CECS 327 Project 2: Peer-to-Peer Distributed Messaging Framework

Section 1

Marco Pacini (026818491), Sarah Phan (027046186), Sarah Santos (027361020)

***Abstract*—This report delves into the idea of decentralized communication through the use of a peer-to-peer (P2P) messaging system utilizing a UDP connection. The architecture is designed to facilitate communication among different peers, each within their own Docker container. The peer.py code file allows these peers to have flexible communication with each other by allowing them to select and change communication partners.**

The use of a UDP connection provides low-latency and operational simplicity, reinforcing the efficiency of our messaging system. By testing within Docker containers, the robustness and isolability is ensured for each peer.

The system's ability to initiate, transition, and terminate between messaging connections emphasizes its adaptability to dynamic network conditions.

This project provides insight on the practicality of a P2P messaging system in real-world scenarios. As peer interactions evolve, the system consistently demonstrates stability and responsiveness. This type of architecture demonstrates its potential as a solution to distributed messaging problems that may exist.

I. Introduction

We decided to use Docker to simulate a peer-to-peer messaging system for multiple reasons. A notable advantage of Docker is its ability to encapsulate applications along with their dependencies into lightweight containers which allows us to simulate virtual machines communicating in a peer-to-peer manner. Docker containers allow for consistency and reproducibility which gives us an easier time with testing our peer-to-peer messaging system without the need to worry about issues with compatibility or different underlying systems if we were to use real machines instead.

The isolation provided by Docker containers is the main reason we were able to successfully orchestrate the peer-to-peer communication; each Docker container is operated independently from one another, but they are able to run on a single host machine which was very useful for our project by allowing us the simplicity of just running Docker on one computer rather than uses multiple different machines in a real life scenario. Furthermore, the isolation ensures that we are able to avoid interference between different instances so that we could more closely mimic the decentralized nature of peer-to-peer systems. Docker containers also start up more quickly and they are resource-efficient for simulating numerous peer nodes concurrently on our single host machine running Docker.

We also took advantage of some useful tools that are provided with Docker such as the Docker Compose capability, which we used to define and manage a multi-container application when we were simulating our messaging system. In other words, this allowed us to define and manage multiple different instances of the peer applications on the same network in a controlled way. This is very important for the dynamic and distributed nature of peer-to-peer networks where nodes may join or leave the network dynamically at any point in time.

II. Methodology

Our approach focused on using the modular nature of Docker's YAML and Dockerfiles to simulate the peer-to-peer messaging system. We started our project by creating the Docker Compose YAML file that is used for simulating two different peers. Each of the services defined in this file represent each of the peers, which includes two different peers that are to be built using our python file and their corresponding Dockerfiles. Then we exposed different host ports in order to facilitate communication individually among each of the peers.

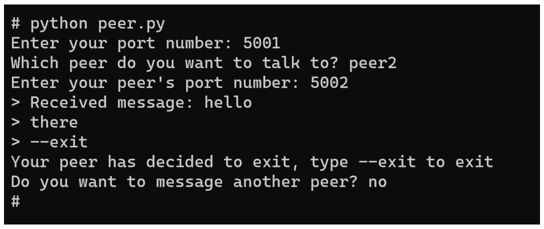
Next, we created the python file representing a peer. The Python file, named peer.py, plays a vital role in orchestrating the communication behavior within each Dockerized peer instance. The script inside this file serves as the main entry point for the Docker containers, defining the logic for sending and receiving messages between peers in the simulated network. The sendMsgs function encapsulates the process of sending messages, utilizing a UDP socket to transmit encoded messages to a specified peer address. On the other hand, the receiveMsgs function continuously listens for incoming messages, decoding and displaying them to simulate the message reception aspect of our peer-to-peer communication system. The interactive communicate function guides users through the setup process, prompting for port numbers, and it also facilitates the dynamic selection of peers for communication. This python file encapsulates the most important functionality required for the simulation, allowing each Dockerized peer to engage in realistic peer-to-peer messaging interactions in a way that allows for control. Next, we created the Dockerfiles that are responsible for assembling the images of the peers and constructing them in our Docker system.

After constructing the necessary files, we incorporated the Docker Compose commands such as Docker Compose -up which initiates deployment of the multiple containers with the ease of a single command; each container representing an instance of a peer in our peer-to-peer network. This gave us a seamless simulation of the peer-to-peer communication. The independence of each container ensured that the interactions between peers were isolated and realistic as if they were real machines.

Finally, the simulation could be executed by connecting to the exposed host ports of each peer. The python script from our file peer.py facilitated message exchange between peers, and the orchestrated Docker setup allowed us to explore various scenarios within the simulated peer-to-peer messaging environment. Our methodology prioritized the aspects of simplicity, reproducibility, and modularity. We made use of Docker's containerization capabilities to provide a simple and flexible platform for simulating peer-to-peer communication systems.

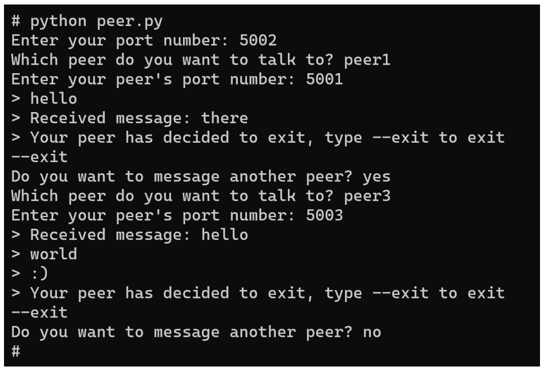
III. Results

Video: https://youtu.be/BR0TlVCTs9A



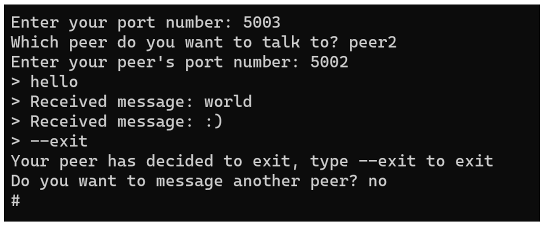
```
# python peer.py
Enter your port number: 5001
Which peer do you want to talk to? peer2
Enter your peer's port number: 5002
> Received message: hello
> there
> --exit
Your peer has decided to exit, type --exit to exit
Do you want to message another peer? no
#
```

Fig. 1. Peer 1 Output



```
# python peer.py
Enter your port number: 5002
Which peer do you want to talk to? peer1
Enter your peer's port number: 5001
> hello
> Received message: there
> Your peer has decided to exit, type --exit to exit
--exit
Do you want to message another peer? yes
Which peer do you want to talk to? peer3
Enter your peer's port number: 5003
> Received message: hello
> world
> :)
> Your peer has decided to exit, type --exit to exit
--exit
Do you want to message another peer? no
#
```

Fig. 2. Peer 1 Output



```
Enter your port number: 5003
Which peer do you want to talk to? peer2
Enter your peer's port number: 5002
> hello
> Received message: world
> Received message: :)
> --exit
Your peer has decided to exit, type --exit to exit
Do you want to message another peer? no
#
```

Fig. 3. Peer 1 Output

IV. Conclusion

Analyzing our output, we are able to see that the peers successfully communicated with each other in an efficient manner. Allowing our peers to exit one conversation and join another demonstrates the system's flexibility in peer demands.

When it comes to the characteristics of a P2P system, we were able to incorporate how peers have the same capabilities and responsibilities through the implementation of how each peer runs the same python script allowing them to function in the same exact way. Additionally, there was no central server managing the communication between peers. Peers were allowed to freely and directly communicate with each other without having to pass the information along to a server that would then send a copy of that message out to the target peer.

One caveat we were unable to address was not having anonymity between the peers. As it was necessary for peers to know the port number of their communication partner as per our design, this goes against the P2P system characteristic of having some degree of anonymity. In the future, we would like to explore having a lookup table that would allow us to store peer port numbers. This will provide two benefits. Not only would peers be unburdened with having to know the port number of their peers, but this will also allow for peers to have more privacy when it comes to their port numbers.

V. Contributions

Marco Pacini: 33%
Sarah Phan: 33%
Sarah Santos: 33%

VI. References

[1] P. Hintjens, "8. A framework for distributed computing," Introduction, https://zguide.zeromq.org/docs/chapter8/ (accessed Nov. 24, 2023).

[2] G. Thomas, "A beginner's guide to docker - how to create a client/server side with Docker-compose," freeCodeCamp.org, http://www.freecodecamp.org/news/a-beginners-guide-to-docker-how-to-create-a-client-server-side-with-docker-compose-12c8cf0ae0aa/ (accessed Nov. 24, 2023).

[3] "The world of peer-to-peer (p2p)/building a P2P system," Wikibooks, open books for an open world, https://en.wikibooks.org/wiki/The World of Peer-to-Peer (P2P)/Building a P2P System (accessed Nov. 9, 2023).

[4] "UDP Peer-to-Peer Messaging with Python" Youtube, youtu.be/IbzGLtjmv4?si=RxhA0VPAFfgDkEx (accessed Nov. 29, 2023).