IMPLEMENTATION OF A SIMPLE PEER-TO-PEER INSTANT MESSAGING PROTOCOL

Jonathan Moore, Isaac Rowe, Michael Sandell

ABSTRACT

Yet Another Instant Messaging Protocol (YAIMP) provides a method of asynchronous, peer-to-peer messaging that can be utilized in areas or mediums where traditional methods such as Internet Relay Chat are not viable. The motivation and need for such a protocol are discussed briefly. The protocol is similar to the Matrix and XMPP protocols and a survey of literature is included. It is implemented in Python and makes extensive use of the sockets and concurrent_futures libraries. The protocol is intended to be easily adaptable to many mediums but was developed and tested on local TCP ports. Experimentation results and a discussion of them are provided to help understand the current functionality of the protocol.

1. INTRODUCTION

Yet Another Instant Messaging Protocol (YAIMP) is intended to provide a method of asynchronous, peer-to-peer (P2P) messaging. This protocol relays messages across connected peers in order to eventually reach the final intended recipient. It operates in an asynchronous manner such that a user can receive a message that was sent to them before they connected and a message can be delivered from a user that has already disconnected. The design of the protocol was intended to be relatively medium agnostic so that it could be easily adapted to be deployed and a variety of transport mediums, e.g. Bluetooth, radio ways, or via satellite.

1.1. Motivation

The motivation behind developing YAIMP stems from a need for communication methods in a situation where traditional avenues are not available; for instance, in a disaster relief scenario where cell towers or internet service providers are not operational. Another situation that a protocol like this can be useful is in satellite communication since the satellites that need to communicate to one another may not always be in direct communication with one another (or connected to any other satellites for a period of time). Finally, in situations where there is communication blockage or censorship, such as jammed cell phone towers, a protocol such as YAIMP could be used to still allow people in that area to communicate with one another.

2. LITERATURE REVIEW

A literature review is a scholarly paper or article that presents the current knowledge including substantive findings as well as theoretical and methodological contributions to a particular topic. In this section, the literary works will show YAIMP is similar to the XMPP and the Matrix protocols. While also diving deeper into the synchronous vs asynchronous and client-server vs peer-to-peer protocols. Finally, it will show how the book provides details on its relations with YAIMP.

2.1. Class Textbook

In Kurose and Ross's book on computer networking [1] they go over the peer-to-peer protocol which is seen in the application layer of the TCP/IP model. This layer is an abstraction layer that specifies the shared communications protocols and interface methods used by hosts in a communications network. The YAIMP which is also a P2P protocol follows similarly along with the P2P BitTorrent protocol as described in their book on pages 140-143 (chapter 2). This In this protocol each peer is added to a tracker. A tracker keeps track of all the peers that are still active or in the torrent. This tracker is where the user will obtain the peer list of the other peers to which the user can share files with. The main difference is how the main user in YAIMP is never constantly asking the peers for the list of chunks that they have.

Kurose and Ross also go over in multiple access protocols (pages 462-463, chapter 6) which is seen in the link layer. In this protocol a multitude of different nodes can share channels with one another. If two or more simultaneous transmissions occur between the nodes then interference will also occur. This is very similar to the threading in YAIMP. While testing this protocol it was seen that two threads would occur at the same time. This lead to interference to occur between the two threads. Threading will be described in greater detail in section 3.

2.2. Client-server vs peer-to-peer protocols

According to the article "A Brief discussion About Client Server System" [2], a client-server system is one that can perform the functions of both client and server at the same time to promote the sharing of information between them. The client-server system allows for a file transfer protocol which also the transmission of the file between the client and server. These files could be as simple as text files or get far more complex such as movies, music, or images. When the data gets stored onto the servers there is far greater security to protect the files that were transferred. This is because the servers can control the access and resources to make sure that only the clients that have the correct credentials are able to get in. However, the downside to using the client-server system is that when the server goes down then all the computers connected to it become unavailable to use.

The Peer-to-Peer module is a system that is allows the user to access any file on the computer if that file is in a shared folder with the other peers. According to the article "Peer-to-Peer Research at Stanford" the P2P systems distribute the main cost data across a network of peers therefore enabling the applications to scale without the need for the expensive and powerful servers [3]. P2P also allows for the user to stay connected even if some of the other peers disconnect. There are some challenges, such as the scale of the network and the autonomy of the nodes make it difficult to identify and distribute resources that are available. Also, since there is much less security with this system then some peers who maybe malicious might harm the other peers.

2.3. Synchronous vs. Asynchronous

In Brian Browns "Data Communication" article [4] as well as in Kurose and Ross's book [1] it can be seen that protocols will use asynchronous, synchronous, or a mixture of both systems to transmit data between various users. In an asynchronous system (YAIMP), the data bytes are sent between the sender and receiver by packaging the data. This data is then transported across the transmission link that separates the sender and receiver. This method of transmission is more suitable for slower speeds less than 32000 bits/sec. When the sender has nothing to transmit the line is idle and the sender and receiver are no longer in synchronization. Asynchronization also has no form of error checking which will make debugging extremely difficult for the user.

In a synchronous system greater efficiency is achieved by grouping together characters. The transmission between the sender and receiver is still the same as before except that both parties must be connected at the time of transmission in order to receive the message. Also, the transmissions can achieve higher speeds than asynchronous can. Though this will lead to some problems such as high overhead. The synchronous transmission is also an error checking protocol. Therefore making it easier on the user to find any error that might occur in the system.

For the actual transmission in YAIMP the system is synchronous because both parties must be on for the data to send between the two users. However, it also works asynchronously the because even if one of the users is not active the file will still be sent to another peer and held until that users comes back online. Both of these systems are working in tandem with one another to make the program work.

2.4. Matrix

The Matrix protocol allows individual users to join real time persistent chat rooms. This is because the Matrix, as described in the Matrix.org Foundation and the paper by Nathan Willis [5], is an inter-operable, decentralized, real-time communication over IP. This means that instant messaging can occur between users. These messages that occur in these instant messaging chat rooms are continuously synchronized between every participating matrix server which allows it to be protected against single-point-of-failure problem. This problem is very common among IRC users.

The Matrix protocol's initial goal [6] was to try and solve the problem of fragmented IP communications. This basically allows the user to message other users without caring about what app the other user is on. Which means the Matrix protocol can allow a new server to connect to a new party chat room. When this happens all the participating servers, who were already in the room, will propagate the party room's history to the new server. This allows for all the servers to eventually reach a consistent stat so that every user can have access to the history of the room. However, the protocol will also allow for the party redact certain messages that they do not want the new user to see. This will leave a message's event ID so that it will strip out the unwanted contents without causing problems for the synchronized algorithm.

2.5. XMPP

The XMPP protocol [7] allows for near to real time exchange of structured data between two or more network entities. It was created by Jeremie Miller in 1998, with the goal of providing a free and alternative way to the instant messaging services of the day. In the Internet Engineering Task Force paper about the "Extensible Messaging and Presence Protocol (XMPP): Core" [8] it is said that that the XMPP is typically implemented using a distributed client-server architecture, where in a client needs to connect to a server in order to gain access to the network and be allowed the exchange of files. This process which is gone into greater detail in section 2.2 describes the process where the client connects to the server, exchanges the file and ends connection.

The architecture of the XMPP shows that it is an asynchronous, end-to-end exchange that takes in data by means of direct and persistent streams among distributed network of addressable clients and servers. This basically means that there is an unlimited number of information transactions between the client and server in their transactions. Because of this architecture the XMPP can act similar to an email and use globally unique address based on the DNS in order to route and deliver messages over the network.

The XMPP servers, clients, and programming libraries all support key features of an instant messaging system. This is especially true for one-to-one messaging as well as messaging with larger parties. This is due to the ever increasing XMPP extensions and new features being added. Thanks to these new features the XMPP has been able to show up in WhatsApp and Zoom.

3. TECHNICAL APPROACH

The YAIMP protocol was coded using the Python language since it allows for straightforward implementation of many networking functions, such as sockets and TCP transmission. The following subsections provide a more detailed insight into the design and implementation of YAIMP. The full source code can be accessed and cloned from a git repository at https://github.com/jdmoore97/EE586_FinalProject.git.

3.1. Topology and Peer Architecture

Figure 1 shows an overall view of an example network topology as well as an exploded view of one of the peers. In this particular network, Peer A, B, and D are all interconnected but Peer C is only connected with Peer D. Due to the nature of YAIMP, this setup would still allow for Peer A to send messages to Peer C by Relaying a message through Peer D. This ability to send messages to someone you are not directly connected to is one of the primary motivations for YAIMP in the first place.

Diving into the exploded view of Peer A reveals an overall idea of the architecture of each node. On the receiving end of things, any message that comes in is first checked with a ”Received Cache” to ensure that the message has not been received previously. If the message is not something that has been received before, it is then passed into the ”Received Message” function, where it processed in order to determine what needs to be done with the message. The details of this process are discussed in section 3.4. If it is determined that the message needs to get forwarded to the rest of the network, it is added to the outbox. The outbox is then sent to any peers that are connected based on what is stored in the peer list. The peer list itself is updated anytime a new peer connects to the network or if there is a failure in sending to a previously connected peer. The last piece seen in Figure 1 is the ”Send Message” function. This handles the logic for sending any messages from either the user input or that are waiting in the outbox. It checks the peer list to send all of the messages to any peers that are supposed to be connected. The details of the logic for this are explained in more detail in section 3.5.

3.2. Detecting Peers

The exact method for detecting active peers will vary depending on which medium YAIMP is deployed on. For the development process, the network was simulated on local TCP ports. In the Python code for each peer, there is a list of ”Broadcast ports” that are used to communicate when a new peer is connected. When a new peer connects, it sends a message to every port in the list so that it can be added to the peer list of everyone who is connected. Each person that receives this broadcast then sends an acknowledgement back so that the newly connected peer can add all the previously active peers to its peer list. This method was used because of the nature of testing the protocol on one machine. Since all the hosts are running on the same machine, multiple sockets cannot listen to the same port, which requires a different port for each peer to listen for broadcasts.

3.3. Message Structure

The YAIMP protocol has its own datagram structure that is optimized for its own use. The datagram contains the following sections:

- **Source:** This field holds the address or identifier of the original message sender.
- **Destination:** This field contains the address or identifier of the intended recipient.
- **Data:** The data field contains the ASCII text of the message that is being sent. In other words, this is the actual payload of the datagram.
- **Timeout:** This field contains a float value that is the system time (in milliseconds) that the message should expire at. The timeout process is described in section 3.3.2.
- **ACK:** This is a flag that can be true or false and indicates if the message is an acknowledgement or not.

In the Python implementation, each message is contained in a Message class that has all of the fields as data members. These message objects are created using a `make_msg()` function, which can generate either normal messages or acknowledgements. The specifics of acknowledgement messages are explained in section 3.3.1.

3.3.1. Acknowledgement Messages

Acknowledgement messages have the exact same datagram structure as normal messages, just with the ACK flag set to True. This means that an acknowledgement for a message will have the same source, destination, data, and timeout fields as the original message. Structuring the acknowledgements this way helps in several ways. First, it makes for a simpler code structure since the `make_msg()` function can be used to create both normal messages as well as acknowledgements. Secondly, having the same format for both the ACK and the original message allows for easy comparison to see which message is being acknowledged.

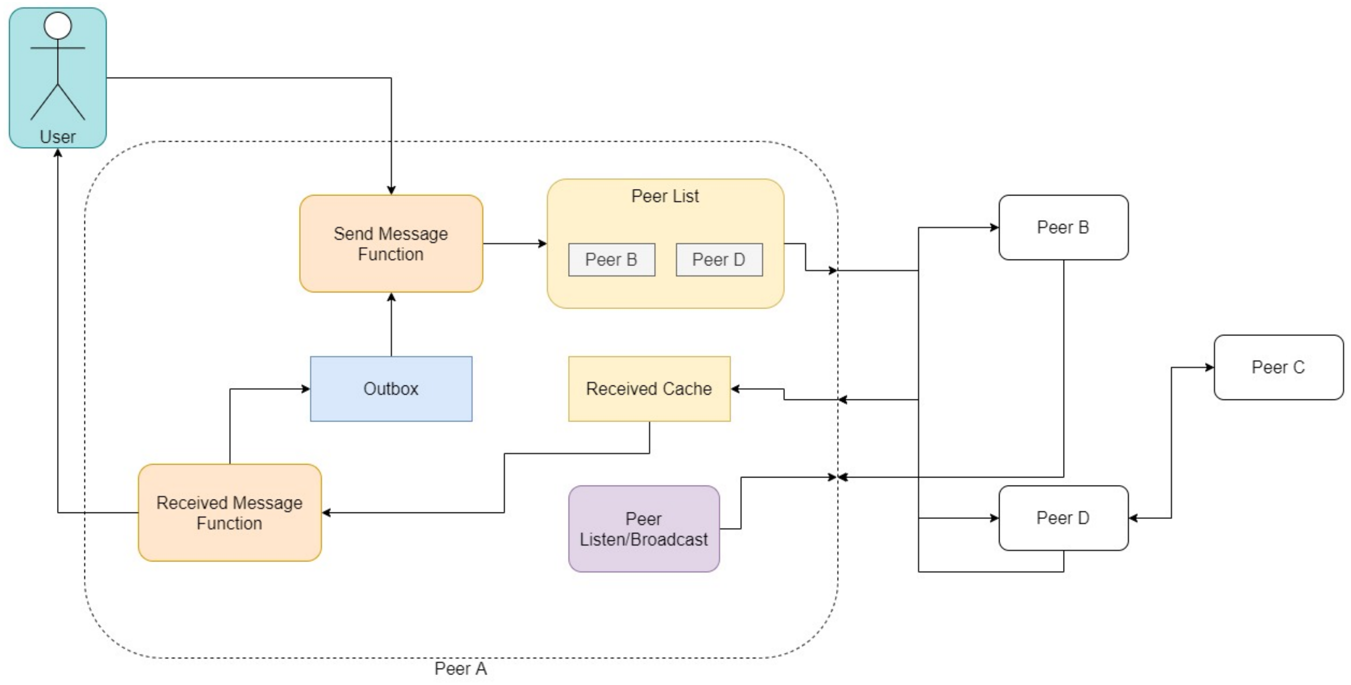

Fig. 1. Network Topology

3.3.2. Message Timeouts

In every message, both normal and acknowledgement messages, the timeout field contains a float value that specifies the system time at which the message should be discarded. In order to update the received cache and the outbox, two functions where created in the Python code: `remove_timeouts()` and `update_cache()`. The `update_cache()` function is used to add or remove a given message from a given cache. It is implemented in such a way to be thread safe and not have concurrency issues. The details of the threading and concurrency are discussed more in section 3.6. The `remove_timeouts()` function simply loops through the given cache and checks the current time with the timeout value of each message. If the current time is greater than the timeout value, then it calls the `update_cache()` function in order to remove the timed out message from whichever cache is being updated. The received cache gets checked for timeouts anytime a new message is received, and the outbox gets checked before any messages are sent to the connected peers.

3.4. Receiving Messages

The logic involved in receiving messages is the largest portion of the YAIMP implementation. The flowchart pictured in Figure 2 shows the overall logical flowchart of what happens when a message is received. It is of importance to note that a few details are abstracted out of the flowchart, but all of the details of the process will be discussed in this section whether they are pictured or not.

The base state of the protocol is waiting for a message to be received. The exact method of ”listening” depends on the specifics of which medium YAIMP is deployed on, very similar to the way listening for peers is as discussed in section 3.2. In the development of YAIMP, it was coded to listen on a local TCP port where the peer would receive any of the messages sent to it. Once a message is received, the code then checks the destination field of the datagram to see if the address is this peer. If this peer is in the destination field, it must then check to see if the message ACK field is true. If it is true, then the message can be discarded because this means that the message is one that this peer originated and it is already in their outbox. On the other hand, if the the ACK field is false then it means that this peer has received a message intended for them so it must deliver the message to the user. After delivering the message, it must then generate an acknowledgement message as described in section 3.3.1, add that acknowledgement to the outbox, and then send the outbox to all of its connected peers.

Going back in the flow a bit, if the destination field is not this peer, then it must still check to see if the message ACK field is true. If it is true, then it must check the source field to see if this peer is the original sender. If this peer is the original sender, then it means that the user’s message has been delivered so they should be notified of which of their messages was delivered to the intended recipient. If this peer is not the original sender, the acknowledgment message should be added to the outbox and then sent to all of its connected peers so that

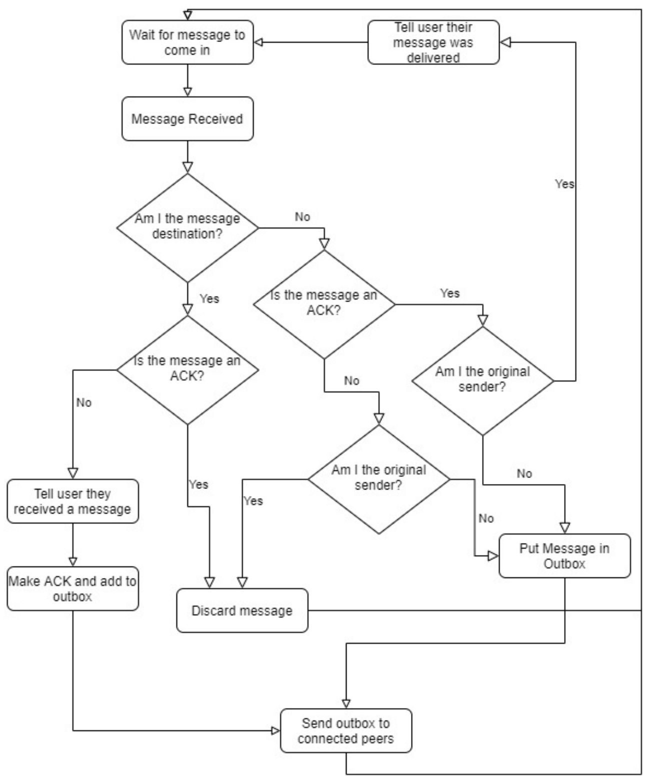
Wait for message to come in
Tell user their message was delivered
Message Received
Am I the message destination?
No
Yes
Is the message an ACK?
Yes
Is the message an ACK?
Am I the original sender?
No
No
Yes
Am I the original sender?
Yes
Yes
No
No
Tell user they received a message
Put Message in Outbox
Make ACK and add to outbox
Discard message
Send outbox to connected peers

Fig. 2. Receiving messages flowchart

the original sender can eventually get the acknowledgment message. If the message is not intended for this peer, and it is not an acknowledgment, then the source field will still need to be checked to see if this peer is the original sender. If this peer is the original sender, then the message can be discarded because it is already in the outbox. If this peer is not the original sender, it should be added to outbox and sent to all connected peers, since in this case this peer is a middleman and needs to simply pass the message along.

3.5. Sending Messages

The logic for sending messages is contained entirely in a single function in the code called `send_msg()`. This function can take in a message (or messages) as an optional parameter, which will be added to the outbox if it is not a duplicate. The function starts by updating the outbox to make sure that any timed out messages are removed. Once the outbox is updated, it then adds to the outbox any messages that were passed into the function. From there, it then loops through all the peers in the peer list and sends the entirety of the outbox to each peer. When sending a specific message, the message object is converted to a JSON and then sent via a socket connection as a JSON string using the `sendall()` function. If there is an issue with connecting to the peer, it is add to a ”dead peers” list. Once all of the peers in the peers list have been sent the outbox (or an attempt has been made it send it), the the list of dead peers is used to update the peer list and removing any peers that appear to be inactive.

3.6. Threading

There are three tasks that the program must accomplish concurrently in order for it to function. It must be able to accept user input to write and send messages, it must listen for incoming messages, and it must listen for new peers. Because waiting for input, both from the user and from sockets, blocks the flow of execution, it is necessary that each of these tasks happen in a separate thread. This is achieved by use of the `concurrent_futures` Python library, which allows a thread to be spawned with for a given function. With this approach, only the worker thread that the function runs in is blocked while waiting for input, rather than the whole program. One issue that arises when using multiple threads is the threat of race conditions. In particular, when there is a shared piece of data, it is difficult for one thread to determine when another thread might modify that data.

For this program, the pieces of data that need to be shared between the threads are the peer list, the received messages list, and the outbox. The `broadcast_listen` function adds peers to the list as they come online, but the `send_msg` function, running in another thread, needs to remove unresponsive peers. Similarly, the outbox and receive list are modified by the `send_msg` function, which can be called by one of two threads — the one that listens for user input, and the one that receives incoming messages (and re-sends them if needed). Because calls to this function could happen simultaneously from multiple threads, there is no way to determine if the list will have the most up-to-date date by the time it is accessed. In order to solve this, there needs to be some form of controlled access to the data. Having not previously worked with much concurrent code, we originally tried to implement this with a global Boolean variable that allowed access to a list by marking it as available to modify. If the value was true, then a thread could set it to false, access the list, and then set it to true again, allowing another thread to access it. Unfortunately, this merely created another race condition, because the order of access to the access variable was non-deterministic. Multiple threads would still try to modify the same piece of data simultaneously, and duplicate messages would be sent and received because the lists were not synchronized across threads. The solution was to apply a Python lock object to each portion of the code that could modified shared data, guaranteeing mutual exclusion for thread access. This does introduce some blocking of execution between threads, as one thread may be forced to wait for another thread to complete execution of the protected section and release the lock. In practice, this wait time is negligible and we did not observe any deadlocks. Once concurrency protections were correctly implemented, there were no issues with race conditions.

4. EXPERIMENTAL RESULTS

A demonstration video of the program in action is available at https://github.com/jdmoore97/EE586_FinalProject/blob/master/demo.mov. It demonstrates some of the various properties of the program, such as the ability for two peers to send a message without being directly connected, and the ability for a peer to join the network after a message is sent to them and still have it delivered.

A brief investigation into the network activity generated by this program was conducted using Wireshark. Because each message was sent without encryption, they could be clearly seen within the payload section of a TCP packet. In order to view just the messages generated by our protocol, we can filter out all packets but the ones with the TCP push flag set (`tcp.flags.push != 0`). We can also track the lifetime of a given packet by filtering out all packets except those whose payloads contain a given time out (eg, `tcp.payload contains "\"timeout\": 1606692346.258498"`), as the timeout value is unique to a given message and persists across forwards between peers and acknowledgements. To see just the original message, we can add a condition for the acknowledgement (in our protocol, not TCP) with `tcp.payload contains "\"ACK\": false"`). From this, we were able to see the differences in behavior as the number of peers changed. When a message is sent by the user from one source peer to one destination peer (a ”message event”), in a network of just those two peers, it results in 2 TCP transmissions. For each transmission there are 3 packets for handshake, one to increase the window size, the pushed data packet, the acknowledgement packet, and 4 packets for teardown). As a proxy, we just measure the number of packets with the push flag set, in this case 2, one for the original message and one for the acknowledgement. Each subsequent new message sent by one of the clients results in 2 more push packets being sent, along with those that were previously generated but have not yet timed out. This continual re-sending ensures that the messages will stay in circulation such that when a new peer joins they have all the messages in the network. When the network is expanded to 3 peers, we can see that the number of transmissions made for each message is 6 for the first one sent, then varies between 4 and 8 additional packets for each subsequent event. The reason the number varies is because the original message is removed from the outbox if the acknowledgement is received. However, a peer may first receive a message and then send the acknowledgement, and the original message will remain in its outbox until the acknowledgement it generated is sent back to it by some other peer. The number of transmissions begins to level off after a minute if the rate of messages created is constant, because the older messages timeout and are deleted by the peers. The behavior is visualized in Figure 3.

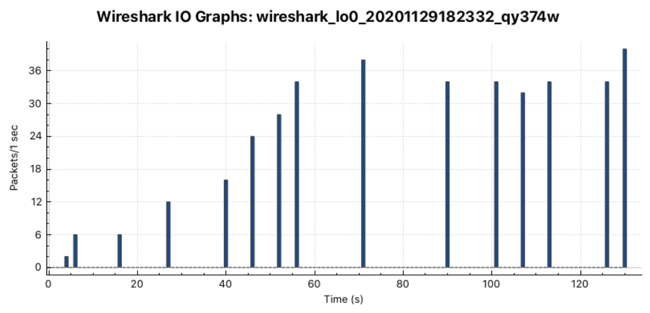

Fig. 3. Program network activity

5. DISCUSSION

There are two main implications of what was discovered during out experimentation with Wireshark. The first is that the logic ought to be modified such that messages are not propagated through the network once an acknowledgement has been issued. As long as new messages and acknowledgments are propagated, new peers will have all the information they need. As it stands, the program is wasting network capacity by passing on messages after their usefulness has expired. There is currently a check to remove the matching message when an acknowledgement is received, but our tests make it clear that this is either ineffective or not sufficient. The effect of the wasted capacity is mitigated by the fact that messages are only retained until their timeout, but this could become a larger issue with more peers, because the number of transmissions needed per message will grow exponentially. In the case of many peers and a limited transmission capacity, it may be also necessary to investigate shortening (or in other scenarios, lengthening) the timeout period for optimal performance.

The other implication is that in order for this to be a practical messaging platform, some sort of end-to-end encryption is needed. Because of the distributed nature of the protocol, it is likely that a message will pass through many other peers for whom it is not intended. Unless we consider this an equivalent to a public chatroom (which may be desirable in some circumstances), the sender and receiver will likely not want their messages to be read by intermediaries. As we saw with Wireshark, without encryption, reading the messages is as trivial as inspecting the TCP packets.

In addition to encryption, there are a few other technical milestones that would need to be achieved before YAIMP would be ready for practical use. As described in section 3.2, our current implementation uses a preset list of local ports as a broadcast channel. We would need to adapt this to some other method in order for messages to be sent to other peers without knowing *a priori* how to locate them. Using Bluetooth to discover and address peers seems like the most promising way to do this; one can imagine a scenario where an application is able to broadcast to all other devices running the same application within range. This would allow YAIMP to fulfill the goals from the motivation set forth in section 1.1.

6. DIVISION OF LABOR

Most of the work on the code was done synchronously with all team members so that everyone was as up to date as possible on the project. That being said, each member had slightly different focuses during meetings:

- **Jonathan:** Focused primarily on the code for sending and receiving messages. Also wrote the Abstract, section 1, and sections 3.1 through 3.5 in this document.
- **Isaac:** Worked on the code to handle threading and concurrency among threads, ran most of the testing on his local machine, and wrote sections 3.6 through 5 in this document.
- **Michael:** Did the bulk of the research in outside sources and scholarly articles to give references and background for the project. Also helped Jonathan with the coding for receiving messages and wrote all of section 2 in this document.

7. REFERENCES

[1] James F. Kurose and Keith W. Ross, *Computer Networking: A Top-Down Approach*, Pearson, 2019.

[2] Arora Jindal Gupta, Narang, "A brief discussion about client-server system," *International Journal of Advanced Research in Computer Engineering Technology (IJARCET)*, vol. 6, no. 3, pp. 2278–1323, 2017.

[3] Mayank Bawa, Brian F Cooper, Arturo Crespo, Neil Daswani, Prasanna Ganesan, Hector Garcia-Molina, Sepandar Kamvar, Sergio Marti, Mario Schlosser, Qi Sun, et al., "Peer-to-peer research at stanford," *ACM SIGMOD Record*, vol. 32, no. 3, pp. 23–28, 2003.

[4] Brian Brown, "Part11: Asynchronous and synchronous protocols," 2020, https://www.angelfire.com/journal/brownrb/datacomm/dc_011.htm, accessed 2020-12-3.

[5] Nathan Willis, "New specification for federated realtime chat," 2020, https://lwn.net/Articles/632572/, accessed 2020-12-3.

[6] "Matrix," 2020, https://matrix.org/faq/#what-is-the-difference-between-matrix-and-xmpp/\%3F, accessed 2020-12-3.

[7] "Xmpp," 2020, https://xmpp.org/ accessed 2020-12-2.

[8] P. Saint-Andre, "Extensible messaging and presence protocol (xmpp): Core," 2020, https://tools.ietf.org/html/rfc6120, accessed 2020-12-3.