CS262 Engineering Notebook

Pratyush Mallick | Anika Lakhani

Programming Assignment 3

Understanding the Specs

To be honest, it took us a good week to figure out what the specifications even meant. Eventually, we realized that they were as follows:

1) **What happens if a client loses connection?** *Implement logic so that messages during a given session can be recovered. AKA create persistence by saving message info in a database.*

2) **What happens if the hosting server loses connection?** *Create backup servers called ‘replicas’ that are ready to take over when a main server goes down.*

Trying a UI – Week 1

We wasted a good week trying to create a better chat UI because we thought it would be easier to represent changes in the network with a better visual experience for the user. We created:

1. A buggy React app that we scrapped for:
2. A simpler JS/Python/HTML/CSS combo that it became very confusing to implement server/client logic in

We weren’t getting anywhere with this, so we ditched the idea in the interest of time.

SQL Implementation – Week 2

We then moved on to creating a SQL implementation to work into our existing code. We wanted to create persistence of data by using SQLite to store message info that could be retrieved at a later date if desired.

We quickly realized that, with our current baseline of code, threading made it very difficult to properly append outgoing message info to a database. We eventually created a working version, but we weren’t proud of our baseline code that it was built atop of.

Aside: Creating a Session ID

By putting ourselves in the shoes of the end user, we assumed that end user would want to replay messages from a given session ID if their connection was cut during the session and they were rejoining the chat.

So, we created a method to keep track of session ID each time our server (we only had one at the time) started up. This way, we could tag each message with the session ID it was sent within. We created two new options on our main(): 1) Re-Generate Chat and 2) Replay Chat.

Our New Main Menu Offerings

6. Re-Generate Chat

This function would query our server’s SQL database for all messages sent within the current session ID (at the time, we only had one server and one database).

Then, it would print each message with the following format:

[username]: ‘[message]’; sent at [timestamp]

7. Replay Chat

This function would perform the same backend operations as option 6, but it would use waits in between printing each message so it felt like the user was watching a video replay of the chat session.

Messages were printed in this format:

[username]: ‘[message]’

Our Code Baseline

We had several issues starting this project because we realized that we had versioning issues and, somehow, new bugs since when we last touched the baseline code.

After creating the working SQL implementation, we abandoned the current code in order to give our baseline code a major facelift.

We switched from focusing on group chats to focusing on DMs (direct messages). We would re-implement this SQL work at a later time after prioritizing other TODO items.

Fault Tolerance Basics

Byzantine FT: Requires 2t+1 replicas

Failstop FT: Requires t+1 replicas

We learned that, in order to prove 2-fault tolerance for failstop faults, we would need 3 operating servers. This is because:

- 1 server goes faulty
- 2 servers hopefully remain safe (1-FT for Byzantine)
- The ‘healthy’ servers outnumber the faulty server
- If you want to plan for 2 consecutive failures, you need 3 total servers… assuming that the servers behave as requested
- If they don’t behave as requested, you have byzantine faults and would need 5 servers for 2-FT!

Choosing a Protocol

We first opted for a **Paxos protocol** because this protocol seemed the most robust and secure, and we liked it better than the **RAFT algorithm.**

We liked that Paxos had a Python library because we thought this library would greatly speed up our code.

However, Paxos is quite complicated to understand, and the most important aspect of a protocol choice for us was being able to profoundly understand it.

Our classmate, Michael, encouraged us to look further into the **secondary replication model**. We also saw people on Ed questioning using Paxos.

Secondary replication was much easier to understand, and it seemed to work better with our persistence mechanism of using a SQL database for each server.

We decided to go with secondary replication and haven’t looked back since!

Our (First) Checklist Until Completion!

Now that we really had our bearings and felt ready to tackle fault tolerance, we made a roadmap for our work:

1. Re-implement our old server session ID logic in our new (facelifted) baseline code

2. Re-implement our database insertion logic within our facelifted code so we can insert new entries into the database every time a message is sent

Our (First) Checklist Until Completion! (Continued)

3. Recreate the re-generate and replay chat functions we had in the old code

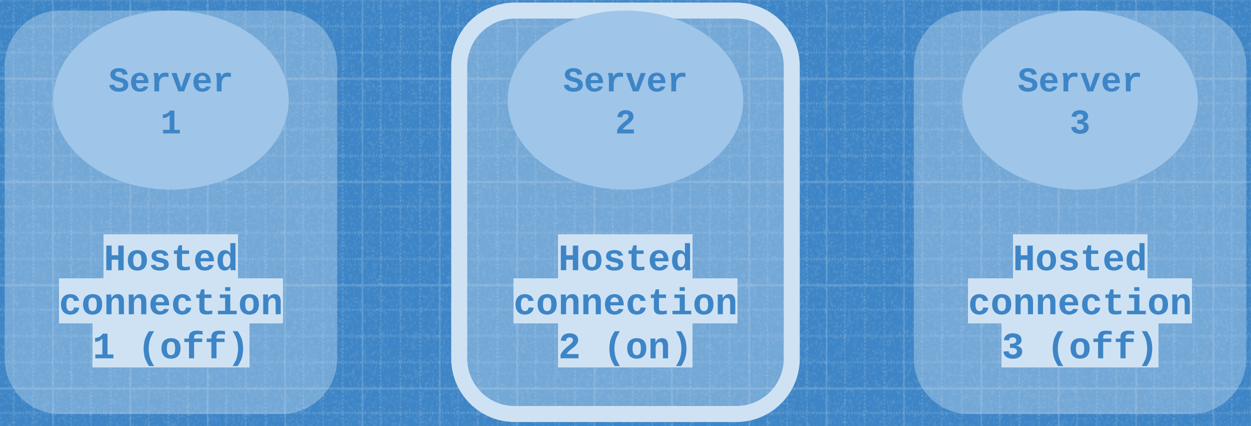
4. Create client-side logs (perhaps store this by creating a table for each username and appending rows to that user’s table) as well as a master log for fault tolerance

As you can see, we were operating under some misconceptions about how data was going to be stored and shared in the distributed network. We ended up revising this checklist as we realized more about work dependencies and secondary replication.

Switching the Client's Connection

We created some client-side code so that a client tries a list of IP/port combos when looking to connect to a server and connects to the first combo that works.

We tested this by creating a few servers, only running one of them, and making sure that the client was able to join that server since it was the only available one.



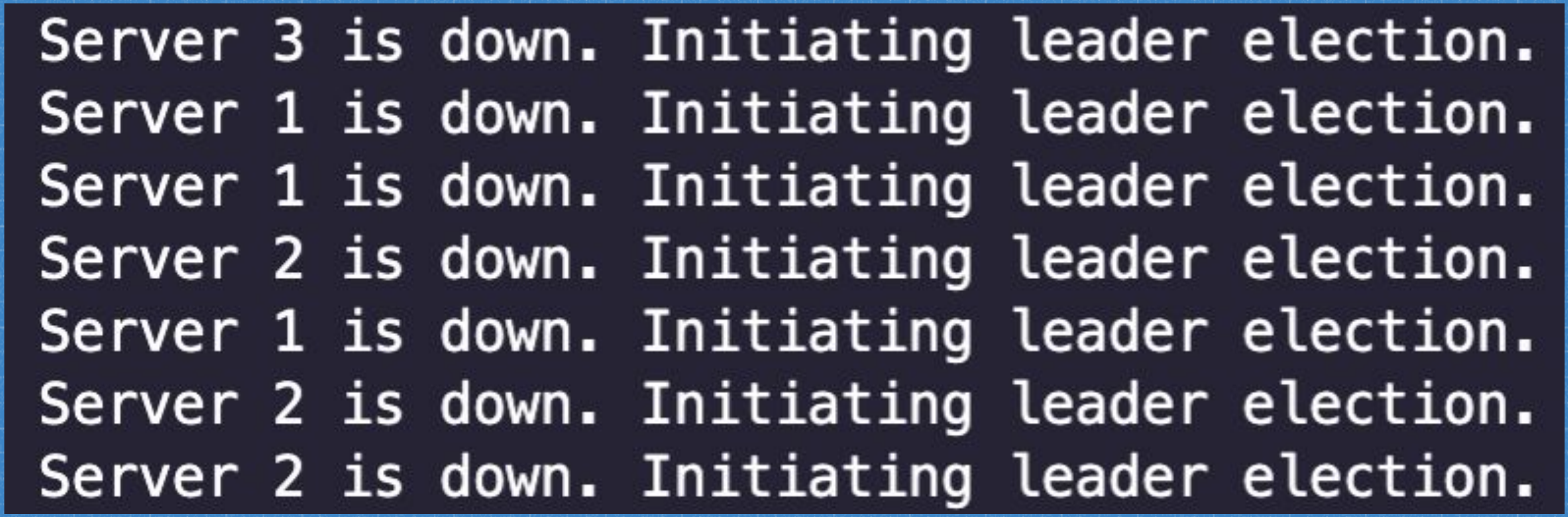
The Heartbeat Mechanism

Now that we had our first logic that allowed a user to connect to a working connection chosen from a list of connections, we needed a way for both client and master server to know when the connection has been severed. We thought of using a **‘heartbeat’ mechanism**, in which the client sends a message to the server every second to confirm that its connection still functions.

If the server drops offline, the client’s heartbeat message won’t go through and it will know to switch to another server connection.

(Abandoning) the Heartbeat Mechanism

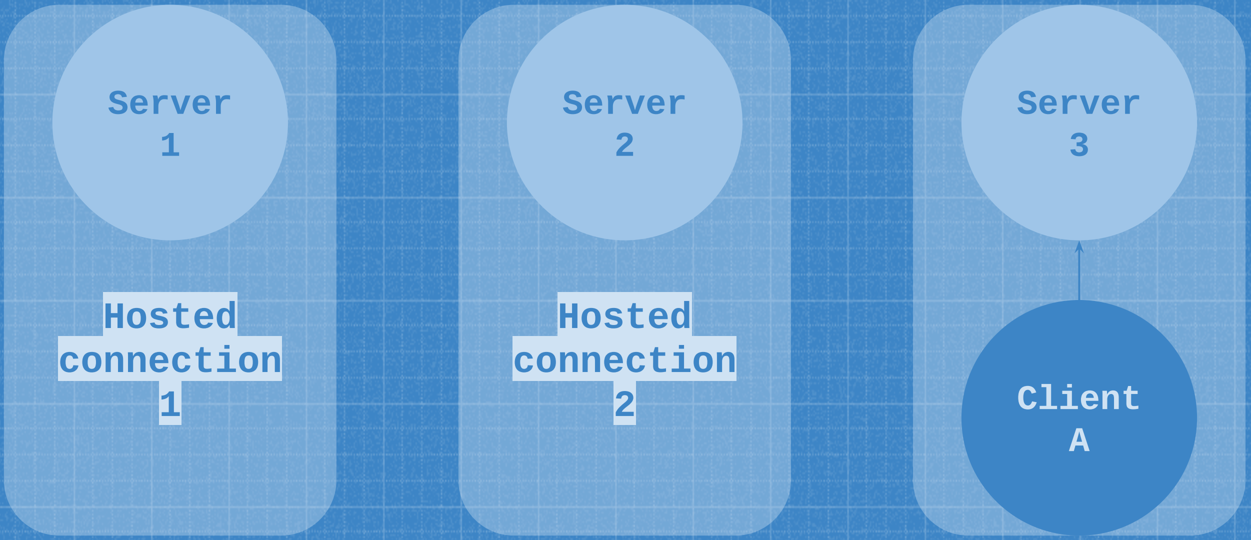
We then realized that we didn’t even need a heartbeat mechanism because we could simply prompt the client to connect to a working connection and prompt the servers to re-elect a master as soon as a network interruption was detected!



```
Server 3 is down. Initiating leader election.
Server 1 is down. Initiating leader election.
Server 1 is down. Initiating leader election.
Server 2 is down. Initiating leader election.
Server 1 is down. Initiating leader election.
Server 2 is down. Initiating leader election.
Server 2 is down. Initiating leader election.
```

Resolving Misconceptions; Hitting a Wall

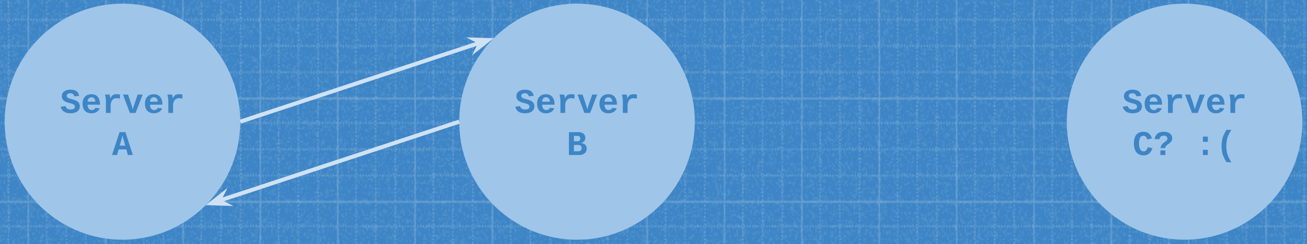
We quickly realized that each server should have their own database of identical information so that we are truly in safe hands if the ‘master’ should fail. This design change meant that we would have to come up with a way for servers to communicate with each other. This is what we were initially picturing our network to look like:

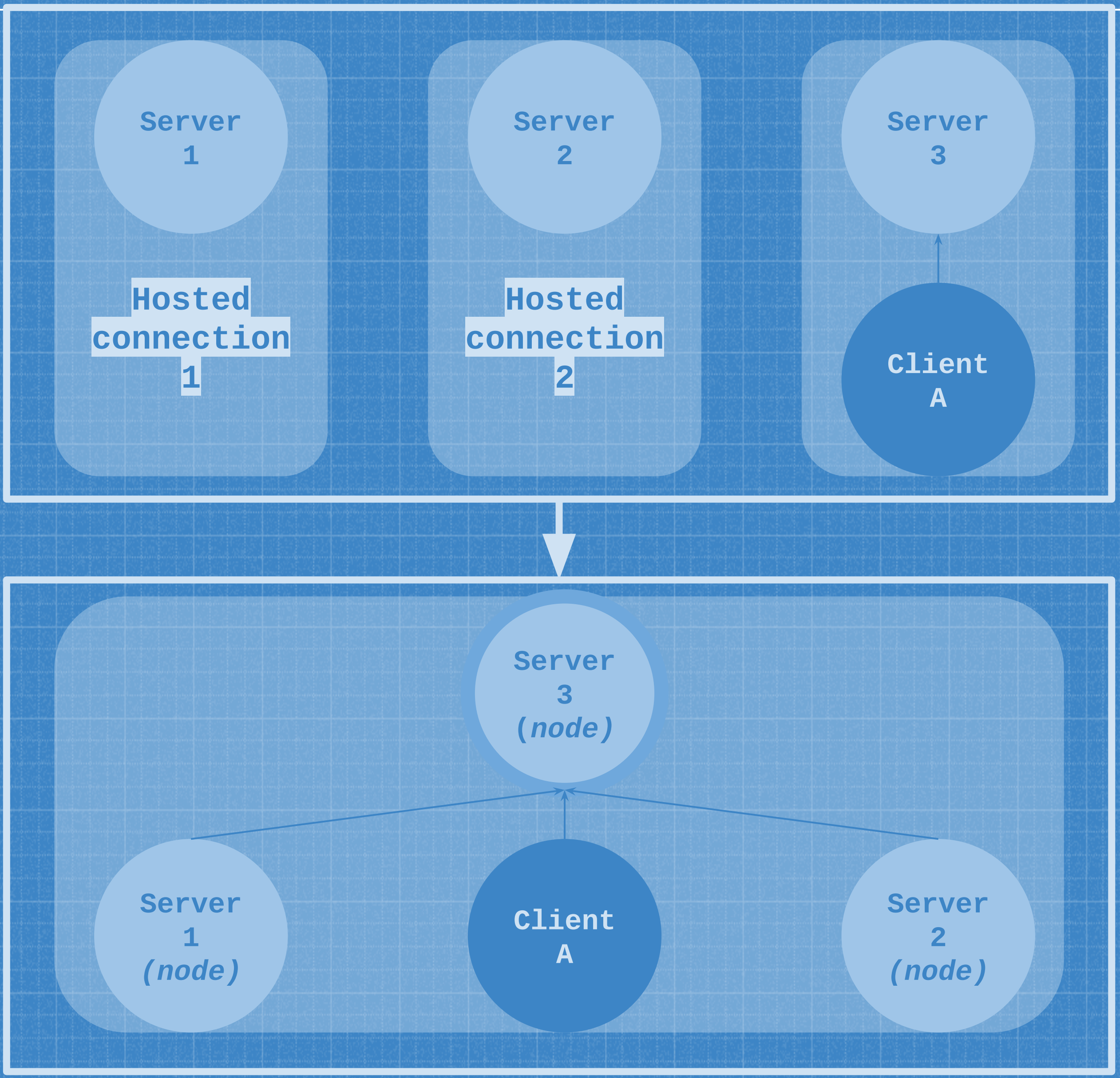


How do Servers Talk to Each Other?

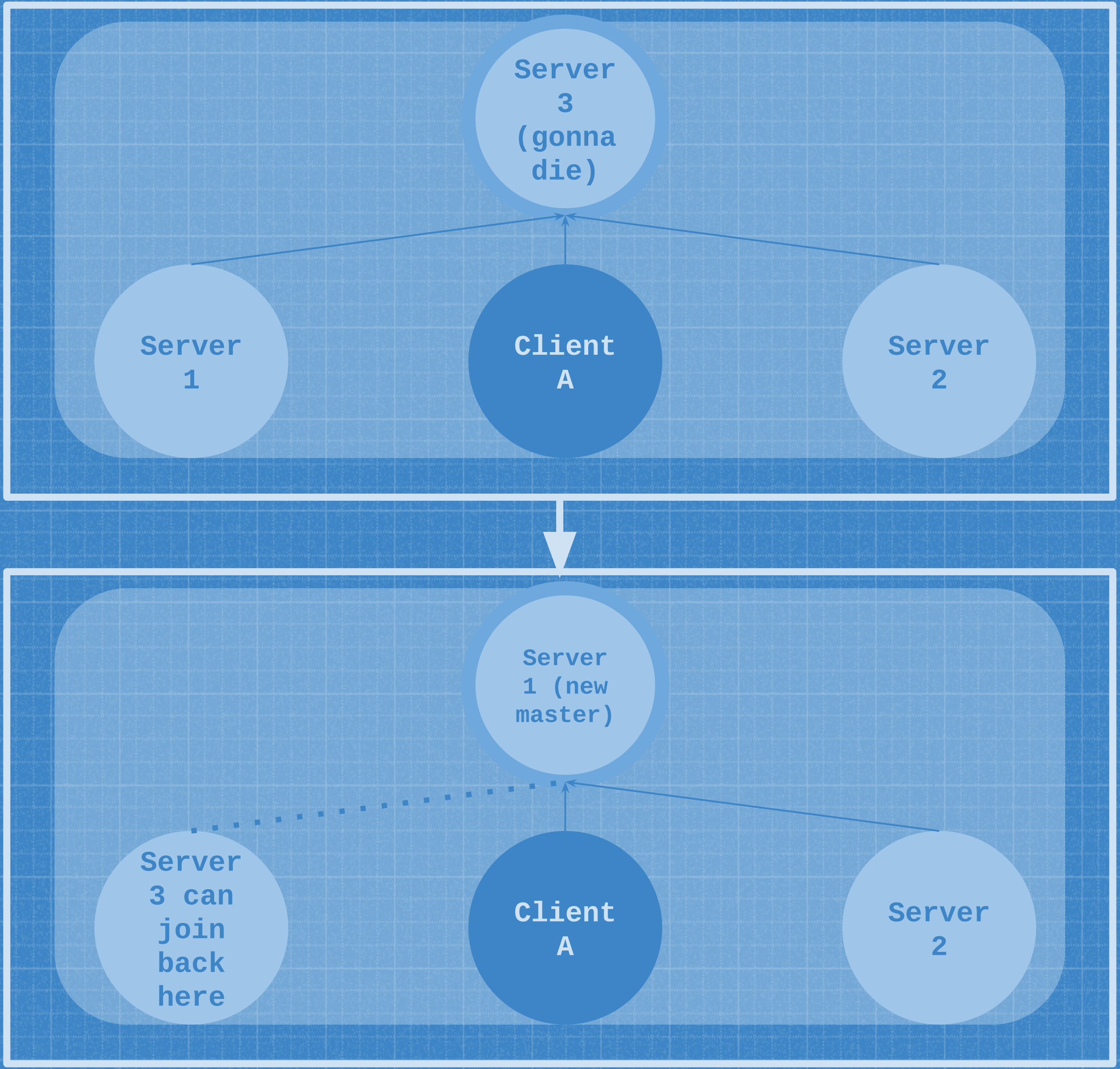
Under this current framework, we were lost as to how to allow servers to talk to each other. We explored Server A hosting a connection that Server B connects to, and Server A connecting to Server B’s connection, but this method would get exponentially more difficult when you introduce a third server into the mix.

We also wanted to be able to toggle on/off a server’s ability to receive connections because we only wanted the current master to be able to host (simplifies the client-side connection-joining logic). The below framework would have foiled our plans:





So, we reworked our model:



What happens if the connection dies:

Our New Checklist!

1. Create the **master_flag** functionality that also functions as an on/off switch for an individual server's ability to host a connection
2. Create functionality for each server to be able to *join* a connection as well
3. Use the same "connection trial" code that we currently have for client.py —> implement this in the servers as well within an if statement linked to master_flag, so that they are able to join the master server in later steps

Our New Checklist! (Continued)

4. Write server code that allows for a pipe between replica and master. This is where the communication between master and replica will come from and is the same functionality we currently have to connect the server and the client

4a: Once we create our database implementation, we will add to this logic to automatically append/send ACKs according to network data (ex: adding a new message's data to one's personal ledger as a replica)

4b: Error handling for this: for example, what happens if the master doesn't receive an ACK? Should they resend? Assume the replica to be dead? Etc.

Our New Checklist! (Continued)

5. Send the connection health info to the replicas from the master; determine the algorithm that the replica will use to determine if it will become the next master. *(Steps 5-8 had already been accomplished via our election protocol, but we needed to test)*

6. Update the server flags according to the connection health info and elections

7. Error handling for if the new master is unable to start up their connection

8. Ensure that clients cannot try to type out a message during a network change– this will prevent us from needing to recover messages that do not get recorded in a database or sent while the system is ‘masterless’

Our New Checklist! (Continued)

9. Testing the error handling for the following situation:

Port 1 = current master

Ports 2, 3 = replicas, currently awake

- - - - - Port 1 loses connection - - - - -

Port 2 = elected master, but loses
         connection first

Initial Election Protocol Idea: Lowest Port

We need a protocol that will deterministically yet pseudo-randomly choose a new server in the event that the master fails. The choosing function should be able to run on three separate non-communicating servers and still end up with the same elected master. We were thinking of using mods or choosing by lowest port number. But, with example port #s 1, 2, and 3, look what happens:

Port 1 knows it will always be the next master if it is a replica

Port 2 knows it will be the next master if port 1 is the current master

Port 3 knows it will only be the master if Ports 1 and 2 both fail

Because Port 3 is least likely to be chosen, we decided to go with the **randomized leader election** instead of the lowest port # method to choose a new master.

Leader Election: Randomized Leader Election

The leader election method used in the provided code is a simple random-based leader election.

When a leader is needed, the **get_new_leader()** function is called. This function generates a random index within the range of available servers and returns it.

Slides 25 and 26:

Aggarwal, Ashok K., Shlomi Dolev, and Sajal K. Das. "Randomized Leader Election: Parallel Algorithms for Leader Election in a Complete Network." Computer Science Technical Reports, Paper 1058, Purdue University, 1989.

Advantages of Randomized Leader Election

Load Balancing: Randomly selecting a leader from the available servers distributes the leadership role across the system, preventing any single server from becoming a bottleneck. This can improve the overall performance and reliability of the system.

Simplicity: Randomized leader election is relatively easy to implement and understand, as it does not require complex algorithms or communication protocols.

Reduced Contention: In some systems, having multiple nodes trying to become the leader at the same time can lead to contention and increased message overhead. Randomized leader election can help mitigate this issue by decreasing the probability that multiple nodes will compete for the leadership role simultaneously.

Graceful Degradation: When a leader fails, a new leader can be selected randomly from the remaining servers, allowing the system to continue functioning with minimal disruption.

Multiple Databases for Fault Tolerance

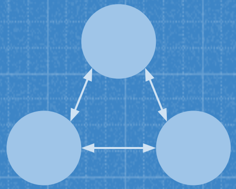
We implemented an architecture that utilizes a single database connected to each server. In the event that one server disconnects, the other databases have the correct information to then continue the process. The advantages to this over a P2P database system is:

- **Redundancy:** When each server has its own database, the data is replicated across multiple locations.
- **Load balancing:** With multiple databases, the load can be distributed across different servers. If one server/database becomes unavailable or experiences performance issues, the other servers can still handle the incoming requests.
- **Isolation of faults:** When each server has its own database, a failure in one server/database is less likely to affect the other servers.
- **Easier recovery:** In case of a failure, having multiple databases allows for faster recovery.This minimizes downtime and ensure the system remains available to users.

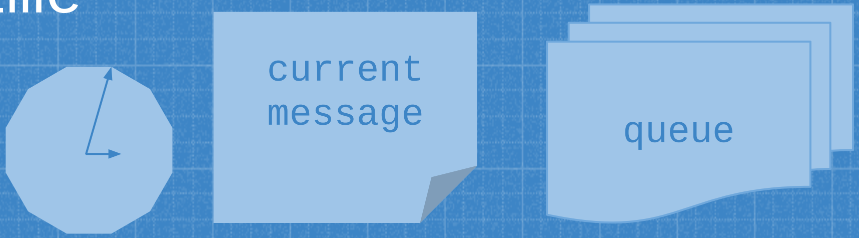
Programming Assignment 2

Understanding the Assignment and Specs

- We will be simulating three machines who can send messages to each other

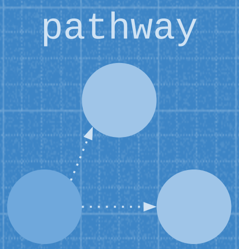


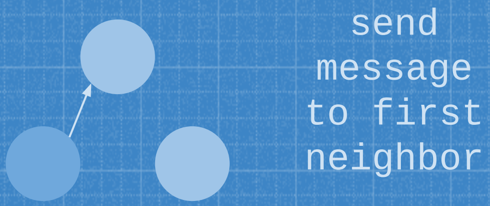
- We are simulating different machine speeds because each machine has a clock that limits the number of commands it can perform in real time

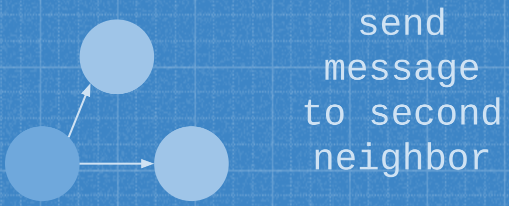


Thanks, Professor Waldo!

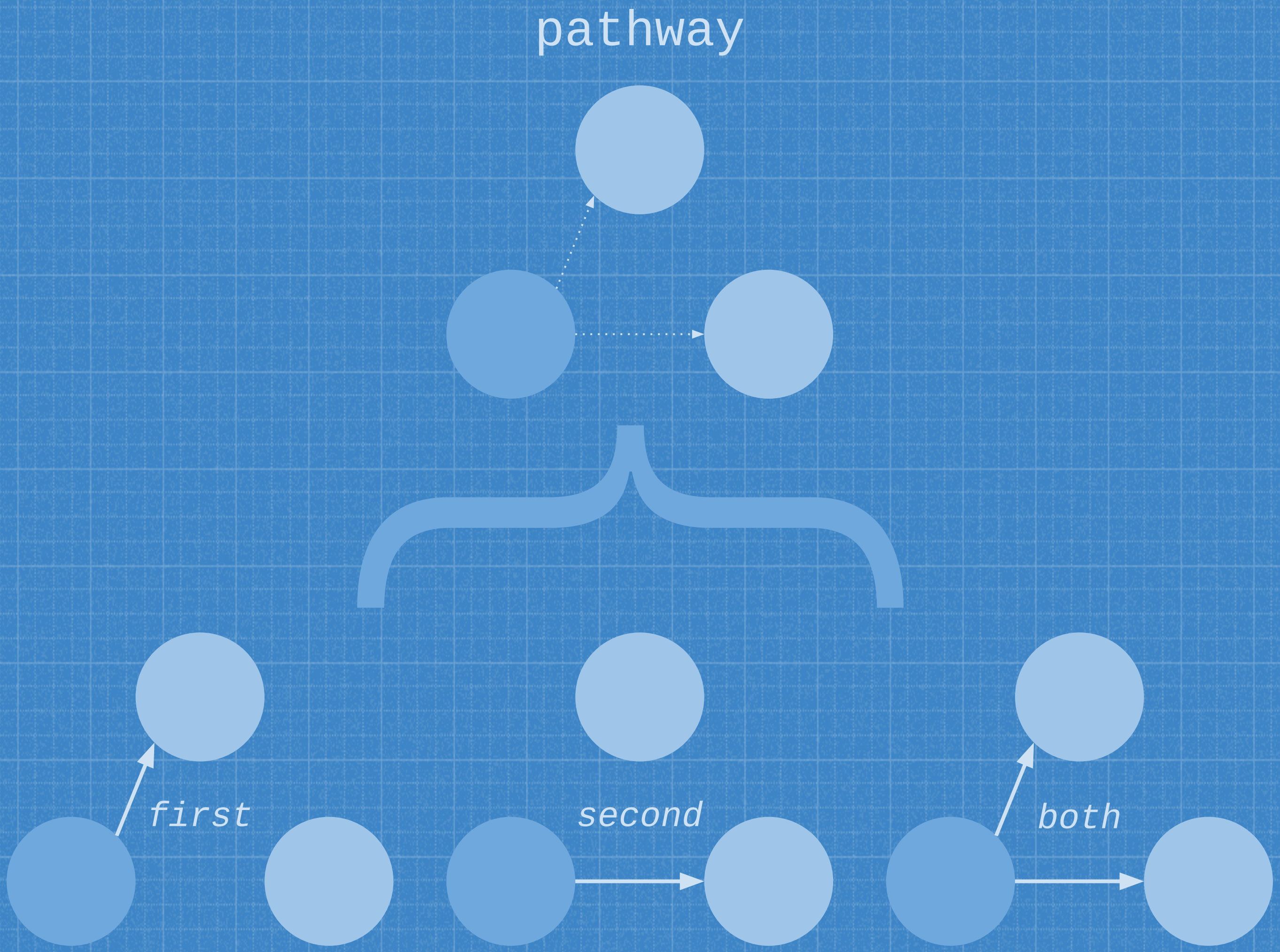
Professor Waldo’s code served as a fantastic framework for our project. What we learned is that there are six options of what can happen: each node can send a message to its first or its second neighbor (2 choices), and there are 3 nodes (2*3 total choices).







If we break down the “pathway” diagram in the previous slide, **three “options” emerge**, which we coded in clockSimMore.py:



Each Machine

- Maintain a queue of pending events and an internal clock
- Write events to a machine-specific log
- Bind to a socket, reserved for this machine, and listen for incoming
- messages, which are added to the queue.
- Execute (at most) a pre-specified number of the certain instructions; write to a log

Steps for a Processor

1. The processor will have a randomly-generated speed
2. We need to establish a connection
3. The processor will execute as many actions as it can based on its given speed and whether there are messages in the queue (as per the specifications)

- If we’re sending, we can either send to the first neighbor, second neighbor, or both (we literally define the functionality for “both” as first, then second)

Design Choice: Breaking the Problem Down

When we really abstracted the message sending, we were able to define “first” and “second,” then use these definitions to define “both.”

This abstraction saved us time and hopefully made our code a lot simpler + easier to understand!

From the professor’s code, we were able to expand to 2- and 3-way messages.

Lamport’s Timestamp Algorithm

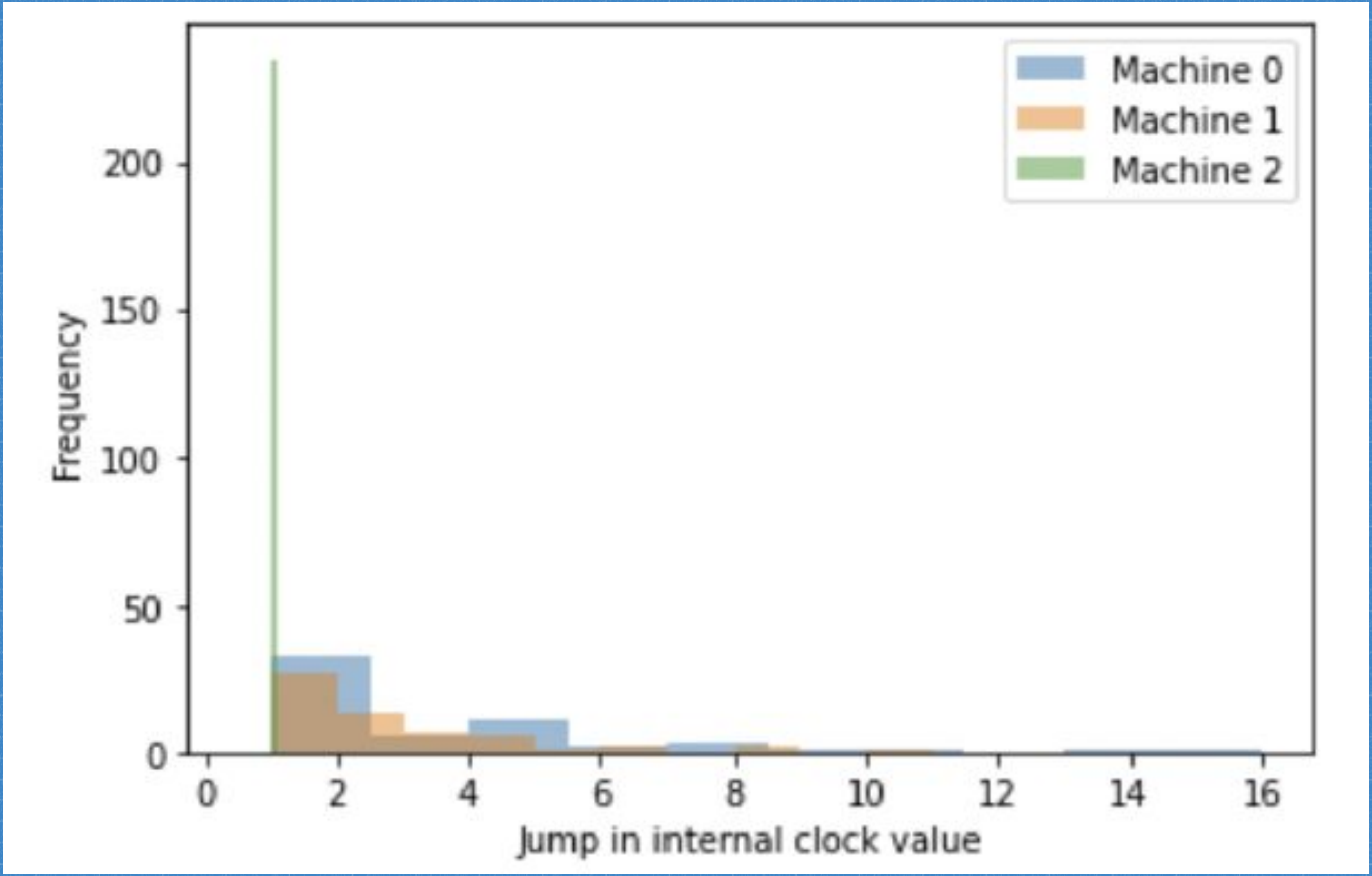
To maintain the internal logical clocks, we used Lamport's timestamp algorithm. We chose this algorithm because it is simple (each machine just maintains a single clock value), but still allows us to infer some information about the ordering of events from the logs. We update the internal clocks according to the following rules: The clock is incremented before executing any instruction (listed above). Any messages sent by a machine include the clock value of that machine.

Lamport’s Timestamp Algorithm (cont.)

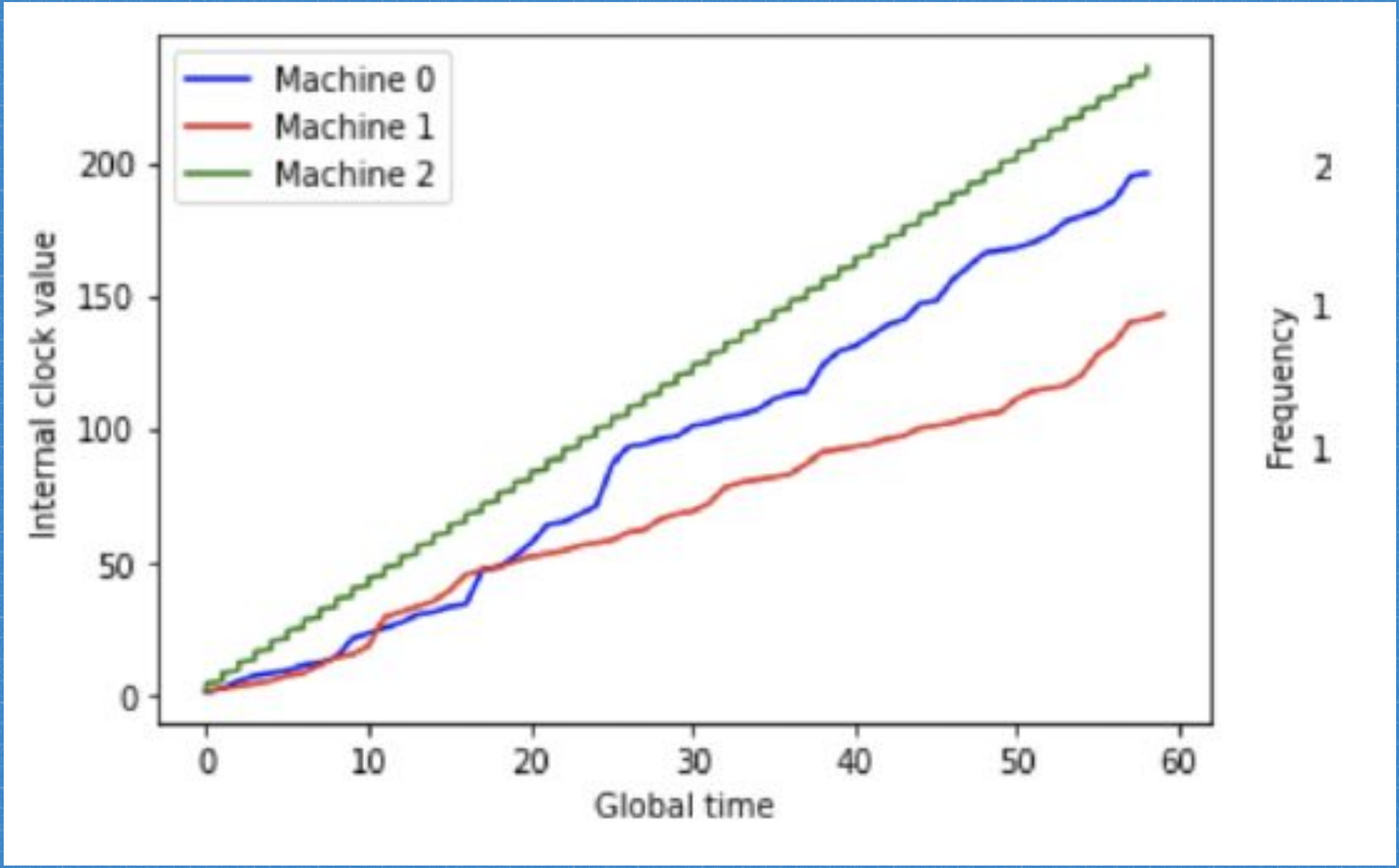
Having received a message from another machine, which includes a clock value, the clock value on the current machine is updated to be the maximum of itself and the received clock value.

We also decided to run the virtual machine as a separate Process, using Python's multiprocessing package. To model the fact that separate machines do not share memory, we used Processes instead of Threads (which do share memory).

Data Analysis



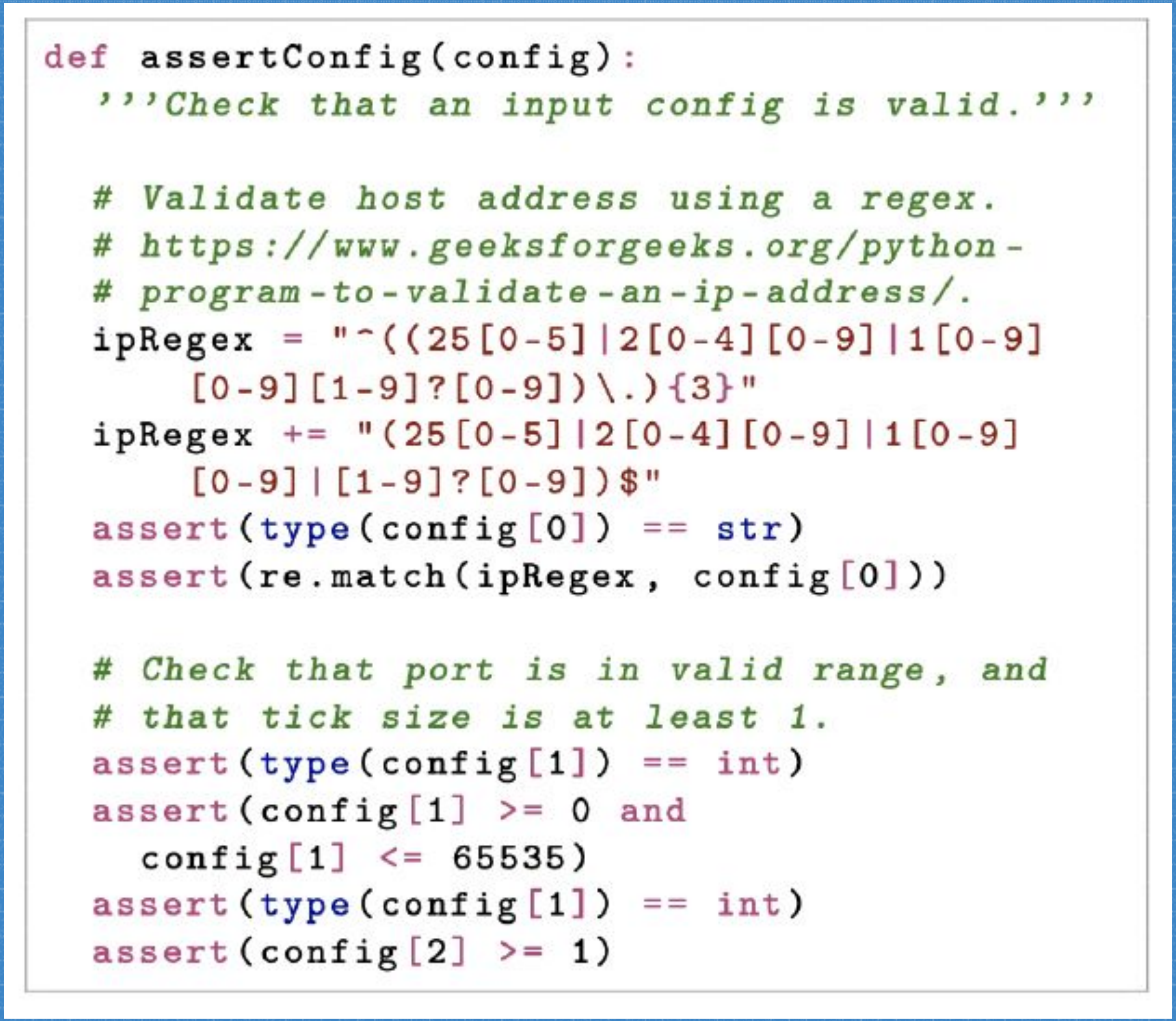
Data Analysis (cont.)



Data Analysis Findings

- Smaller variation in clock cycles naturally resulted in less drift and smaller jumps in clock values. As expected, queue sizes will still be large if there is also a smaller probability of an event being internal.
- Smaller probability of an event being internal significantly exacerbated the effects described above. Namely, we observed significantly larger jumps in values of logical clock values, larger queue sizes and larger drift, for slower machines. This makes sense; smaller probability of internal events means more messages are sent.

Unit Testing



```
def assertConfig(config):
  '''Check that an input config is valid.'''

  # Validate host address using a regex.
  # https://www.geeksforgeeks.org/python-
  # program-to-validate-an-ip-address/.
  ipRegex = "^((25[0-5]|2[0-4][0-9]|1[0-9]
      [0-9][1-9]?[0-9])\.){3}"
  ipRegex += "(25[0-5]|2[0-4][0-9]|1[0-9]
      [0-9]|[1-9]?[0-9])$"
  assert(type(config[0]) == str)
  assert(re.match(ipRegex, config[0]))

  # Check that port is in valid range, and
  # that tick size is at least 1.
  assert(type(config[1]) == int)
  assert(config[1] >= 0 and
    config[1] <= 65535)
  assert(type(config[1]) == int)
  assert(config[2] >= 1)
```

Unit Testing (cont.)

Because of our design choices, our implementation of clocks does not contain any input or output methods but rather runs several workers through different processes and threads that don’t necessarily return any values. These processes modify data structures and print to the LOG txt files. We completed unit testing by using assertions within each method, ensuring that messages were sent and received in a valid format.

Programming Assignment 1

Monday, February 6th: Specifications

1. **Users can create, log into, and delete accounts**
   a. What happens if a user tries to delete an account before their message has delivered?
   b. If a user isn’t currently logged in, their message should be queued; deliver their message as soon as they log in
   c. Confused by “deliver undelivered messages to a particular user”
   d. Could use a SQL database for this, but based on the scope of the assignment, we decided that it makes more sense to use a dictionary to store account usernames and passwords (limitation: passwords are not secure this way because they are stored on an accessible txt file!)

Monday, February 6th: Specifications (cont.)

2. **Build a wire protocol**
    a. One file that defines the functionality of a server
    b. One file that defines the functionality of a client
    c. Need to build a client, server that use these functionalities
    d. Support multiple clients; don’t need to support multiple servers
    e. Write a README with detailed instructions on how to use these files as server and client wanting to chat
3. **gRPC implementation**
    a. Use gRPC to instead build the wire protocol, then compare over various factors: 1) complexity, 2) performance, 3) size of buffers, etc.

Tuesday, February 7th: Preliminary Research

- Message — python-can 4.0.0 documentation
- Documentation | gRPC
- Welcome to gRPC Python’s documentation!
- Can someone explain what a wire-level protocol is? - Stack Overflow
- About: Wire protocol
- WebSocket: Simultaneous Bi-Directional Client-Server Communication | by Gabbie Piraino | Medium
- Socket in Computer Network - GeeksforGeeks

Wednesday, February 8th: C++ Implementation Attempt

We first thought to implement with C++ because we were both fairly familiar with the language and wanted to use its range of packages.

After looking into a Python implementation, we realized that Python was far more efficient, elegant, and easy to code in (even though C++ compiles faster and is generally more comprehensive.)

The next slide shows the common pseudocode between our C++ and Python implementations:

Wednesday, February 8th: Pseudocode Behind the Approach (CLIENT-SIDE)

1. Establish the host and port
2. Create the socket and connect
3. Create a message that stores the client’s input
4. Until the message reads ‘bye’, signaling the end of the conversation, use the socket to send the message to the server. Be able to receive and print the server’s response. Then, take in a new message.
5. Repeat step 4 as needed
6. Close the program

Wednesday, February 8th: Pseudocode to Code Ideas (CLIENT-SIDE)

```
1. Establish the host and port
2. Create the socket and connect
3. Create a message that stores the client’s input
4. while (message != “bye”)
   a. Use the socket to send the message to the server
   b. Receive and print the server’s response
   c. Take in a new message
5. Close the program
```

Wednesday, February 8th: Pseudocode Behind the Approach (SERVER-SIDE)

1. Establish the host and port
2. Create the socket, bind the host and port
3. Listen for and connect to new connections
4. Receive and decode the client’s message; return an error if this is undesired data
5. Create a message that stores the server’s input
6. Encode and send this message to the client
7. As long as you’re receiving data, keep doing steps 4 through 6
8. Close the program

Wednesday, February 8th: Pseudocode to Code Ideas (SERVER-SIDE)

```
1. Establish the host and port
2. Create the socket, bind the host and port
3. Listen for and connect to new connections
4. while True:
   a. Receive and decode the client’s message; error message + break if undesired data
   b. Create a message that stores the server’s input
   c. Encode and send this message to the client
5. Close the program
```

Thursday, February 10th: Testing it Out

We tested the C++ code we generated from the pseudocode on one computer. We realized that, using this design, we created a requirement for a certain order in our communication: unless exactly one message is being sent alternately from client to server (A->B, B->A, repeat), our code does not work.

We would like to create an approach that does not restrict the messaging patterns of server and client. At this point in time, we also had not switched over to using Python.

Thursday, February 10th: Testing it Out (cont.)

We were also unsure of how to handle putting both client and server on one host. For now, we designed code in which the client and server are on the same computer.

We had yet to test this with two different computers and also felt very lost as to how to implement gRPC. For instance, it took a while to figure out how to auto-generate the Python code via the terminal after writing our proto file.

Monday, February 13th: Buggy Python Implementation

We finally switched over to Python and saw immediate improvements. For instance, because of Python’s functionality that can support “.connect,” for example, our earlier steps in the pseudocode became much simpler:

- Creating a client socket:
  - `client_socket = socket.socket()`
- Connecting the client socket:
  - `client_socket.connect((host, port))`
- Taking input:
  - `message = input(" -> ")`

Tuesday, February 14th: Remaining Todo List

1. Fix the Python implementation of the wire protocol
2. Create code that supports specification pertaining to accounts
3. Start on gRPC implementation
4. Upkeep design notebook
5. Compare the two approaches; record findings in design notebook

Wednesday, February 15th: Implementing Login User Flows

We created functionality for users to register and log in. We record the username and hashed password in a txt file to be referenced when a user is trying to log in again. We still need to figure out account deletion as well as link this login flow to our client.py.

We are a little stuck as to how to create an “inbox” for queued messages that we can show the user once they log in. We also do not know what to do if a user logs off before their message can send.

Figuring out the Delete Functionality

We first tried to create an option from the main menu for a user to delete their account but then realized this would be a security issue if a user can delete an account before logging in. We played around and initially put the code into client.py because client.py would only be called from login.py once the user is logged in.

We then copied the contents of client.py into login.py, but we realized that having this content in one file could create dependency issues.

We also wanted users to be able to delete their account at any time as long as they are signed in, so we decided to add the Delete functionality by detecting when a user types “/delete account” in to the terminal. After a confirmation prompt, the account information will be deleted from the .txt file.

Deletion was tricky to figure out– we realized that there isn’t a great way to simply remove a specific line from a .txt file in Python. So, we used a temp file to create the updated accounts information list, then renamed the accounts information list to the name of the temp file.

We also decided not to auto-delete an account after multiple failed login attempts and instead just direct the user to create a new account. We tried to put ourselves into the user’s shoes to design the most helpful user flow.

Friday, February 17th: Starting Proto

gRPC was initially a nightmare to figure out. Getting familiar with package installation and M1 Mac problem solving was a big barrier to entry for getting started on Part 2. After researching multiple client/server gRPC approaches, we saw many different and complicated ways to display a chat stream. It took a couple days for us to understand the technical concepts enough to realize that the chat proto line is actually very simple: to send a message, you take in a message stream and output a message stream (the messages can even be of the same struct!).

- `rpc Chatter (stream Reply) returns (stream Reply);`

We worked on first building an MVP, then adding extra functionality and complexity. The first version of our gRPC was a chat with no accounts and no login that kept crashing.

It was difficult to figure out what logic went in what file– did a given function belong in client.py or in the gRPC code, for instance? We tried adding functions and modifying the auto-generated code to no avail, eventually realizing that we didn't need to touch those two files and instead would need robust client and server Python files.

Because so many files share common names and functions, we had a lot of inconsistency based errors in our initial gRPC code where we didn't update an aspect of the code in each relevant file or were using inconsistent naming conventions across files. Taking a step back, simplifying, and first focusing on the MVP was really helpful– we started over around three times and became quicker at building up the basic functionality from scratch.

We wanted to explore other uses for gRPC beyond bidirectional communication, so on Saturday and Sunday, we worked on creating Unary, Server-Side, and Client-Side implementations.

Friday, February 17th: Trustable, Working Part 1

We also finally connected functionality between client.py and login.py, also testing and refining our Delete function, to get a nice, acceptable version of Part 1!

We added functions to display a user list as well as search for a user using the first letter of their username.

If we have time, we’d love to continue making our code more efficient, add more comments and README documentation, and possibly introduce fun colors into the CLI interface to improve the user experience.

Friday, February 17th: Remaining Todo List

1. Allow for multiple clients in Part 1 (want to switch to more of a node implementation where the server is merely the connector between two separate clients)
2. Figure out database implementation for ‘inbox’ (message queueing) functionality
3. Create and use unit tests for our code
4. Compare the two approaches; record findings in design notebook
5. Write README, commenting, general cleanup

Brainstorming for Inbox Database Implementation

- Switch over to SQL so we can use joins and eloquently store a larger amount of data
- Create a ‘message array’ for each user that is printed out when a user logs in
- Create a txt file for each user in which messages to that user are tagged with the receiver username and appended to the text file → delete new messages from txt file when receiver username is logged in (they’ve been read) → print out txt file when a user logs in on their next session because only their unread messages will remain
- Use console logs to record messages and print them if necessary

Saturday, February 18th: We added a gRPC Easter Egg…
try to find it!

Saturday, February 18th: Global User Functionality

1. Consolidated client.py and login.py in Part 1 into just a client.py file; fixed any possible dependency issues that could come with this change
2. Redesigned Part 1 using a ‘node’ approach in which two independent clients can log in with username/password and message each other
    a. This new design will help us because, previously, the server wasn’t logged in as an account while chatting with the client.
    b. Now that we have accounts for each chatter, we can assign a receiver to each message so that we know where to send queued messages
3. Created threads so conversations can happen independently

Saturday, February 18th: Nice-to-Haves for Part 1

1. Ability to still send messages if the other user you were chatting with logs off
2. A prettier CLI look and feel
3. A working message queue system
4. More messaging freedom than the one-for-one chat system we have (where each user trades off sending one message at a time, in an alternating order)
5. Functionality if a user wants to change their password or forgets it

Sunday, February 19th: Auth System

1. Improved reimplementation of server.py (originally changed to handle multiple clients and support threads)
2. Implemented functionality for /logout; created a system to tell whether an existing user were authorized
3. Reimplemented Delete() to account for changes made to server.py
4. Implemented inboxes and wrote unittests
5. Wrote a robust and detailed README.md with instructions for users

COMPARING PART1 and PART2 APPROACHES

PART1

- Message response time is noticeably slower
- Coding experience was a lot more conventional– played to our existing experience
- Buffer size was larger
- More room to be creative with the client/server setup (for instance, we switched from a traditional model to a ‘node’ model)

PART2

- Very big learning curve at first that honestly made this method less efficient
- However, once we understood how to implement functionality, our bonus three functions were very quick to make
- Buffer size was smaller because of tight packing of protocol buffers
- Code is less complex, but also less familiar to the average coder