thoughts?

- We can have a “logical timestamp” (sometimes called a logical clock). We can have some sort of counter to identify the action taken.

- If the server receives these two messages

```
action_id = 5;
action = withdraw(75);
```

```
action_id = 6;
action = withdraw(75);
```

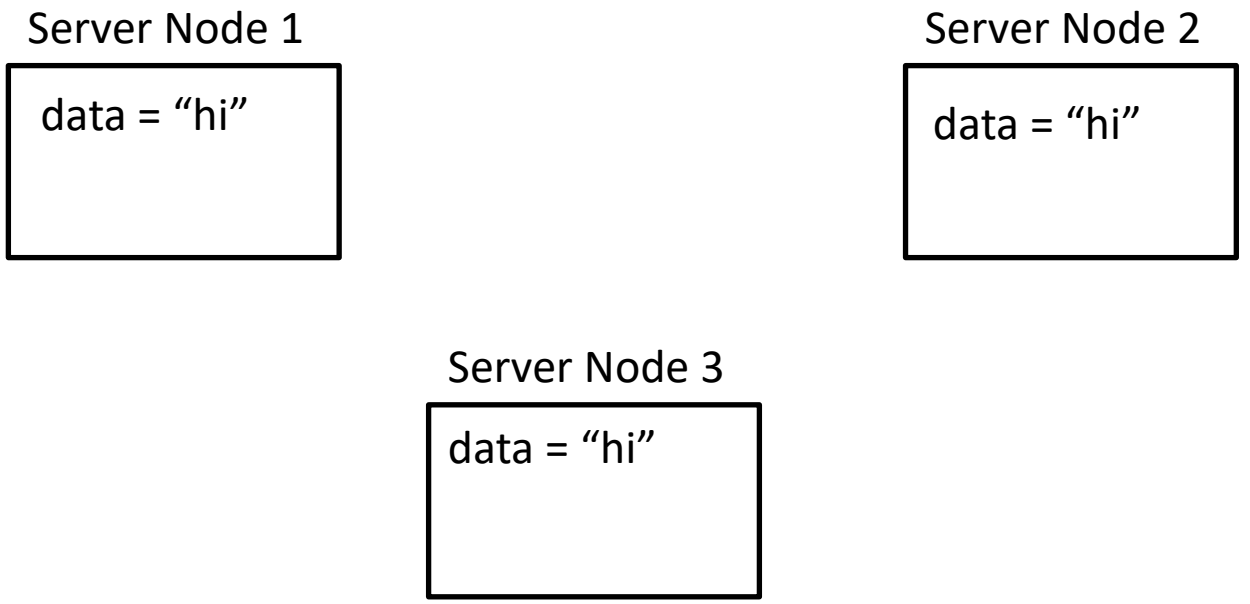
it means something different than receiving these two

```
action_id = 5;
action = withdraw(75);
```

```
action_id = 5;
action = withdraw(75);
```

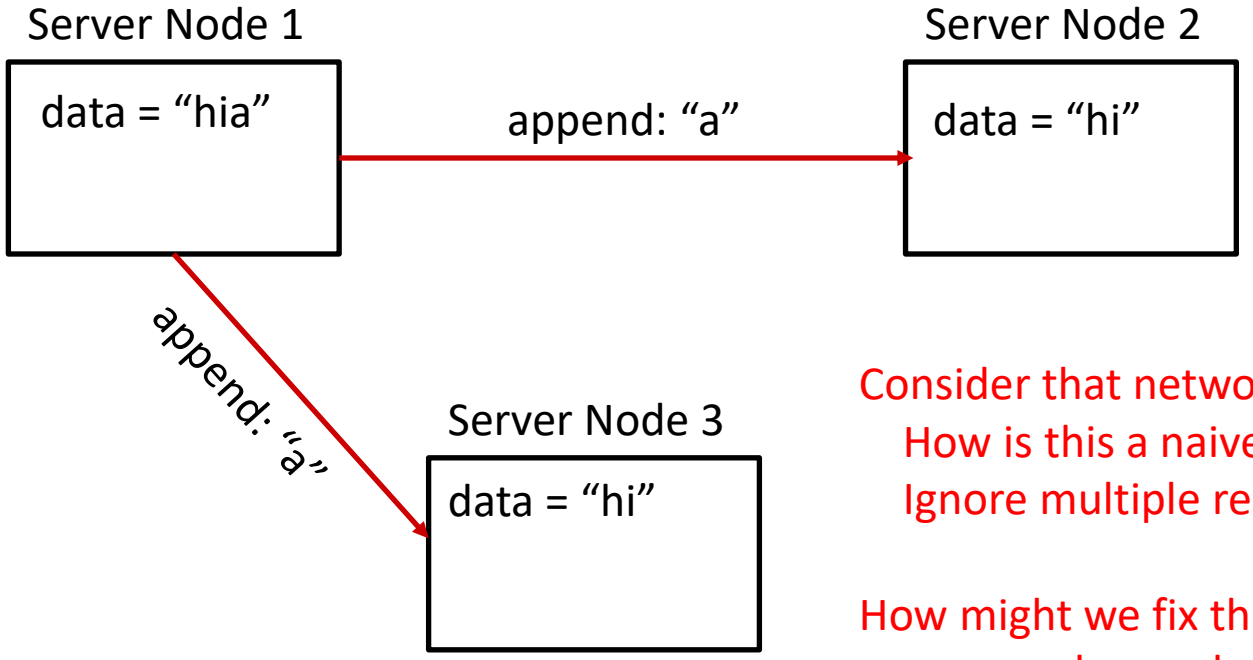
Shared State in Distributed Systems

- Let's say we have a collection of computers that together share the state of a single string.
- A client can connect to any node and submit a command
  - E.g. append, get, set, etc.



Shared State in Distributed Systems: Network Delays

- Let's say a client connects to node 1 and sends a request to append "a" to the end of the string.
- Node 1 then does the action then forwards the request to other nodes so that we can maintain a consistent state across all nodes.

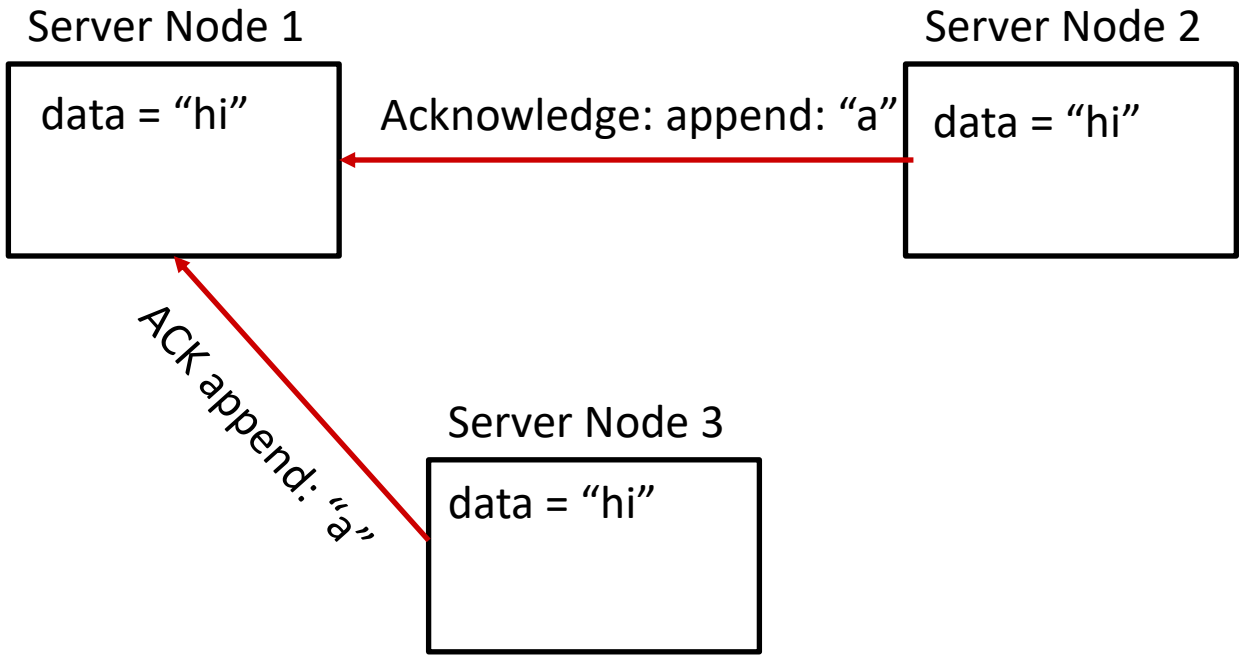


Consider that network messages may be delayed/lost
How is this a naive implementation?
Ignore multiple requests for now

How might we fix this issues?
we can always change the contents of the message or send more messages. How might we do that?

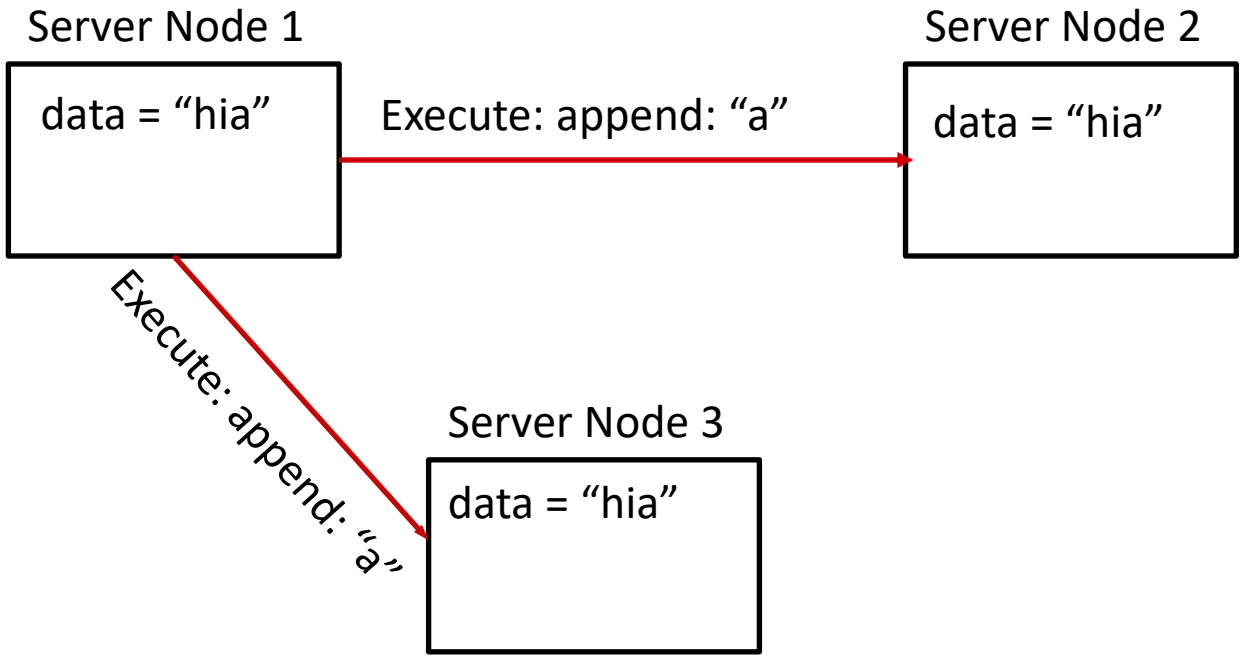
Shared State in Distributed Systems: Network Delays

- Fix part 1: Don't execute action yet. Have other nodes send an acknowledgement back. If the node doesn't send an ack, then we re-send the command to them intermittently until they do ack it.



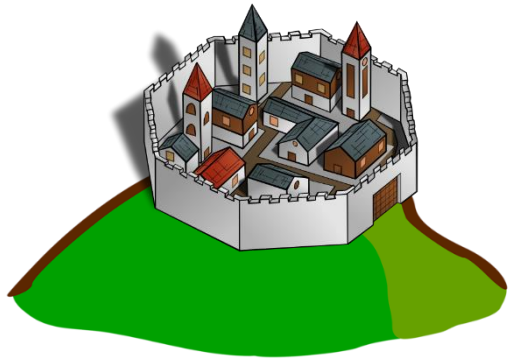
Shared State in Distributed Systems: Network Delays

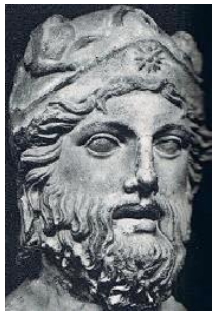
- Fix part 2: Once all nodes have acknowledged a request, the node sends a "execute" command to let all nodes know that everyone reached a consensus.



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.





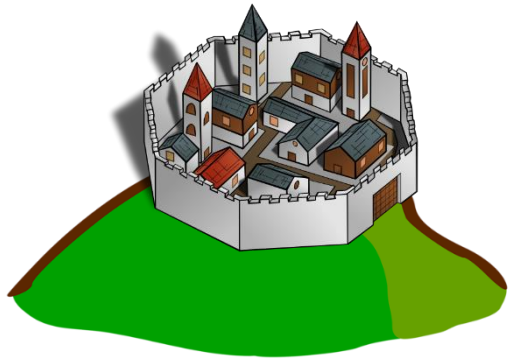
- How do they ensure they can take the city?

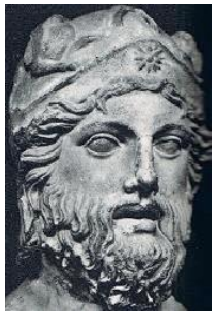
Coordinated Attack

- **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

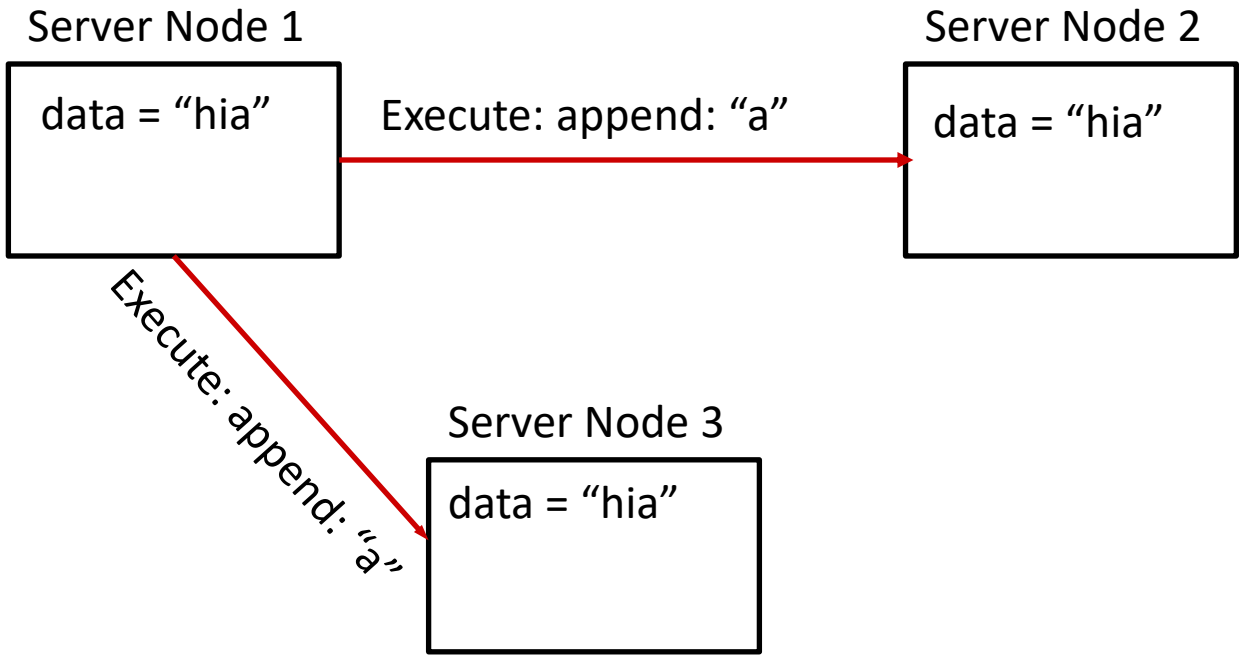
- To coordinate an attack, the problem requires common knowledge
- With the messengers, common knowledge is never reached.
- What happens when we add more generals?
- What happens when some of the generals are malicious?





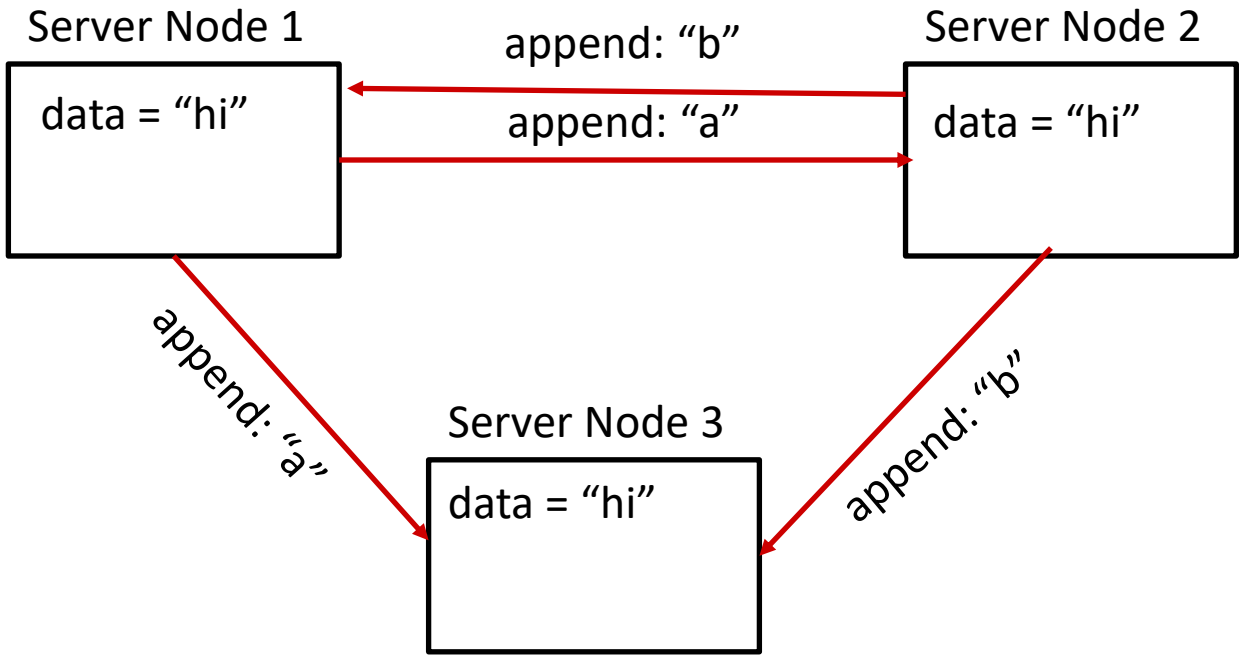
Shared State in Distributed Systems: Network Delays

- Going back to our action / acknowledge / execute plan
  - Does this work in all cases?



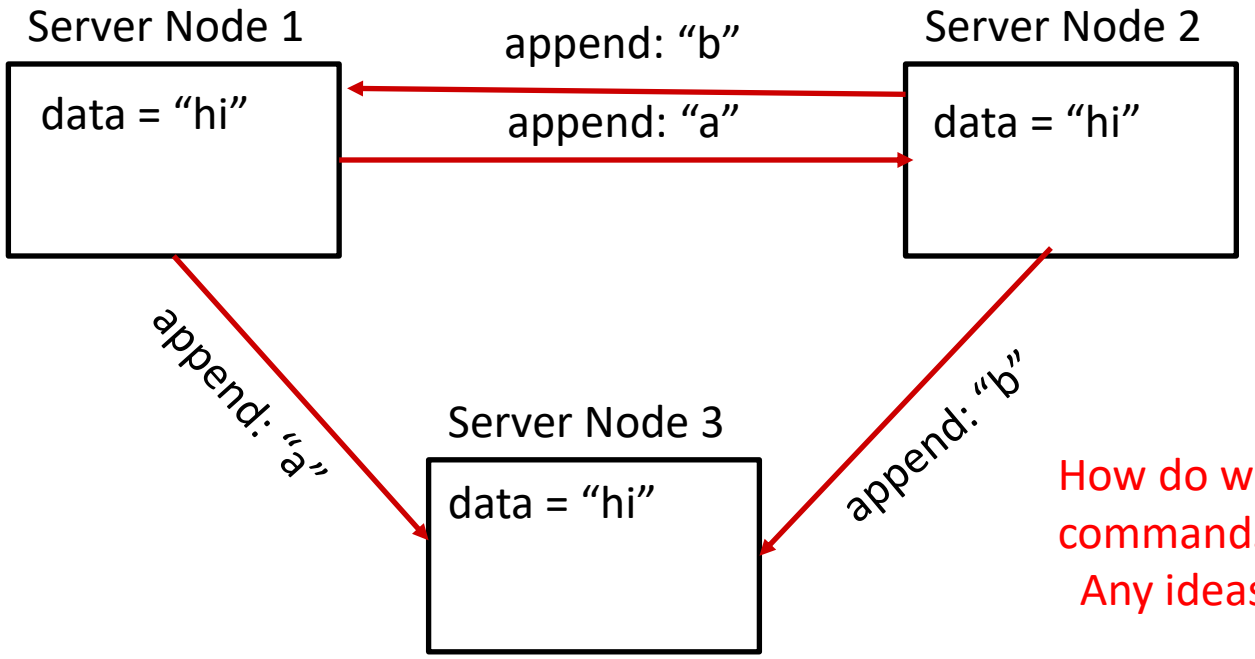
Shared State in Distributed Systems: Concurrent Requests

- Let's say that we have two clients:
  - One connects to node 1 and wants to append a
  - Another client connects to node 2 wants to append b



Shared State in Distributed Systems: Concurrent Requests

- Let's say that we have two clients:
  - One connects to node 1 and wants to append a
  - Another client connects to node 2 wants to append b



How do we maintain an ordering among the commands so that we know which to execute first?
Any ideas?

A lot more complicated……
include some amount of "logical timestamps"

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
    - Nodes become malicious and behaves incorrectly “on purpose”
    - Paxos could possibly recover from any of these (to some amount)
    -
- Paxos guarantees consistency, and the conditions that could prevent it from making progress are difficult to provoke.

Performance

- Taking a step back from fault tolerance
- Another concern with doing actions across a distributed system is trying to make efficient utilization of the nodes in the system

- If we have a large task, how do we split up the work roughly evenly across nodes in the network so that it is completed faster?
  - Avoid having one “coordinator” node if possible
    - Then nodes may have to wait for the coordinator to tell them what to do and there is less coordinators)
  - Try to treat the nodes equally like rational actors so that they can all do work at the same time.

An Interview Question:

- 100 Nodes in a cluster of computers
- Each Node is numbered 0 through 99
- Each node has 1,000,000 integers
  - No node can hold all the integers, but you can assume each node can hold more than 1,000,000.
- We want to sort all the numbers so that node 0 contains the first **1%** of the integers in sorted order (the lowest million integers). Node 1 contains the next million lowest integers, etc.

- How do we do this efficiently?
  - Afterwards: How can we make this algorithm minimally dependent on the original contents of the nodes.

An Interview Answer

- Answer:
  - Have each node sort their 1,000,000 integers
  - Send the bottom 1% to the node 0, the next 1% to node 1, etc. etc.
  - This should get you most of the way there. From there you can have each node sort their internal list and “bubble sort” values up and down the nodes as needed.

- This works well in the “general” case where the nodes aren’t sorted
  - What happens if the nodes were already sorted tho?
  - Can we assume that each node’s data is representative of the whole set?

  - Answer: fix this by making each node first randomly distribute their integers. Then each node will have a random sample of integers to work with that should be roughly representative of the whole data.

This was just an “intro” to the field ☺

- Lots of details left out, but 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?