RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

553: Internet Services

Hive-Talk

A Secure Distributed P2P Chatting Application

PROJECT REPORT

Team Members:
Bala Sai Lokesh Kodavati- **BK576**
Sai Chaitanya Chaganti - **SC2482**

05/07/2023
Prof. Srinivas Narayana

1. Abstract

Hive-Talk is a decentralized chat application primarily based on IPFS (Interplanetary File System) technology. It offers a steady and green manner for customers to speak with every other in real-time, without counting on relevant servers or continual TCP connections. The chat is offered handiest through a MetaMask ID without compromising privacy.

The existing communication systems depend upon centralized servers, which can be costly to preserve and prone to security breaches. The Hive-Talk undertaking leverages IPFS and OrbitDB to save all information locally inside the browser, supplying an extra scalable and secure alternative to centralized systems.

This application makes use of EventEmitter to set up peer-to-peer connections between customers, permitting them to communicate freely without any intermediaries. The chat UI is displayed on both users' browsers, permitting them to chat in real-time. Additionally, the application makes use of Vue.Js for a responsive and UX driven interface and Tailwind CSS for a greater stylish appearance.

The application’s structure is designed to be absolutely decentralized and secure, offering users full management over their information and communique. By using contemporary technology which includes IPFS, Blockchain and OrbitDB, the Hive-Talk application highlights the power of decentralized and secure communication systems.

2. Introduction

The demand for connecting with each other across the world has been ever-growing. But the effort by the companies handling the applications in terms of privacy and security is quite non-existent as most of these applications are owned by monopolies and user data is often used without permission. This is due to centralized servers. Therefore, the requirement for a decentralized system with no centralized monopoly is ever-present. Therefore, one thing we can do is utilize a network in which users need not depend on central servers.

3. Proposed Solution

The Hive-Talk project was developed in response to the limitations and concerns of existing communication systems. These systems rely on persistent TCP connections between users and servers to facilitate communication, which is expensive and not scalable. As these systems grow, more machines need to be added to handle the connections, increasing costs and electricity usage. Additionally, centralized system architecture is vulnerable to security attacks, as all information is stored in central servers.

Moreover, these centralized communication systems have limitations on privacy and data security. All correspondence goes through the server of the corporate owning the messaging app, and the company can dictate its rules, block messages on a particular subject, or prohibit the transfer of certain files. The company may also be subject to government pressures and may be asked to disclose users’ correspondence or impose certain restrictions upon their request. This centralized approach to communication hinders privacy and freedom of speech.

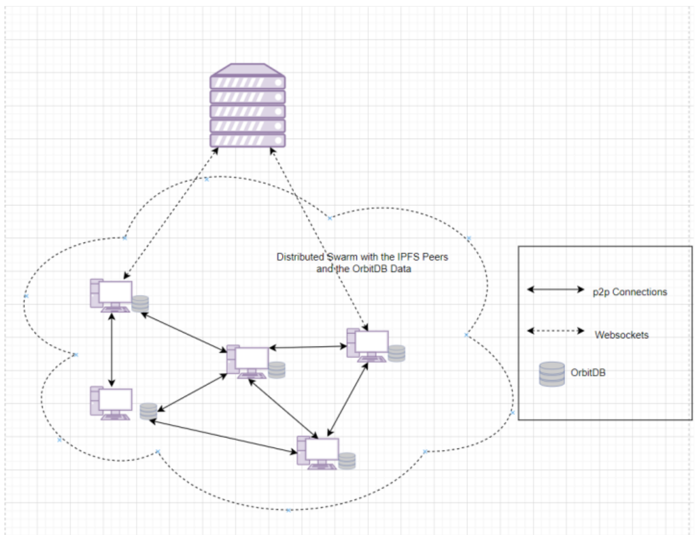
The Hive-Talk project aims to address these limitations and concerns by providing a secure, scalable, and decentralized communication system. By leveraging IPFS and OrbitDB, it eliminates the need for centralized servers and persistent TCP connections, reducing costs and increasing security. The use of EventEmitter allows for peer-to-peer connections between users, enabling them to communicate freely without intermediaries.

Furthermore, the Hive-Talk project provides a more secure and private alternative to existing communication systems. The chat is accessible only via a MetaMask address, ensuring user privacy. By storing all data locally in the browser, users have full control over their data, and no central database or server can access it.

This primarily involves the usage of 2 components and 1 pseudo-component:

- **User (Peer) Devices**: These act as the user and server because the Orbit DB nodes reside locally on the peer nodes and are replicated within peers.
- **A WebRTC star server** which handles WebSocket connections initiates a handshake between the peers, allowing them to exchange information about their network addresses, capabilities, and protocols. The process of establishing a connection through the signaling server involves the use of the Session Description Protocol (SDP). SDP is a text-based format that describes the media components, transport protocol, and other network

configuration details required for the communication session. In summary, the signaling server acts as a facilitator between peers, helping them establish a direct peer-to-peer RTC Data Channel by exchanging SDP messages. The SDP protocol describes the media components and network configuration details required for the communication session. The use of a WebSocket connection allows the signaling server to handle real-time communication between peers, ensuring reliable and efficient communication.



Proposed System

In summary, the Hive-Talk project aims to solve the problems of expensive, centralized, and insecure communication systems by providing a secure and decentralized communication system that prioritizes privacy and data security.

4. Technologies Used

The Hive-Talk project uses a range of technologies to create a secure, scalable, and decentralized communication platform.

IPFS (Interplanetary File System): IPFS is a distributed file system that provides a peer-to-peer mechanism for storing and sharing hypermedia in a partitioned file system. IPFS is used in Hive-Talk to provide persistent storage, eliminating the need for use of centralized servers, and minimizing system costs and vulnerabilities. It uses content addressing to assign a unique identifier to each resource. This identifier is usually a hash. When content is added to IPFS, it is stored and can be accessed by connected peers. This allows multiple peers to respond simultaneously during data retrieval, leading to better performance for high-latency networks. Additionally, the data can be verified.

OrbitDB: OrbitDB is a serverless, distributed database built on top of IPFS. All the data is stored in the local browser storage and synchronized with other peers on the network. Usage of OrbitDB eliminates the need for centralized databases, providing greater privacy and data security.

OrbitDB provides a series of DB types such as

- **Log**: This is an immutable (append-only) log with traversable history. Suitable as a message queue.
- **Feed**: A mutable log with traversable history. Entries can be added and removed.
- **Keyvalue**: A key-value database.
- **Docs**: A document database to which JSON documents can be stored and indexed by a specified key. Useful for building search indices or version controlling documents and data.
- **Counter**: Useful for counting events separate from log/feed data.

In our case we are utilizing Docs type so we can easily store and retrieve the user JSON data.

OrbitDB utilizes a conflict-free replicated data type (CRDT) data structure to maintain eventual consistency, allowing operations to occur at separate times without coordination, with the expectation that they will eventually synchronize. When interacting with an OrbitDB database, we are dealing with a snapshot in time, as distributed databases operate both online and offline. However, it requires at least one peer to persist the database to prevent data loss upon disconnection. Data is interconnected through content addresses instead of the centralized web's location-based addressing, where the application code runs on a centralized server.

Ethereum Blockchain: Here MetaMask is used to create user identity and the underlying network used by it is the Ethereum Blockchain which provides an elevated level of security due to its decentralized nature and use of cryptographic algorithms.

Vue.js: Vue.js is a progressive, lightweight framework for building user interfaces. It is used in Hive-Talk, making the interface responsive and dynamic.

Tailwind CSS: Tailwind CSS is a first-of-its-kind CSS framework that provides predefined methods and classes to create responsive and modern layouts. This is used in Hive-Talk for its elegant layout and responsive system.

Random User API: Hive-Talk uses the Random User API to generate random usernames for those who do not have one. This allows users to remain anonymous despite being able to communicate with others.

Event Emitter: Enables asynchronous, event-driven applications that can process multiple events and callbacks simultaneously. It is used in Hive-Talk to trigger actions when a specific event occurs, such as when a new message is received or when a user joins or leaves a conversation.

Overall, with the use of these technologies, Hive-Talk creates a decentralized and secure communication system that prioritizes privacy and data security. By eliminating the need for centralized servers and databases and using peer-to-peer communication, Hive-Talk provides a cost-effective and scalable solution to the limitations of existing communication systems.

5. Prior Work

There are countless applications that are used daily to enable communication between users, such as WhatsApp, Messenger,etc. These are based on a Client-Server architecture and based on keeping a TCP connection between users and server to facilitate communication between end users.

This approach, while effective, is not scalable, and the users are bound to increase with time, and the existing approach would mean additional cost to the server hosts.

This can be avoided using a decentralized server; there are a few applications that provide this functionality, such as Briar, Matrix, Tox, etc.

These typically incorporate secure encryption standards for security. For instance, Tox uses NaCl encryption library, Matrix uses TLS, and Briar, on the other hand, also uses Tor Network in addition to this so that the source or destination cannot be traced back. Some cons of these approaches are that the network bandwidth is hogged by message flooding.

6. Risks Involved and Challenges Faced:

The Version incompatibilities between orbit DB and IPFS gave a really hard time as it was hard to debug incompatibility as it happened in the official code of OrbitDB and IPFS instead in the actual application.

Understanding how swarm addresses with the underlying WebRTC work.

And as mentioned previously, since this relies on the Ethereum network, the latency and throughput are heavily dependent on the nodes to validate the transactions.

7. Design

UX based development: The application is designed to be responsive and easily navigable.

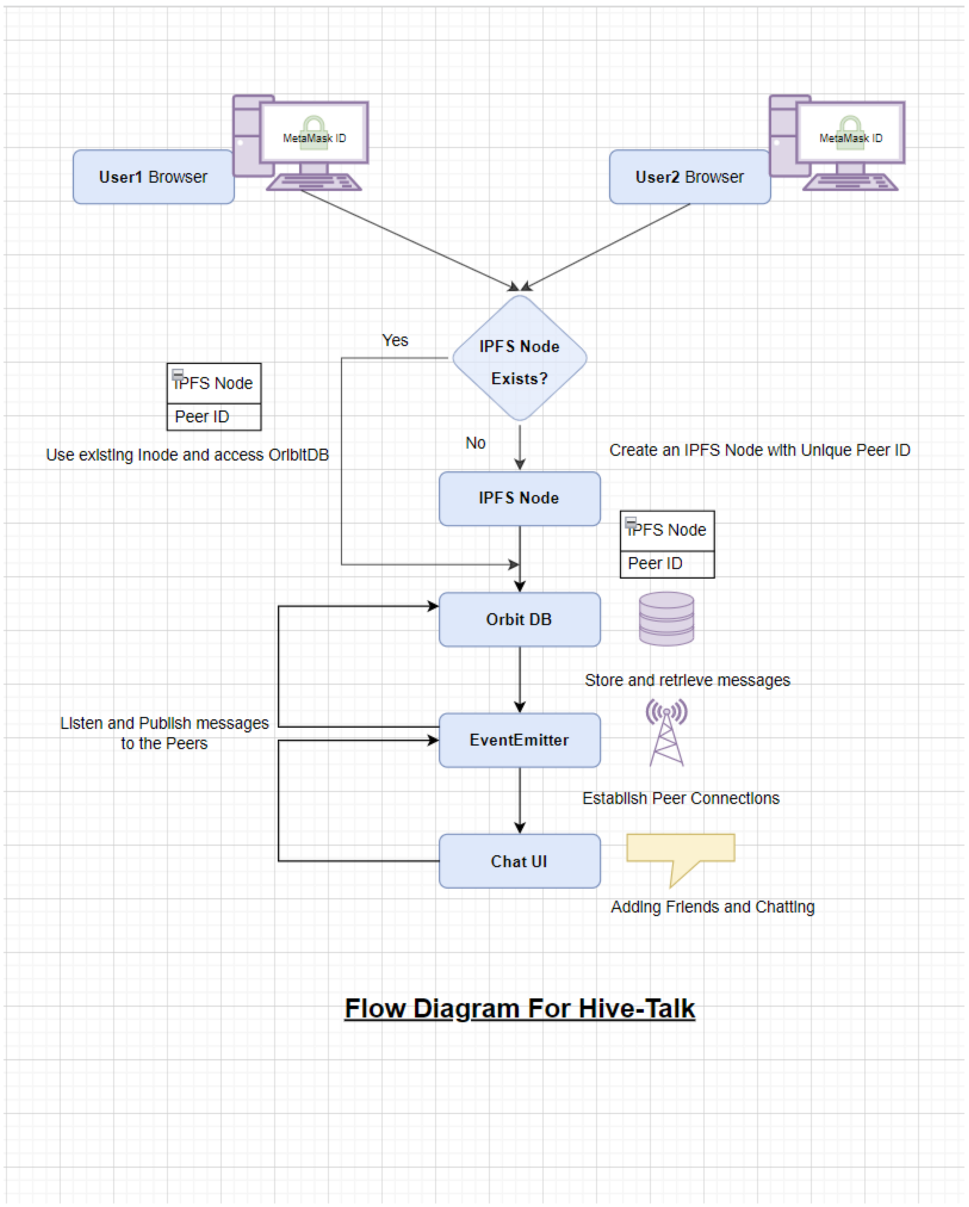
The Registration Page consists of a user form with the Username(nickname) which is randomized by default, Avatar which can be changed if wanted and the ID for the user is filled from MetaMask information provided by browser.

If the user is already present it redirects to the app directly or else, it prompts to register an account.

Secure Messaging: The technologies used to underlie the application like IPFS and OrbitDB have all the information within them enabled with content addressing which means the content is addressed using cryptographic hashes and therefore any changes done to the information would result in change of hash and therefore any kind of MITM attacks can be avoided. Also, every user node in the IPFS network has a unique public-private key pair. Therefore, only authorized nodes can access the data.

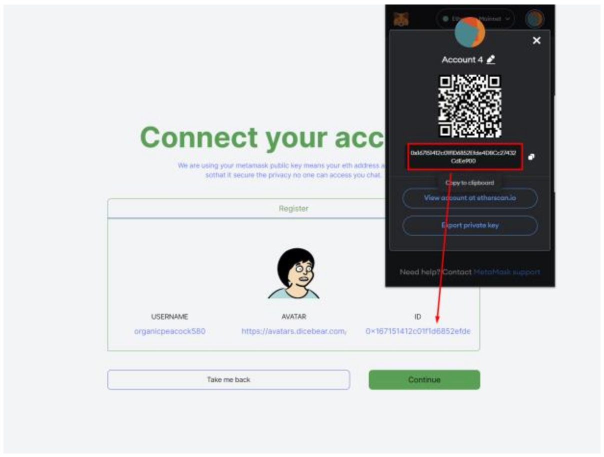
Then the application follows the below process to enable users to have communication

Flow Diagram:


MetaMask ID
User1 Browser
MetaMask ID
User2 Browser
Yes
IPFS Node Exists?
IPFS Node
Peer ID
Use existing Inode and access OrbitDB
No
Create an IPFS Node with Unique Peer ID
IPFS Node
IPFS Node
Peer ID
Orbit DB
Store and retrieve messages
Listen and Publish messages to the Peers
EventEmitter
Establish Peer Connections
Chat UI
Adding Friends and Chatting
Flow Diagram For Hive-Talk

The architecture diagram shows the interaction between the different components of the Hive-Talk project.

The primary step of the process for a user that connects to the app involves of them Creating a MetaMask ID and connecting it to the website, only then, the UserID field is populated, and the app allows them to register. This is one way to ensure that there are no spam accounts and bots.This is done using the ethereum.selectedAddress in the meta view page we define in one of our views



MetaMask ID being used for the User ID

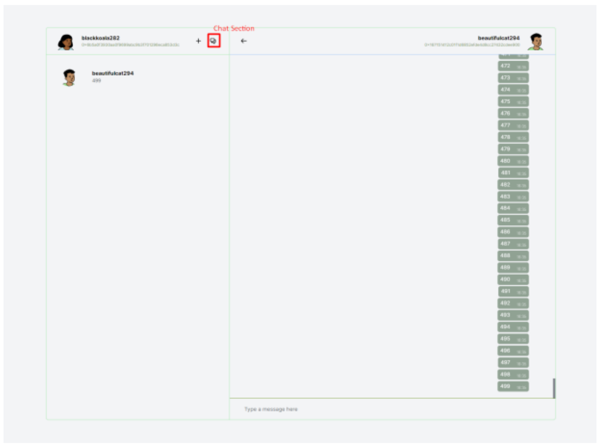
Once the User form has all the information necessary, the next step is to create an IPFS node for entering the FS with an identity. When we create an IPFS node, we are provided with a Unique ID which is Base-58 encoded key and usually starts with a "Qm" or a "12". This is called our Peer ID. This is generated depending on the config and repository we provide while creating the IPFS node.

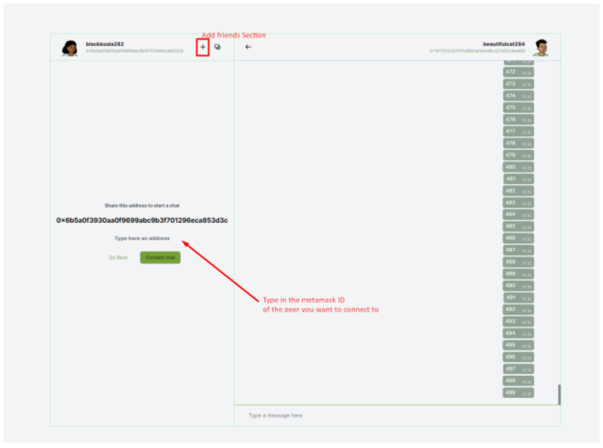
This config file consists of details like the Swarm addresses, bootstrap nodes, and any authentication methods that we want to use, the swarm address which provide is nothing but the webrtc star signaling server that allows the user nodes to identify each other as peers. In a WebRTC P2P network, the peers may not know each other's IP addresses, so they need a signaling server to exchange the necessary information for establishing a direct connection. WebRTC Star servers act as a mediator to establish this connection and enable secure and reliable data transfer between peers.

Once we have our Peer ID, we then create an OrbitDB database main DB which is the root Database of our application and contains all the entries about users i.e., the user ID, the avatar and the MetaMask ID used for registration. Then a DB for each user which internally contains the chat DBs with the connected peers is created for the user and at last the user is navigated to the chat.

There is also a case when an existing user loads the application and since they already are registered, they can directly access chat if at least one peer exists and was replicating the OrbitDB data.
The Chat UI consists of 2 pages:

- The Chat Section
- The Add Friends Section





The **Chat Section** contains the lists of peers added as friends and one can start chatting with them by clicking on the respective block in the chat section.

The **Add Friends Section** contains the field to enter the MetaMask of the peer you are trying to add as a friend. Once provided a chat specific to you the peer gets created in yours and his Chats DB and finally the peer gets added as your friends into your friends list which you can access in the chat Section.

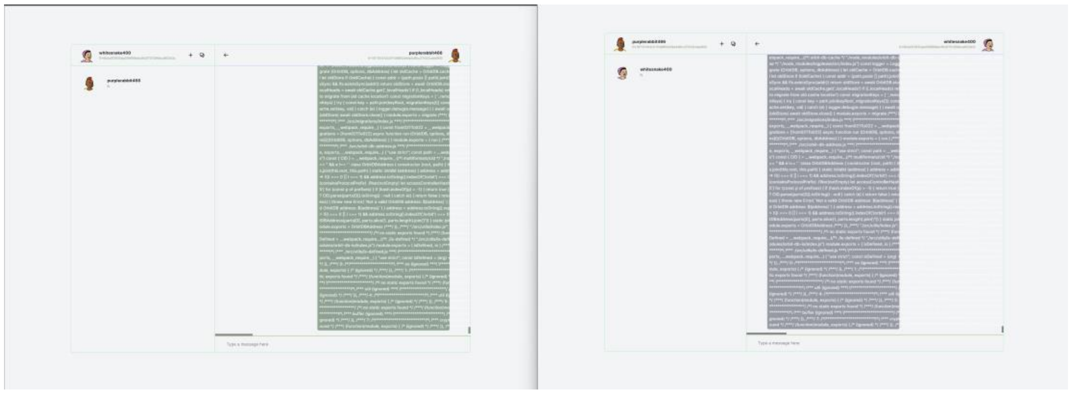
Once the user accesses any of the chats with the other peers a p2p connection is started between the two and upon filling and sending a message the message is populated in the topic and is published to the channel. The subscribed peer then receives the message by pulling them from the OrbitDB that has been created in the User1 and User2 chatDBs.

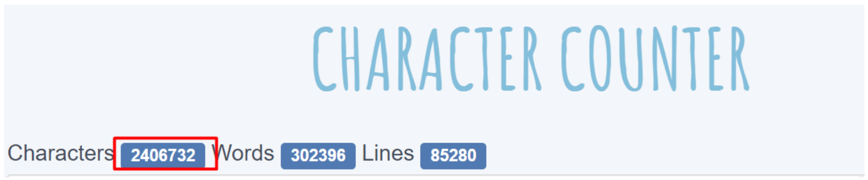
8. Results and Discussion

A Few Metrics that can be taken into consideration for the Evaluation are:

Reliability: The Application should be able to perform its intended functions in a consistent and precise manner without errors or failures. That is, the messages should be delivered to the users without being dropped midway through.

Test 1: Sent a message with **2406732** characters and the Application did not drop the data and successfully sent it

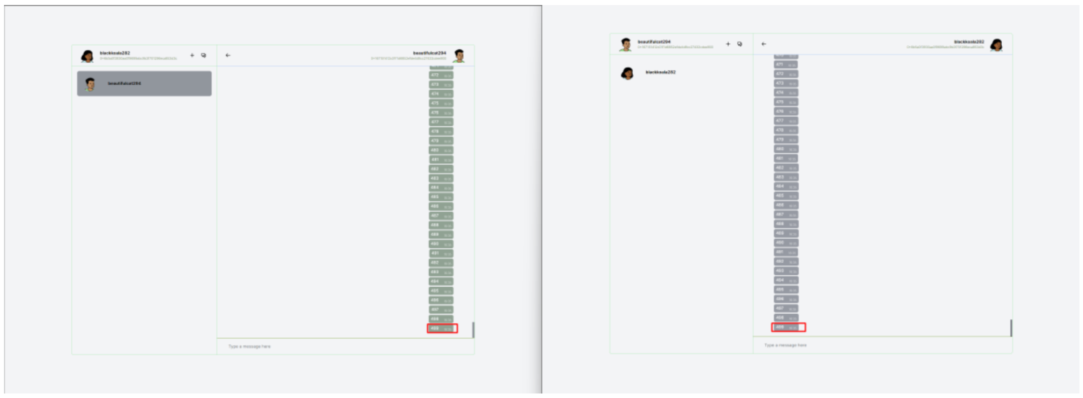




CHARACTER COUNTER

Characters 2406732 Words 302396 Lines 85280

Test 2: Sent 500 messages at a time with data starting from 0 to 499 by calling the OnClick() function sendMessage in the chat view.

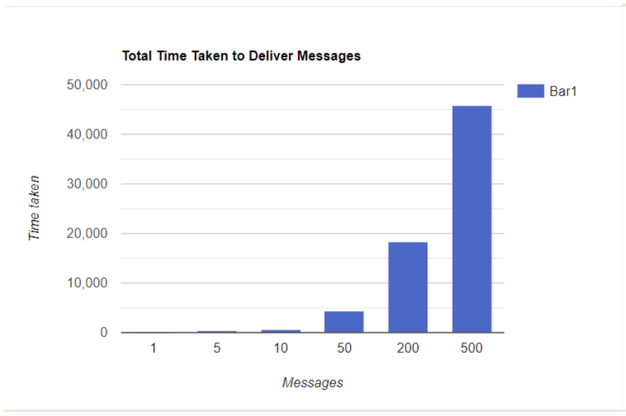


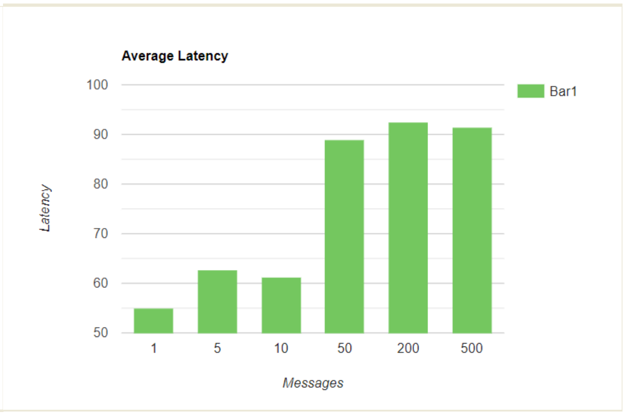
500 messages were delivered to the 2nd user without dropping any data.

Latency: This can be calculated by measuring the time between the instant the user who is sending a message initiates a sendMessage() async function till the destination user gets the complete message that is sent by user 1.

The Test Scenario to latency was to change the number of messages sent at time to see how the system performs under heavy load.

Number of messages sent at a time	Time Taken in total	Average latency
1	55 ms	55 ms
5	314ms	62.8 ms
10	613ms	61.3 ms
50	4449ms	88.98ms
200	18403ms	92.515ms
500	45730ms	91.46 ms



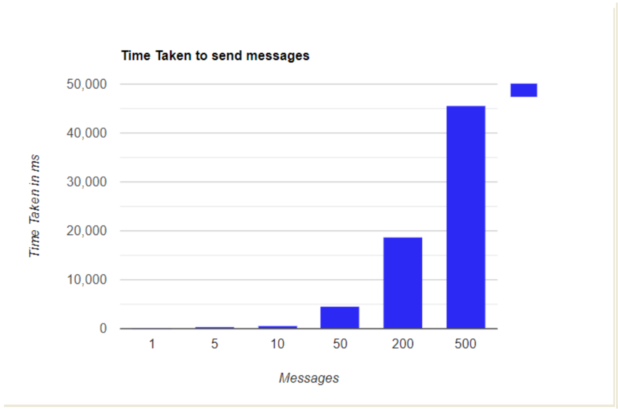


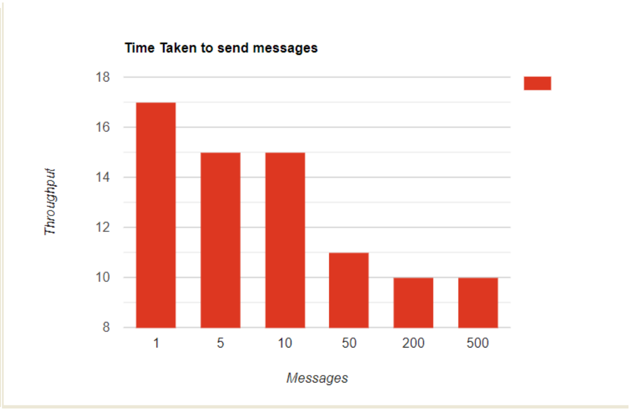
We observed that as the number of messages sent at one time increased the average latency for delivery is increasing gradually. This is because of the limitations of the network bandwidth which is getting flooded by the topic.

Throughput: This scenario can be tested by sending huge file sizes and can be compared against time taken for number of such messages sent in total.

Test 1: In this scenario we are send **1-to-3-character** sized messages

Number of messages sent at a time	Time Taken in total	Throughput(messages/sec)
1	58 ms	17
5	332ms	15
10	645ms	15
50	4587ms	11
200	18702ms	10
500	45550ms	10





Test 2: In this scenario we are send **2406732-character** sized message

Size of the message	Time Taken
2406732	5593ms

9. Conclusions

Firstly, we emphasize that the Hive-Talk application is a novel solution to solve the problem of centralized communication systems. Leveraging the distributed nature of IPFS and OrbitDB, Hive-Talk provides users with a secure, private, and efficient way to communicate with each other in real time. It also eliminates the need for a central server and enables users to communicate directly with each other. Hive-Talk ensures that the communication is secure and cannot be accessed or manipulated by any third party.

Furthermore, the blockchain technology adaptation combined with the use of IPFS and OrbitDB, Vue, Tailwind CSS, Random User API, provides efficient user interaction, user-friendly access to the Hive-Talk application.

The robust design ensured that the application is responsive and can be easily accessed from any device. Our project implementation was successful, with all desired features and functionality added to the application. We thoroughly tested the application and found and fixed all the bugs and errors.

Finally, we conclude that the Hive-Talk application has the potential for further development and growth. The use of blockchain technology and distributed storage provides a good foundation for building more advanced and secure communication applications such as focusing on integrating additional features such as audio and video calling, file sharing and group chat functionality.

10. Acknowledgments

We would like to thank Prof. Srinivas Narayana for his invaluable guidance and support throughout this project. His expertise and knowledge in Internet Design have been instrumental in understanding the key concepts. We built and developed an effective solution for the problem of Decentralized Chat Application with these fundamentals.

11. References

[1] orbit-db

[2] ipfs

[3] JS IPFS

[4] pushing-the-limits-of-ipfs-and-orbitdb

12. Appendix

[1] Code base: https://github.com/loki-jarvis/hive-talk

[2] Project Video Walkthrough