CSU33D03 Computer Networks Report

Group 19
Elizabeth Woodcock(21365394)
Nicholas Giles(21364255)
Stephen Komolafe(21336975)
Suhani Singla(23364401)

Introduction

This report outlines the steps our group has taken in designing and implementing a computer networking consensus protocol. In designing the protocol, it must have the ability to achieve consensus and interoperate with up to three other groups. There are many objectives that this project strives to achieve, including node discovery, negotiation between nodes on their offered services, capabilities, and constraints, and the ability to reach consensus in order to perform an agreed-upon action. This report provides an overview of our consensus protocol, giving detail to the specifications, capabilities, design choices, and implementation of our codebase. It shows our protocol's successful interoperability and consensus-reaching capabilities.

First steps taken

The first step taken in this project was to decide on a basic protocol that we wanted to implement. Upon discussion, it was decided that we would write our code in Python and develop our code using TCP. We chose to use TCP as it is a reliable data transfer protocol that implements congestion control. It is a connection oriented service, meaning a 'handshake' occurs between the client and server before any messages are sent. The TCP handshake is a three-step process, as three segments are sent between the client and server before a connection is established. This allows us to keep track of what nodes are communicating with the server and can ensure that data is being sent and received between nodes.

We communicated with other groups in the class to see who we would collaborate with depending on their ideas for consensus and agreement. A discussion was had between three other groups in order to decide upon the collaborative activity of consensus and agreement. We agreed on implementing a voting system that would prompt clients to vote based on a question provided by the server. Each client was able to cast their vote, and if over 50% of the answers were the same, a consensus would be reached. This consensus was then broadcast to all the clients.

Setting up the Client-Server protocol

To start, we had written code that implemented a simple client-server connection. Originally, a single code file was implemented that functioned as both a client and a server (i.e., it commences operation as a server, but if it detects another server already running, it switches to a client in order to connect to the other server). However, as it became increasingly difficult to keep track of which parts of the code were responsible for server-side operations and which parts for client-side operations, we eventually decided on splitting the code into a separate server and client file. For both the client and server, a socket is created using the function '`socket.socket(socket.AF_INET, socket.SOCK_STREAM)`', with the first parameter specifying the address family (IPv4 in our case) and the second parameter being the stream-oriented socket for the TCP protocol.

When the server establishes itself on a certain port, it listens from that port, and clients can connect to it via that port number. However, the client then sends and receives messages from a different port that it sets up, known as the client port. This allows the server to connect to multiple clients simultaneously. When a server successfully connects to a client, it sets up a thread to handle each client. Different parameters are passed to the thread, such as the client socket and the client's IP address.

Provided Services

When a client successfully connects to the server, they are offered a variety of services to choose from. The output on the clients terminal is as follows

```
Correct! You may now vote and request for node discovery and
negotiation.
Your PIN is: 206
<<'setup' to declare node info
<<'discover' to discover nodes
<<'negotiate' to negotiate with server (store node info)
<<'receive' to receive stored node info from server
<<'vote' to enter voting consensus
<<'stats' to view current consensus statistics
<<'exit' to exit consensus protocol
 [Select an option]
```

Each client that connects is given a unique 3-digit PIN that is generated using a random number generator function. This PIN serves as a means for clients and the server to easily and efficiently identify other connected clients without having to type in each client's entire IP address every time.

The '`setup`' option allows the client to initialise their node information, which is the client's services, capabilities, and constraints, which would be used in the 'negotiate' and 'receive' options. This option was added fairly late into the development of our protocol to improve its compatibility with the client files from other groups. We had initially intended for the node information to be hard-coded in the client-side, however, it was only after we realised that the client files of the groups we were collaborating with might not have had node information hard-coded in, or not have node information at all, that we made it so our server prompts clients to input their node information themselves.

The '`discover`' option announces the client to the server and in return the server sends the client a list of all currently connected clients including their ip address, port and PIN. This is especially useful as, when a client first connects to the server, they can only see their own

PIN. Therefore, to obtain the PIN of another connected client they may wish to obtain node information from, they are required to use this option beforehand.

The `‘negotiate’` option allows the client to negotiate with the server. To elaborate, the client collects all its node information and notifies the server, where the server then asks for permission to take this node information and store it in a server-side dictionary. The client then has the final say in this request, where it can either grant the server permission or deny it. Once permission has been granted, the node information has been stored in the server-side dictionary, and the entry in said dictionary is linked to the PIN of the respective client.

The `'receive’` option allows the client to retrieve node information from the server about any other client connected to the network. When prompted, they can choose between receiving information from either a specific client by inputting their corresponding PIN number or from all currently connected clients by inputting ‘all’. With this option, any client can now know more about the services, capabilities, and constraints of their fellow clients.

The ’`vote`‘ option allows the client to enter the voting system we have in place to form a consensus. The server prompts the client to cast their vote of either ‘A’ or ‘B’ on a simple “would you rather?" question. A consensus can only be formed if there is more than one vote cast, the number of votes for ‘A’ and ‘B’ is not the same, and the number of votes for either 'A' or 'B' is greater than half the connected clients.

Finally, the `'stats’` option has the server perform the necessary calculations to produce statistics detailing what answers have been collected by the server so far and the percentage frequency of said answers. These statistics are then returned to the client by the server for their observation. This option was a last-minute addition to the development of our consensus protocol, as we needed a means to effectively demonstrate the use of data generated by the voting system.

Adding Security Measures

When meeting with the groups that we were interoperating with, we found that they all had implemented a security measure into the code in the form of a riddle. In order to ensure compatibility between our code and theirs, we had to add a similar security measure to our consensus protocol. The way it works is that when a client attempts to connect to the server, it must first answer a riddle correctly in order to gain access to the network. So before any options are provided to the client, they are required to answer the following question correctly:

```
‘What has keys but can’t open locks?’
Answer = ‘keyboard’
```

The Voting Process

When a client chooses the 'vote' option, the server prompts them to make a choice between two options (A or B in our case). The client then sends its vote to be handled by the server.

Once the server receives the vote from the client and adds it to the list of votes, it checks if a majority consensus has been reached by calling the `'check_for_majority()'` function, passing in the list of votes and the set of authenticated clients.

In this function, it first checks if there are any votes to record. If the list of votes is empty, ‘not votes’, it implies that there are currently no votes, thus the function returns None, indicating that no consensus can be reached yet. If a majority consensus is reached, the '`broadcast_consensus()`' function is called.

This function sends the message to all authenticated clients, informing them that a consensus has been reached. Clients receive this message and can continue participating in the consensus process by either casting another vote or choosing one of the other options.

Encountered Problems

The most significant problems encountered during the development of this project were:

1. Handling concurrent connections to multiple clients, and
2. Making our code compatible with other groups’ code.

For (1), our solution was to implement multithreading via the Python standard library *threading.py*. In the final implementation of our consensus protocol, this is used in the server code to handle each client node as a separate thread running in parallel. To do this, a new thread is started when a client joins the consensus server. This thread runs the function to send and receive messages from the client.

Regarding (2), the requirement of compatibility with other groups’ code necessitated the rewriting of much of the code and the structuring of the consensus protocol to have a simple client function that sends and receives messages from the server, while the main consensus functionality takes place on the server side. This allowed for other groups’ client implementations to connect to our server without difficulty. In addition, having simple client code meant our clients could connect to the other groups’ servers, regardless of their consensus implementation.

Limitations

Although our established consensus protocol works well in its current state, there are a few limitations that could affect its reliability and performance in larger situations. The most significant issue is the scalability of the client-server model.

As the number of clients increases, the server could become a bottleneck, leading to the potential for server overload. Additionally, with the current setup, clients are not automatically notified when a new client joins or an existing one leaves the network. This lack of dynamic network awareness may lead to outdated information being held by the clients, affecting the overall reliability and efficiency of the consensus process.

Moreover, the users have to manually prompt for actions like discovery, negotiation, or receiving updates. This manual intervention may cause delays in decision-making and inefficiencies in the consensus process, especially in situations where real-time data is essential.

A fundamental challenge is the protocol's reliance on the server, which represents a single point of failure. Without it the network as a whole would become nonfunctional, leaving it more vulnerable to interruptions and security flaws. These limitations highlight the need for an improved approach that enhances scalability, automates the network, minimises user-required interventions, and reduces dependence on a single server.

Potential Solutions

To address the limitations of our current server-client model, particularly scalability and dependence on a central server, a feasible option is to switch to a peer-to-peer (P2P) architecture.

In a P2P model, each node or peer serves both as a client and a server, allowing for direct data exchange with other peers without the need for a central server. This decentralised approach can improve the scalability of the network, as the addition of more peers can increase the overall capacity of the network.

Moreover, the P2P model markedly diminishes the risk associated with a single point of failure. In our current server-client model, the failure of the central server could potentially bring down the entire system. Conversely, in a P2P network, the failure of a single peer has no impact on the network's overall functionality.

Implementing a P2P model could also enable more dynamic and flexible consensus processes, as peers can directly negotiate and reach consensus among themselves. This would require implementing effective discovery and synchronisation mechanisms to ensure that peers can efficiently find each other and stay updated on the network's state. Despite the challenges of implementing such mechanisms, the advantages of increased scalability and flexibility in consensus formation make the P2P model a compelling alternative for enhancing our current protocol.

Real-World Adaptation

With further development, we believe that our consensus protocol could potentially be applied to real-word scenarios on a small or large scale providing a solid framework for decision making in a distributed system of devices

For instance in a small-scale scenario such as a board meeting, the consensus protocol can be used to reach an agreement among board members on critical company-wide decisions. Each member can participate as a client by voting on proposed actions and the server facilitates this process by storing each vote and ensuring that a consensus is achieved before proceeding with the business decision. This ensures that decisions are made collaboratively and democratically, fostering transparency and accountability within organisations.

Our consensus protocol would also be able to be used on a larger scale in systems that involve nationwide political polling, where thousands of individuals participate in voting processes as clients to elect leaders or determine public policies. In this scenario, the server would act as a central hub for collecting and aggregating votes from diverse populations across different regions. The protocol ensures that each vote is accurately recorded and counted and that real-time statistics are also given. The outcome of these elections or referendums would then be decided by the majority consensus. This would promote fairness and legitimacy in the democratic process, allowing for the representation of diverse perspectives and ensuring that the will of the people is accurately reflected in the final decision.

Overall, with more time to develop, our consensus protocol certainly could be applied to a variety of real-world scenarios, from small-scale board meetings to large-scale international political polling systems, in order to support democratic decision-making processes and ensure the integrity of voting.

Lessons learned

Through this project, we gained valuable insights and practical knowledge in several key areas of computer networking and protocol design. One of the fundamental skills we acquired was the implementation of TCP using sockets in Python. This improved our understanding of how TCP is used in transfer of reliable data between clients and servers, guaranteeing that messages are delivered in order and without getting lost, which is crucial for maintaining the integrity of communications within our network.

Additionally, we explored the rules of creating a consensus protocol that required effective interoperability among various nodes in the network. This process involved not only technical implementation but also coordination and negotiation with other groups, which taught us a lot about teamwork. Handling multiple clients simultaneously was another significant challenge we overcame.

We used threading, allowing the server to communicate with multiple clients at once.This highlighted the importance of concurrency control in network programming. Lastly, we learned about using IP addresses and port numbers, to connect nodes within our network. Overall, the project was a comprehensive learning experience that enhanced our technical skills in networking and consensus protocols.

Conclusion

This project gave us a great insight and understanding of computer networking and consensus protocols. We successfully implemented TCP for reliable data transfer and achieved consensus with three other groups. Throughout this project we learned the importance and effectiveness of interoperability and network communications. We overcome multiple challenges including handling multiple connections in a network and ensuring compatibility with other groups. We demonstrated the potential for our design to be implemented into a real-world application, however, we acknowledged the limitations of our design and have provided a solution to overcoming these limitations. We understand the scalability of our model and we have learned from the challenges that arose when designing and implementing our consensus protocol.

Appendix

[Code provided in separate submission]

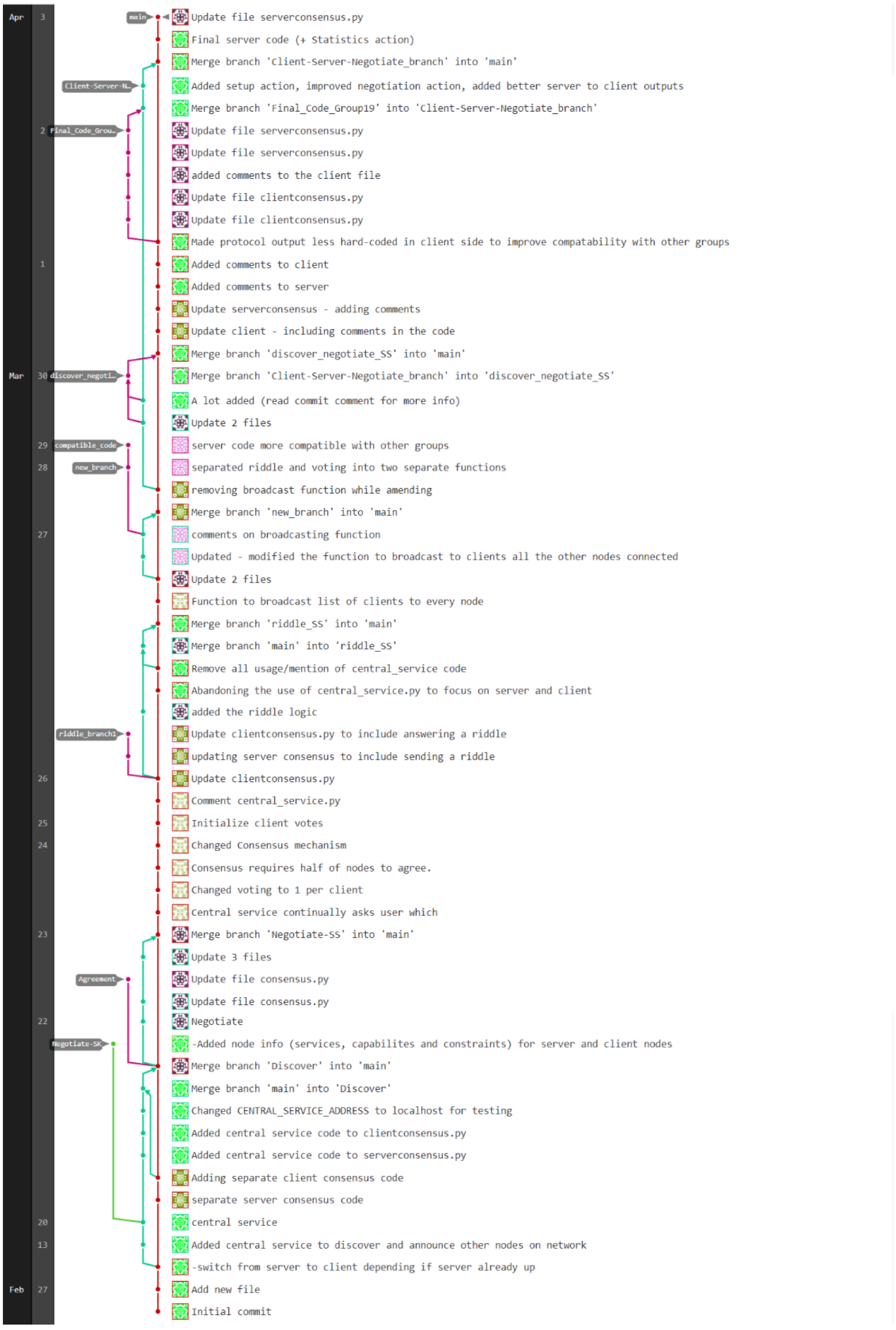
[GitLab Repository Graph]

-> Stephen Komolafe

-> Elizabeth Woodcock

-> Suhani Singla

-> Nicholas Giles



```
Update file serverconsensus.py
Final server code (+ Statistics action)
Merge branch 'Client-Server-Negotiate_branch' into 'main'
Added setup action, improved negotiation action, added better server to client outputs
Merge branch 'Final_Code_Group19' into 'Client-Server-Negotiate_branch'
Update file serverconsensus.py
Update file serverconsensus.py
added comments to the client file
Update file clientconsensus.py
Update file clientconsensus.py
Made protocol output less hard-coded in client side to improve compatability with other groups
Added comments to client
Added comments to server
Update serverconsensus - adding comments
Update client - including comments in the code
Merge branch 'discover_negotiate_SS' into 'main'
Merge branch 'Client-Server-Negotiate_branch' into 'discover_negotiate_SS'
A lot added (read commit comment for more info)
Update 2 files
server code more compatible with other groups
separated riddle and voting into two separate functions
removing broadcast function while amending
Merge branch 'new_branch' into 'main'
comments on broadcasting function
Updated - modified the function to broadcast to clients all the other nodes connected
Update 2 files
Function to broadcast list of clients to every node
Merge branch 'riddle_SS' into 'main'
Merge branch 'main' into 'riddle_SS'
Remove all usage/mention of central_service code
Abandoning the use of central_service.py to focus on server and client
added the riddle logic
Update clientconsensus.py to include answering a riddle
updating server consensus to include sending a riddle
Update clientconsensus.py
Comment central_service.py
Initialize client votes
Changed Consensus mechanism
Consensus requires half of nodes to agree.
Changed voting to 1 per client
Central service continually asks user which
Merge branch 'Negotiate-SS' into 'main'
Update 3 files
Update file consensus.py
Update file consensus.py
Negotiate
-Added node info (services, capabilites and constraints) for server and client nodes
Merge branch 'Discover' into 'main'
Merge branch 'main' into 'Discover'
Changed CENTRAL_SERVICE_ADDRESS to localhost for testing
Added central service code to clientconsensus.py
Added central service code to serverconsensus.py
Adding separate client consensus code
separate server consensus code
central service
Added central service to discover and announce other nodes on network
-switch from server to client depending if server already up
Add new file
Initial commit
```

[Rough Project Timeline]

-had first group meeting

- Discovered the socket protocol, consider it as a potential direction to take the project in.
- Have made communications with groups we planned on interoperating with.

- Create and use single python file where client and server are combined as TCP consensus protocol base [consensus.py]

-Central service file [central_service.py] made for discovery component of protocol(announcing nodes)

- Server and Client node info (services, capabilities, constraints) added to have something for central service to negotiate for.

-Split [consensus.py] into [clientconsensus.py] and [serverconsensus.py] as it was becoming arduous keeping track of which parts of the code were responsible for server-side and which parts for client-side

- Localhost IP [127.0.0.1] added to test codes' current functionality on a single device running several instances of VS code for server, client(s), and central service.

-Negotiation aspect functionality further developed to allow server to obtain node info from client. Questioned if this counted as negotiation as the server just takes info immediately.

- Added threading library to client, server, and central service to run code concurrently with each other

-Added prompt for central server to ask client/server which node info to show

-Added simple voting system to riddle functionality in protocol, each client gets 1 vote and consensus is formed when number of votes is greater than half the connected clients

-Updated Client and server to include riddle functionality and logic (for compatibility with other groups)

-Abandoned the use of a central service as we realised that the server acts in a similar fashion and can retain all its functionality, ergo, all mention of central service was removed from server and client code.

-Improved discovery aspect by adding a function to broadcast a list of clients to every node from server

-Separated riddle and voting functionality into separate functions that can be called by the client

-meet up with other groups we planned on working with to make our consensus protocol compatible with their clients and vice versa

-Agreed on a form of negotiation to take place in our protocol (Server asks for permission from client to take and store node info)

-Dictionary added to server to keep track of client node info stored

-Added randomly generated PIN's to identify each entry in dictionary(each client node)

-Decided on separating each individual function as options to have the server prompt the client to choose the order they want to initiate each function

-'receive' option added to grab specified or all node info from server dictionary using the corresponding pin identifier or 'all' respectively.

-’setup’ option added to allow client code from other groups to set up their own node info to use in negotiation

-Code finalised, outputs refined and comments added

-last minute addition of consensus question and ’stats’ option to record the frequency of unique votes (in percentage form)

-held presentation

-professor mentions that we should look at how our current consensus protocol could have been improved further through the implementation of P2P network