The Hong Kong Polytechnic University
Department of Computing

COMP4913 Capstone Project
Report (Final)

dCollab: A Decentralized e-Learning Collaboration Platform

For Online Preview Only

Author:	@CrabAss
Supervisor:	Dr Henry Chan
Date:	23 May 2020

Abstract

Collaboration plays an important role in the university education. By encouraging the students to collaborate on specific tasks, they will become better team players in their future careers. Today some successful e-learning web applications like *Blackboard*, *Kahoot*, *Padlet*, and *Pear Deck* already exists, but these applications are all managed on a centralized basis, which may lead to multiple drawbacks including high maintenance costs, the vulnerability to network errors and privacy concerns. In this report, the author describes a decentralized web application which will help the students to collaborate on their academic coursework. This application consists of the features of online group chats and collaborative Kanbans.

Table of Contents

List of Figures and Tables

Background

This chapter introduces the necessary background required for this work. The concepts of blockchain, smart contracts and *Ethereum* will be briefly discussed. This will be followed by an overview of the *Whisper* protocol which enables the peer-to-peer communication between the nodes on the Ethereum network.

- **Blockchain**

  Blockchain serves as a distributed database which is stored and maintained by a peer-to-peer network. Every node on the network stores a partial or full copy of the blockchain. Blockchain consists of blocks which records the transactions on the network. Each block on the blockchain contains a nonce and a hash value of the parent block, which increases the computational cost to tamper with the early blocks. As the number of nodes on the blockchain network grows, the blockchain database would become more difficult to be tampered with [1]. Some important characteristics of blockchain are as follows [2]:

  - **Decentralized**

    In a typical transaction system, a trusted third party (e.g. a bank) is always required to validate the transactions and protect the database of transaction records, which would make the transaction system centralized. The trusted third party becomes a potential single point of failure in this centralized transaction system. By contrast, the validation process in blockchain network is decentralized. A group of "miners" will validate the transactions, and the consensus algorithms would ensure that these anonymous validators could be trusted.

  - **Anonymous**

    Users can generate as many addresses as they want to interact with the blockchain network. The addresses are randomized and not linked with personal identities.

  - **Persistent**

    As mentioned previously, the decentralized nature of the blockchain database and the high computational cost to produce a new block make the transaction history on the blockchain extremely difficult to be tampered with.

- **Ethereum**

  The concept of "smart contract" was first introduced in 1997 by cryptographist Nick Szabo. Szabo defined a smart contract as a collection of digitalized promises, and the corresponding protocols are also needed for different parties to operate on these promises [3]. This idea has not been realized and popularized until the emergence of the consensus algorithms and the distributed ledger technology. Ethereum is an open source blockchain implementation which introduced a comprehensive solution to host and run sophisticated and customized smart contracts on its network thanks to its *Ethereum Virtual Machine* (EVM) which is Turing-complete. EVM acts as a runtime environment for the Ethereum smart contracts, running on every node of the Ethereum network. The smart contracts could be coded by some high-level scripting languages like Solidity, compiled down to EVM bytecode, and deployed on the Ethereum network for further execution [4]. Ethereum is currently one of the most popular development platforms for the programmers to develop and deploy their own *decentralized applications* (dApps) [5]. The decentralized applications based on Ethereum has already covered various areas such as gaming, gambling, live streaming and so on.

- **Whisper Protocol of Ethereum**

  Whisper is a peer-to-peer identity-based messaging system for decentralized applications running on the Ethereum network [6]. In Whisper, the communication between peers utilizes the *ÐΞV-p2p Wire Protocol*. The instance of a decentralized application is able to create an identity which is required to send or receive messages in a node connected to Whisper. Messages on Whisper are encrypted and transmitted through ÐΞV-p2p, the underlying protocol of Whisper. The messages can be encrypted either asymmetrically using *ECIES* or symmetrically using *AES-GCM*. The decentralized applications on Ethereum can interact with Whisper protocol through *web3* package.

Related Works

This chapter provides some related works with decentralized messaging and collaboration platforms. The previous researches on decentralized messaging using Bluetooth and the blockchain will be introduced. The attempts to utilize the peer-to-peer wiki engines in classrooms are also covered in this chapter.

- **Decentralized Messaging**

  As for the decentralized instant messaging applications for end users, *Bluetooth* is a commonly used technology as the medium for message transmission. In [7] and [8], Bluetooth chat messengers on Android were developed and discussed. Although it has a high availability in the situations where the internet is unreachable, the usage scenario is limited due to the short range of the Bluetooth signal (30 or 150 ft, depending on the hardware). This problem was partly solved by Gogy and Thomas [9]. They proposed and implemented an algorithm for intermediate nodes to relay the messages to their destinations. This algorithm would enable two users who are out of the direct Bluetooth range of each other's device to communicate. This solution still requires intermediate devices which are running the messenger software and within the direct Bluetooth range of the adjacent relay nodes. Therefore, it will become impractical if the existence of the relay nodes is not guaranteed.

  Because of the decentralized nature of blockchain network, some researchers suggest it as the medium to store and transfer the instant messages. Partala [10] introduced a method to embed encrypted messages to the blockchain by creating as many payment addresses (whose least significant bits could form the encrypted text) as the length of the text, and send transactions to one of the addresses sequentially every time a new block is published. This mechanism could be costly and time-consuming because of the high transaction fee and the long time to produce a new block in some major blockchain networks like Bitcoin. Some other researchers suggested taking advantage of the existing anonymous peer-to-peer communication protocols like Tor to implement a decentralized communication platform [11]. But these protocols may not fully guarantee the anonymity and may fail to prevent the networks from flooding attacks [12].

  Some researchers tried to design and implement decentralized messaging using the Whisper protocol. Lee's team [13] built a messenger and coupon exchanging mobile application based on Whisper. For each session, the application requires to exchange a topic between the recipient and the sender, which would happen outside the application. This could be a potential drawback of this implementation. In [12], the Abdulaziz's team also implemented a decentralized messaging application based on Whisper. They claimed that it is able to preserve the complete anonymity of the participants.

- **Decentralized Collaboration Platform**

  There are several researches in decentralized collaboration which focus on the wiki engines. Compared with the traditional client/server (C/S) solutions like Wikipedia, a peer-to-peer wiki system has lower costs in maintenance and a higher availability. Mukherjee, Leng and Schürr [14] proposed a purely p2p-based wiki engine called *PIKI*. It has a collection of features such as version control, simultaneous editing and text search. Davoust's team [15] conducted an experiment in Carleton University to see how a peer-to-peer wiki system could promote the teaching and learning process in classrooms. They used another wiki engine called *P2Pedia*, and found that it allowed the students to solve the proposed problems on their own while being exposed to each other's works.

Problem Definition

Collaboration plays an important role in the university education. By encouraging the students to collaborate on specific tasks, they will become better team players in their future careers. As smart devices are widely used by students and lecturers in universities nowadays, more and more educators have been wondering how to utilize these smart devices to improve the experience of teaching and learning.

Today, some successful e-learning web applications like *Blackboard*, *Kahoot*, *Padlet*, and *Pear Deck* already exists, but these applications are all managed on a centralized basis. These centralized web services usually have a high cost in maintenance which may lead to an unaffordable pricing for students. Furthermore, these centralized services may become single points of failure, which may significantly affect the progress of the students' coursework. In this project, the author will attempt to build a decentralized web application which will help the students to collaborate on their academic coursework.

Objectives

This project aims to implement a decentralized web application for university students to collaborate on their academic coursework. The targeted users are the university students. More specifically, this decentralized web application will enable the users to:

- create a team and invite other members to the team,
- chat with the members in the same team,
- review the progress of the tasks in a team, and
- create, modify, toggle and delete any task in a team.

The proposed system consists of the following components:

- **Personal Identity Configuration**

  The users can review and modify the information of their personal identity like nicknames.
- **Group Configuration**

  The users can join and leave groups. The groups are identified by the group ID.
- **Instant Messaging**

  The users can send and receive messages within a team. Plain texts and emojis will be supported in the messages.
- **Shared Kanban**

  Kanban is a collaborative board. Contents are written in *cards*, and these cards are grouped into multiple *lists*.

  In a typical use case, a Kanban usually has three lists whose titles are "To Do", "Doing" and "Done". Each task is specified in a separate card, and the users can assign this card to any of the lists according to the status ("To Do" / "Doing" / "Done") of the task. The users can add to-do items to a card.

  One of the existing applications which are not based on blockchain is Trello.

The system component dependency diagram is shown in Figure 1.

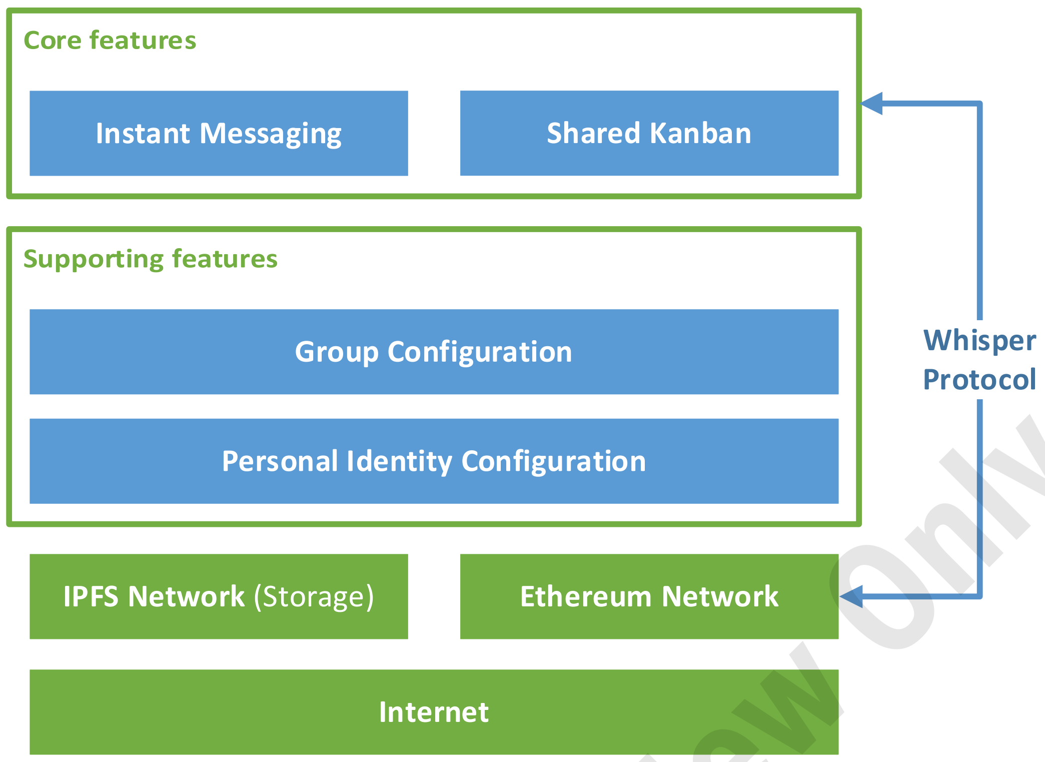


Figure 1: System component dependency diagram

System Design

- **Overview**

The technology stack of the proposed application is shown in Figure 2. The application could access the Whisper protocol through the Ethereum JavaScript API, and the Whisper protocol is implemented on Ethereum networks. However, the Ethereum network cannot efficiently store the frontend resources of the web application. IPFS is adopted as the decentralized storage solution in this application. Users can download the frontend resources required by the web application from IPFS Gateway (http://gateway.ipfs.io/ipfs/). The gateway provides an easy way for users to access the IPFS decentralized network without installing the IPFS client. This web application still needs network access, but no centralized server is involved when the users are loading and using this web application.

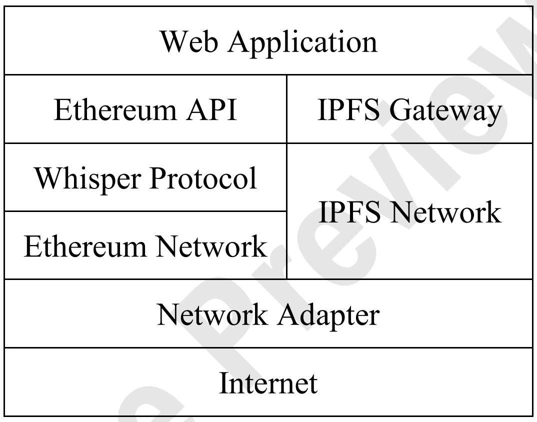


Figure 2: Technology stack of the web application

The system architecture diagram is shown in Figure 3. In this diagram, the web browser would first access the IPFS network and download this web application from the decentralized IPFS network in Process ①. After loading the front-end resources, the JavaScript API would initialize its connection with the Ethereum client in Process ② and communicate with other users through Whisper protocol powered by Ethereum network. This diagram only illustrates the situation where two users are using this web application. If more users are involved, the process should be similar for each additional user. During the initialization of the web application for each user, Process ① and ② would be executed as mentioned above. How users can use this web application to communicate with others in real time through Ethereum network will be introduced in the last section of this chapter.

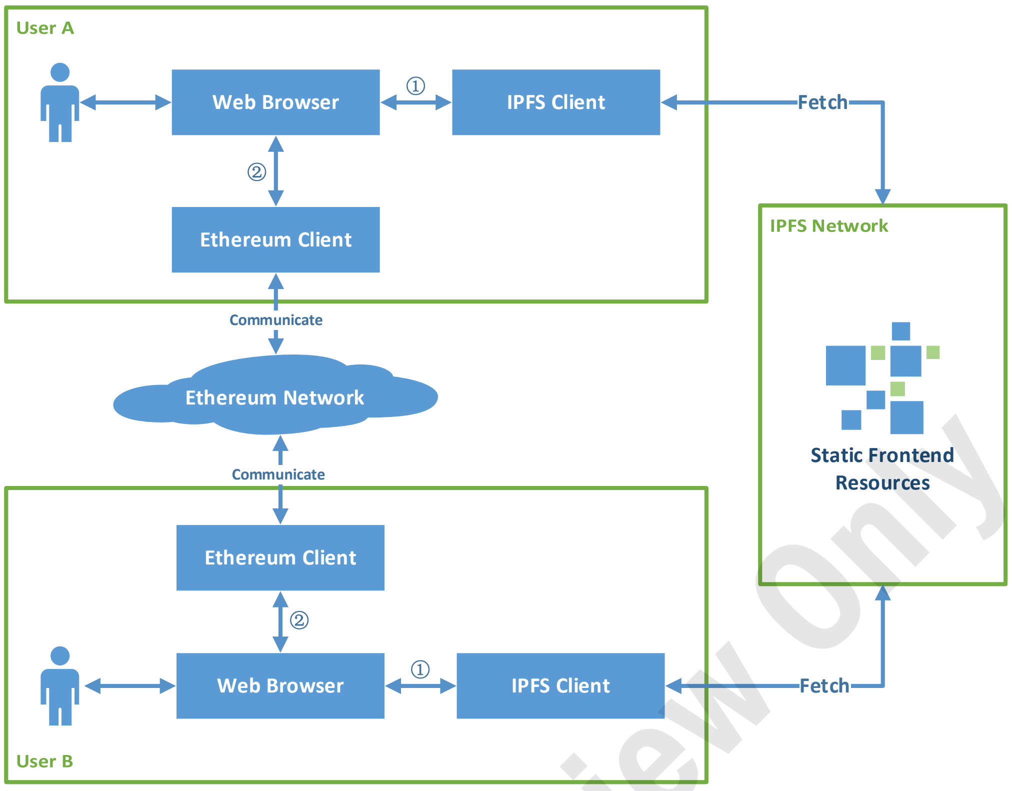


Figure 3: System architecture diagram

- **Web Frontend**

  React and Vue are the two popular modern frontend frameworks for web development. According to the Stack Overflow Developer Survey 2019 [16], React is the most loved web framework. According to *npm* trends [17], React is used by more developers than Vue. Therefore, React is more developer-friendly and has a larger ecosystem.

  Besides, React also provides various state management solutions which could help solve the problem of data persistence after the user leaves the application since there is no central database to save the user's state.

  Therefore, React becomes the final choice for this project.

- **Decentralized backend infrastructure**

  In order to implement this decentralized collaboration platform, the author will choose a blockchain network to provide the service without a centralized server. The comparison of the popular blockchain networks is shown in Table 1.

Blockchain Platform	**Ethereum**	**EOS**	**Bitcoin**

Communication Protocol for Decentralized Applications	Whisper protocol	N/A	N/A
Comprehensiveness of Smart Contract Ecosystem	☆☆☆	☆☆	☆
Smart Contract Scripting Language	Solidity / Vyper [18]	C++	Ivy

Table 1: Comparison of multiple blockchain platform

Communication between multiple nodes is critical to build such a collaboration platform. Whisper protocol [19] is established in Ethereum network to provide a solution for it. Besides, Ethereum's ecosystem of smart contracts is the most comprehensive one, which also ensured the robustness of the fundamental development tools. Therefore, Ethereum will be chosen in this project.

- **Decentralized storage**
  A decentralized storage solution is also required in this project to store the front-end resources and other static resources uploaded by users. Sia and IPFS are the two popular decentralized storage solutions on the market. Sia focuses on the secure third-party decentralized cloud storage, and the service must be paid by Siacoin, another kind of cryptocurrency [20]. On the other hand, IPFS supports the content self-hosting, which is free to use [21]. Besides, IPFS has more online resources than Sia. Therefore, the author uses IPFS as the solution in this project since it is more affordable and easier to learn.

- **Message-based peer-to-peer communication**
  As a decentralized web application, the synchronization of the states among different clients within the same group would be a challenge since there is no central database to store and process the states of different users. To deal with this issue, all the backend communication among different clients in this web application is based on Whisper messages which is broadcasted to every client (including the sender itself) in the same group. The structure of the message objects is carefully designed by the author in order to meet the requirement of this web application and keep simplified at the same time. Four types of messages have been designed, and the general schema of message objects is shown in Figure 4.

messages	
id	*string*
type	*enum*
text	*string*
sender	*string*
date	*Date*
actionType	*enum*
listId	*string*
todoId	*string*

Figure 4: general schema of message objects

The structure of standard message objects contains the following properties:

- *id*

  This property contains a unique random string generated by the library *shortId*. Even though there is no database to store these messages, a unique id is still essential for React to efficiently render the list of chat messages in the chat panel.

- *type*

  Four different types of messages are introduced in this web application:

  1. SEND_MESSAGE

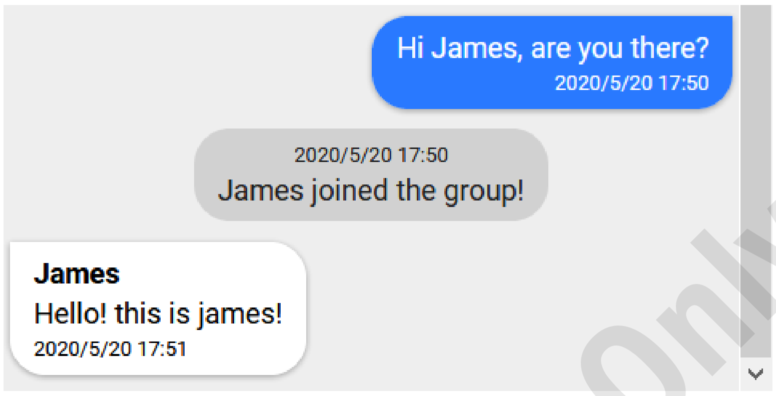
     This type of Whisper messages is to transmit the chat messages sent by the users. The incoming message handler will forward this type of message to the Redux reducer of the state of messages for further processing.

  2. INIT_CHAT

     This type of Whisper messages is sent when this web application is configured and loaded. This type of messages has three functionalities.

     First, the library web3.js only provides a callback function which would be triggered when a message is received from the subscription of Whisper message channel. However, it does not provide a callback function which should be triggered right after the completion of the subscription of the Whisper message channel. Therefore, the *subscriptionID* could not be directly retrieved after the subscription. This kind of messages acts as a workaround for this issue by sending an INIT_CHAT message to the Whisper message channel after the subscription. Since the sender itself can also receive the messages it sent from the channel, the callback function which would be triggered when a message is received is used in this scenario in order to retrieve and save the *subscriptionID* in order to manipulate the subscription to the Whisper message channel.

Second, this message also acts as a notification to other clients in the same group in order to inform them that a new user has joined the group just now. This is an improvement to the end user experience. The users can know who they can talk with by reading this kind of notification in the chat panel as shown in Figure 5.



Hi James, are you there?
2020/5/20 17:50

2020/5/20 17:50
James joined the group!

James
Hello! this is james!
2020/5/20 17:51

Figure 5: notification to other users when joining a group

Third, this type of messages can also act as a request to other clients in the same group in order to retrieve the latest Kanban in this group. When other clients receive this kind of message, they will respond to it with their current Kanban with the message type of INIT_KANBAN.

3. INIT_KANBAN

   A client will send their current Kanban with the message type of INIT_KANBAN when it receives a message with the type of INIT_CHAT. The last modified date of the Kanban is also attached in the sent message.

   When a client receives INIT_KANBAN, it will compare the last modified date of the received Kanban with the local one. If the received Kanban is newer than the local one, then the client will update its Kanban according to the received one. Otherwise, the client will ignore the message.

4. KANBAN_ACTION

   When the user performs an action on the local Kanban, instead of directly alter the local state of Kanban, this application will broadcast this action to every client in the same group. When a client receives the message with the type of KANBAN_ACTION, the incoming message handler will trigger the corresponding action to alter the local state of Kanban. As mentioned before, the sender itself can also receive the message that it sent in the Whisper message channel. So, the user who performed the action can update the state of his local Kanban by broadcasting and receiving the action message, and alter the state of Kanban according to received message.

- *text*

  This property has different usages in different types of message.

  1. SEND_MESSAGE: this property of will contain the text of the chat messages sent by the user to the group.
  2. INIT_CHAT: this property is always null in this type of messages.
  3. INIT_KANBAN: this property will contain the state object of Kanban.
  4. KANBAN_ACTION: this property will contain the updated text which is required in some of the Kanban actions such as ADD_TODO, MODIFY_TODO and MODIFY_LIST_TITLE. In other Kanban actions (like REMOVE_TODO, TOGGLE_TODO) which do not require a text as its parameter, this property will be null.

- *sender*

  This property contains the username which is configured when the user first launches this web application. It is used to identify the sender of the messages, which is critical to distinguish the messages from other users from the messages from the sender itself.

- *date*

  This property contains the JavaScript date object which contains the timestamp when the message is sent. However, in the INIT_KANBAN message, this property contains the last modified date of the sender's Kanban.

In the KANBAN_ACTION messages, some other properties are also included in the message object:

- *actionType*

  This property specifies the particular action the sender performs. These actions include ADD_LIST, REMOVE_LIST, MODIFY_LIST_TITLE, ADD_TODO, TOGGLE_TODO, REMOVE_TODO and MODIFY_TODO. The incoming message handler will trigger the corresponding action to alter the state of the local Kanban based on this property.

- *listId*

  This property specifies the ID of the to-do list in which the action performs.

- *todoId*

  This property specifies the ID of the to-do item in which the item-related action performs (e.g. ADD_TODO, TOGGLE_TODO, REMOVE_TODO and MODIFY_TODO).

Implementation

Since the central server does not exist in this decentralized web application, the traditional client-server paradigm does not apply here, which is the main challenge when implementing this web application in which real-time communication is crucial. This chapter will explain the author's approach to tackle this challenge by utilizing the Whisper protocol.

- **Data structure in application state**

  Since there is no central database to store the data from users, all user states are stored locally and managed by Redux. This web application has the following state objects:

  - *messages*

    This state object stores the list of messages to be rendered in the chat panel. This object consists of a list of msgSegment. Each msgSegment has the following properties:

    1. *id*

       This is the unique random id required for React to render the list of messages. For the type of SEND_MESSAGE, this id is the same of the id of the first element in the property of *messages*. For the type of INIT_CHAT, this id is the same as the id in the received INIT_CHAT message from the Whisper message channel.
    2. *type*

       The possible value of this property is either "SEND_MESSAGE" or "INIT_CHAT". It determines the type of the message that React will render. As shown in Figure 5, the msgSegment with the type of INIT_CHAT will be rendered as a centered join bubble with the text of "… joined the group!". Meanwhile, the msgSegment with the type of SEND_MESSAGE will be rendered as a chat bubble.
    3. *sender*

       This property contains the sender of the messages. The consecutive messages sent by the same sender are grouped as a msgSegment.
    4. *messages/date*

       The key of this property depends on the value of the property of *type*. This property will be *messages* when the type is SEND_MESSAGE. Otherwise, this property will be *date*. The INIT_CHAT messages will not be grouped together like the messages with the type of SEND_MESSAGE.

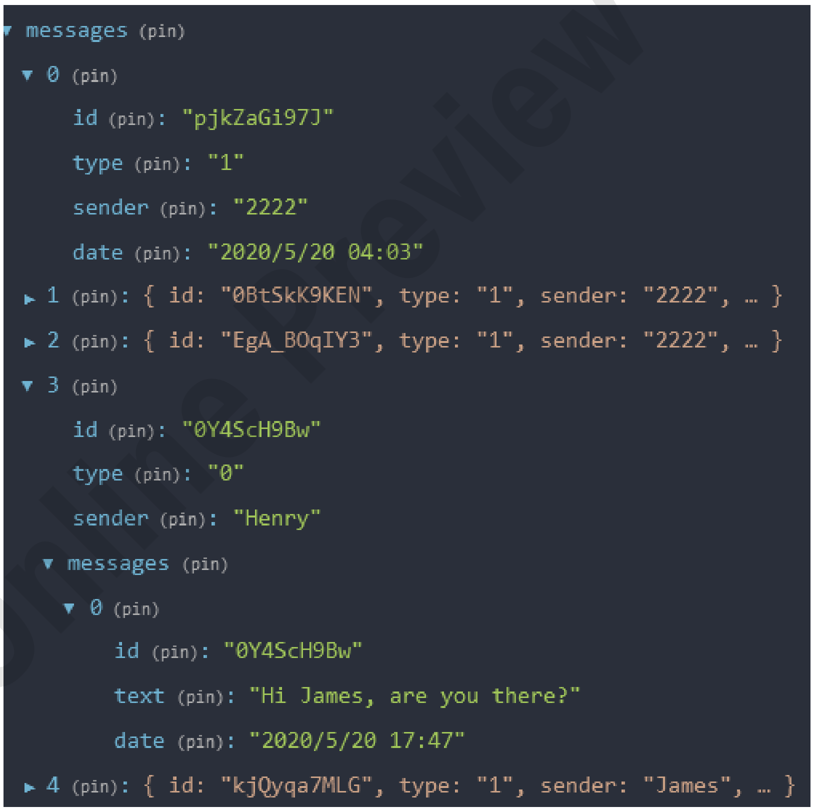
       If the key of this property is *messages*, the value should be a list of *msg* object. Each *msg* has the following properties:

a) *id*: This is the unique random id required for React to render the list of messages, which is retrieved from the id in the received messages with the type of SEND_MESSAGE from the Whisper message channel.

b) *text*: The text that the sender inputted in the message.

c) *date*: The timestamp when the message is sent.

When a new message is received, the reducer will check if the sender of the new message is the same as the sender of the last message in the last msgSegment. If so, the reducer will add the new message to the last msgSegment. Otherwise, the reducer will create a new msgSegment with this message. If the type of the message is INIT_CHAT, the reducer will directly create a new msgSegment instead of merging this message to the last msgSegment. A typical state object of messages is shown in the figure below.



```
▾ messages (pin)
  ▾ 0 (pin)
      id (pin): "pjkZaGi97J"
      type (pin): "1"
      sender (pin): "2222"
      date (pin): "2020/5/20 04:03"
  ▸ 1 (pin): { id: "0BtSkK9KEN", type: "1", sender: "2222", … }
  ▸ 2 (pin): { id: "EgA_BOqIY3", type: "1", sender: "2222", … }
  ▾ 3 (pin)
      id (pin): "0Y4ScH9Bw"
      type (pin): "0"
      sender (pin): "Henry"
    ▾ messages (pin)
      ▾ 0 (pin)
          id (pin): "0Y4ScH9Bw"
          text (pin): "Hi James, are you there?"
          date (pin): "2020/5/20 17:47"
  ▸ 4 (pin): { id: "kjQyqa7MLG", type: "1", sender: "James", … }
```

Figure 6: typical state object of messages

- *kanban*

This state object stores all the data required for React to render a kanban. This object consists of a list of todoList. Each todoList has the following properties:

1. *id*: The unique random string generated by shortId. This would help React to render a number of todoList.

2. *title*: The title of a todoList. The default title is “Untitled List”. Users can modify the title after the creation of a todoList.
3. *items*: This property consists of a list of *todoItem* objects. Each *todoItem* has the following properties:
    a) *id*: The unique random string generated by shortId. This would help React to render a number of todoItem.
    b) *caption*: The text of each todoItem, which could be modified by users.
    c) *isCompleted*: Whether this todoItem is completed. A completed todoItem will be displayed as a finished task on Kanban (Figure 7).

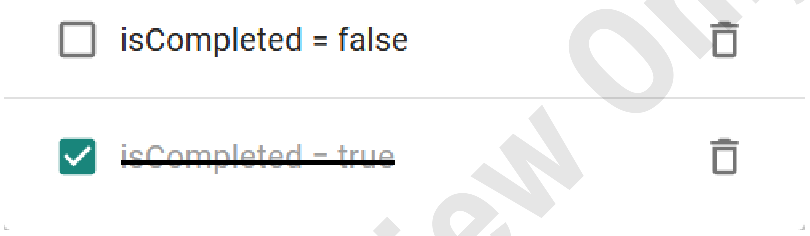
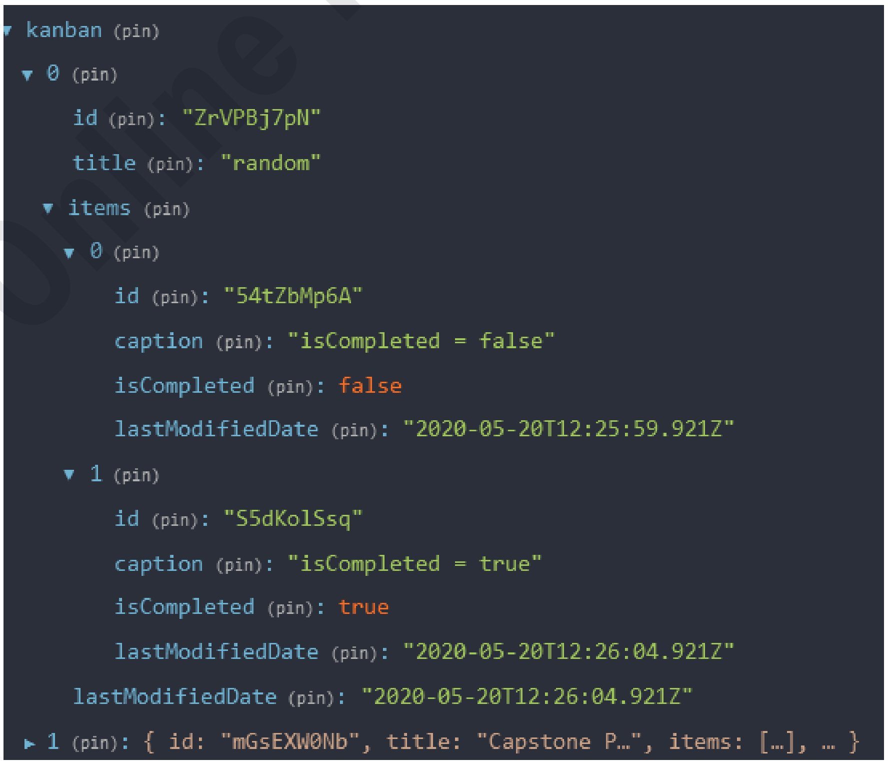


Figure 7: to-do items with different values of isCompleted

    d) *lastModifiedDate*: The last modified date of this todoItem.
4. *lastModifiedDate*: The last modified date of this todoList.

A typical Kanban state object is shown in the figure below.



```
▾ kanban (pin)
 ▼ 0 (pin)
     id (pin): "ZrVPBj7pN"
     title (pin): "random"
   ▼ items (pin)
     ▼ 0 (pin)
         id (pin): "54tZbMp6A"
         caption (pin): "isCompleted = false"
         isCompleted (pin): false
         lastModifiedDate (pin): "2020-05-20T12:25:59.921Z"
     ▼ 1 (pin)
         id (pin): "S5dKolSsq"
         caption (pin): "isCompleted = true"
         isCompleted (pin): true
         lastModifiedDate (pin): "2020-05-20T12:26:04.921Z"
     lastModifiedDate (pin): "2020-05-20T12:26:04.921Z"
 ▸ 1 (pin): { id: "mGsEXW0Nb", title: "Capstone P…", items: […], … }
```

Figure 8: typical state object of Kanban

- *kanbanMeta*

This state object stores the metadata of the local Kanban. This object only contains one property which is *lastModifiedDate*. This application uses this property to update the local Kanban when an INIT_KANBAN message is received. This property will also be updated every time a KANBAN_ACTION is triggered by the incoming message handler.

- *whisper*

  This state object stores the configuration of the whisper protocol in order to communicate with the Ethereum client. This object contains the following properties:

  1. isConfigured

     A Boolean variable which indicates whether the other properties in this state object is configured or not. Default value is false.

  2. username

     This property stores the username of the user. This can be directly specified by users in the settings dialog of this application.

  3. topic

     In the definition of Whisper RPC API 6.0 [22], topics act as a probabilistic message filter to filter out the irrelevant messages on the Ethereum network in order to reduce the workload of the Ethereum client. In this web application, the topic will be automatically assigned with the first ten characters of the symmetric key.

  4. symKeyID

     This property represents the handle to the symmetric key in the memory of the Ethereum client [22]. In the recommended workflow of the Whisper API documentation, the application itself should store the ID of the symmetric key instead of the symmetric key itself.

  5. symPassword

     For end users, this property is called "group ID". The Ethereum client will generate a symmetric key from this password using AES GCM algorithm [22] and store the encrypted key in the memory of the Ethereum client, only returning a random handle to the key. The Ethereum clients with the symmetric key which is generated by the same password can decrypt the messages from each other. Therefore, the concept of symmetric password is simplified to be "group ID". The users with the same symmetric password can communicate with each other.

  6. subscriptionID

     This property records the subscription ID of the Whisper message channel. It will be updated every time this application is reloaded.

The flow chart below shows how this web application initializes its connection with Whisper message channel and how this state object is utilized during this process.

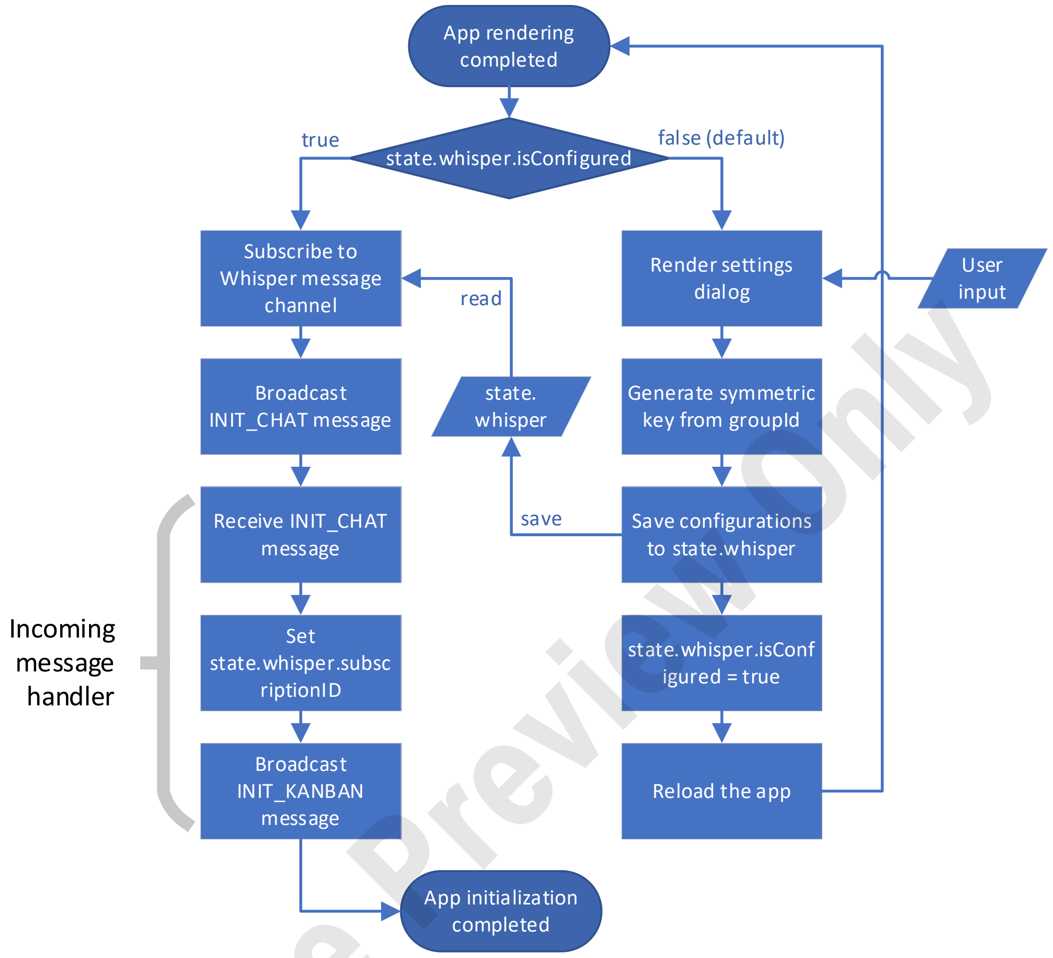

App rendering completed
state.whisper.isConfigured
true
false (default)
Subscribe to Whisper message channel
read
Render settings dialog
User input
Broadcast INIT_CHAT message
state. whisper
Generate symmetric key from groupId
save
Receive INIT_CHAT message
Save configurations to state.whisper
Incoming message handler
Set state.whisper.subscriptionID
state.whisper.isConfigured = true
Broadcast INIT_KANBAN message
Reload the app
App initialization completed

Figure 9: Application initialization flow chart

- *shh*

  This state object stores the web3.shh object, which is the public interface to communicate with the Ethereum client and access the Whisper message channel for the whole application.

- **State persistence and state-driven UI rendering**

  Since there is no central server to store the state of each user, it is crucial to persist the user state locally. Otherwise, the user will loss all his/her data including the to-do lists, chat history and Whisper configuration when he/she leaves this application, which could lead to a terrible user experience. Therefore, the rendering of the components on the user interface is solely based on the user states. The user interface is designed in a way that could be fully restored by a given user state. The state persistence and state-driven rendering will be introduced in this section.

  1. State persistence

This application originally used the built-in state management solution provided by React to deal with the application state. However, this solution has several drawbacks. First, the states are declared and used within the corresponding React components. Therefore, the declarations of the states are usually scattered in multiple components. If a component would like to access the state which is not declared in itself or its parent component, the programmer should lift the state up and pass the state down to its child using *prop*. It would be very complicated and time-consuming to re-construct the code in this way. Second, since there is no central store to keep every state in the web application, it is particularly complicated to manage the persistence of the states in different React components separately.

Due to these drawbacks, I did some research on the internet and found another state management solution called Redux. Redux effectively solves the problems mentioned above by providing a central state store and its container-action-reducer architecture. Besides, there is also a third-party library called *redux-persist* which is able to persist the central state store of Redux in the localStorage of the web browser. This library can save the application state before the user leaves the application, and load the state from localStorage when the user loads this application. In this way, the application state will persist in the localStorage of the user's web browser, and the users do not need to worry about losing their data when they want to use the application in the future.

2. State-driven UI rendering

   Since the problem of state persistence is solved, how to make sure that the application UI will keep the same after reloading? The persisted states managed by Redux could be connected with React by mapping the Redux state objects to the properties of the corresponding React components thanks to the *react-redux* library. The typical virtual DOM tree of a configured web application is illustrated in Figure 10. Blue rectangles represent a virtual DOM element, and the green rectangles above a virtual DOM element represents the Redux state objects which are connected with this element. As you can see in this figure, the rendering of the *Kanban* DOM element is driven by the Redux state of *kanban*, and the rendering of the *ChatPanel* DOM element is driven by the Redux state of *messages*. The actual UI in this situation is illustrated in Figure 11. The state objects of *whisper* and *shh* are essential for every React component which needs to send or receive messages to/from the Whisper message channel, so they are connected to many DOM elements.

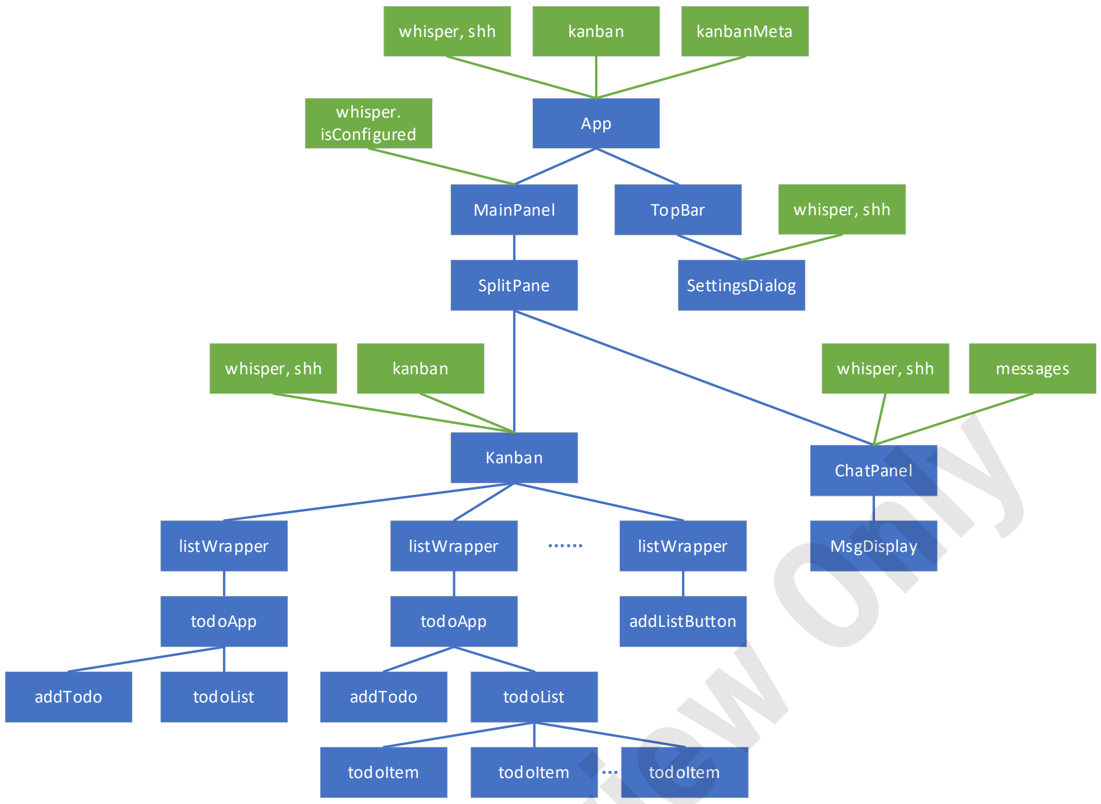
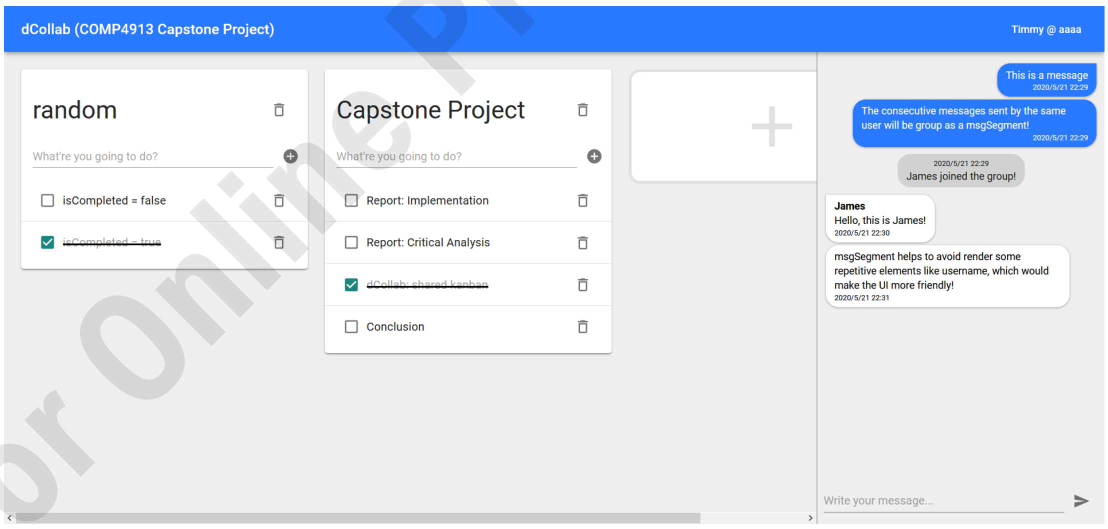


Figure 10: typical virtual DOM tree when whisper.isConfigured == true



random

What're you going to do?

- [ ] isCompleted = false
- [x] ~~isCompleted = true~~

Capstone Project

What're you going to do?

- [ ] Report: Implementation
- [ ] Report: Critical Analysis
- [x] ~~dCollab: shared kanban~~
- [ ] Conclusion

This is a message
2020/5/21 22:29

The consecutive messages sent by the same user will be group as a msgSegment!
2020/5/21 22:29

2020/5/21 22:29
James joined the group!

James
Hello, this is James!
2020/5/21 22:30

msgSegment helps to avoid render some repetitive elements like username, which would make the UI more friendly!
2020/5/21 22:31

Write your message...

Figure 11: an example of UI when whisper.isConfigured == true

The virtual DOM tree shown in Figure 10 illustrates the situation only when the value of *whisper.isConfigured* is true. Otherwise, the visibility of the *SettingsDialog* will become true and the *MainPanel* will render nothing. The virtual DOM tree in this scenario is shown in Figure 12, and the actual UI in this situation is illustrated in Figure 13.

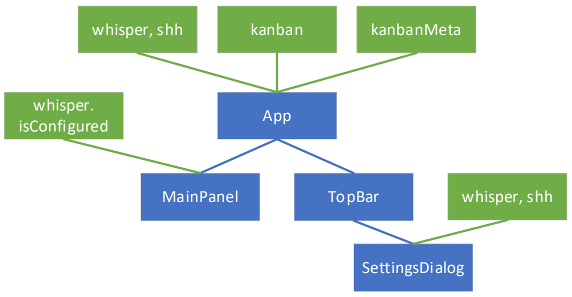
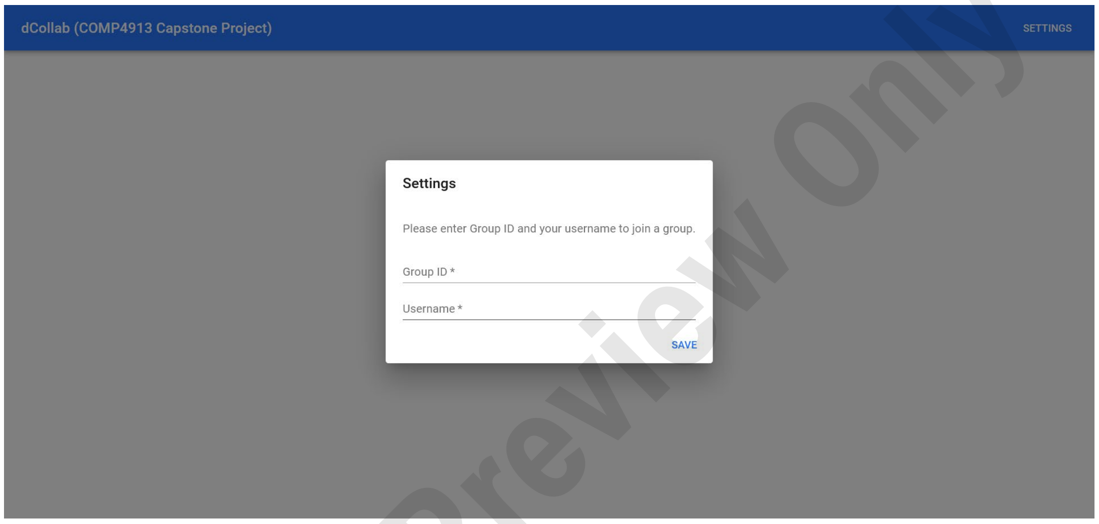


Figure 12: typical virtual DOM tree when whisper.isConfigured == false



Please enter Group ID and your username to join a group.

Figure 13: an example of UI when whisper.isConfigured == false

- **Message handling**

  The message-based peer-to-peer communication has been introduced in the previous chapter. This section will introduce the implementation of Whisper message handlers in this web application.

  1. One incoming message handler

     As illustrated in Figure 9, the application will subscribe to the Whisper message channel when *whisper.isConfigured* equals to true during the initialization. According to the Whisper API [22], a callback function which will be triggered when a message is received is required during the subscription. Therefore, there is only one incoming message handler in the *App* component of this application which acts as a callback function for the Whisper subscription. After receiving a new message, the incoming message handler will first check the *type* of the message. There are four possible types in a message which was introduced in the previous chapter. If the *type* is KANBAN_ACTION, the handler will further check the *actionType* in the message.

After that, the handler will trigger the corresponding state actions to alter the user states and re-render the user interface.

2. Multiple outgoing message handlers

   There are three outgoing message handlers in this application. In the *App* component where the initialization process takes place, there is a handler which is responsible for sending the messages with the types of INIT_CHAT and INIT_KANBAN. In the *Kanban* component, there is also an outgoing message handler which is designed to send KANBAN_ACTION messages. The last handler lies in the *ChatPanel* component, which is dedicated to send messages with the type of SEND_MESSAGE.

Evaluation

The transmission performance of this decentralized web application will be evaluated in this chapter by experiments.

- **Experiment: Relationship between the transmission delay and the message length**
  1. Objective: This experiment aims to discover how the transmission delay of a Whisper message is affected by the length of it.
  2. Environment:
     a) Network: A virtual machine (Ubuntu Server 18.04 running on Hyper-V) and its host computer (Windows 10 Pro 20H1) is connected in the same subnet through the default virtual switch of Hyper-V. Two Ethereum clients (*geth*) are running on the VM and its host respectively. These two Ethereum clients are connected to the same private Ethereum chain and are mutually recognized as an Ethereum peer.
     b) Application: The web application which runs on the two nodes are slightly modified in order to perform the experiment automatically. The production version of the application is used in order to simulate the performance in reality. Both two nodes are using Firefox Browser v76.0.1 to run the web application.
  3. Methodology:
     When the experiment starts, the web application will send a message of a certain number of "0" with the type of SEND_MESSAGE every second. The length of the message begins with 8, and ends with 512. For every specific length of message, the application will repetitively send for 8 times. After that, the length will increase by 8, and the process continues. Therefore, the lengths of the messages sent in this experiment in total are (8, 8, 8, 8, 8, 8, 8, 8, 16, 16, 16, 16, 16, 16, 16, 16, 24, 24, ……, 504, 504, 512, 512, 512, 512, 512, 512, 512, 512). There are 512 messages to sent in total.
     Transmission delays are measured in milliseconds, and it is impractical to synchronize the system clocks of the two nodes at the level of milliseconds. Therefore, the transmission delays will be measured on the node which sends the testing messages. Since an Ethereum client can receive the messages it sent, the messages received on the sender's side are also transmitted through the Ethereum network. Therefore, this approach would not affect the accuracy of the results.
  4. Result and Findings:

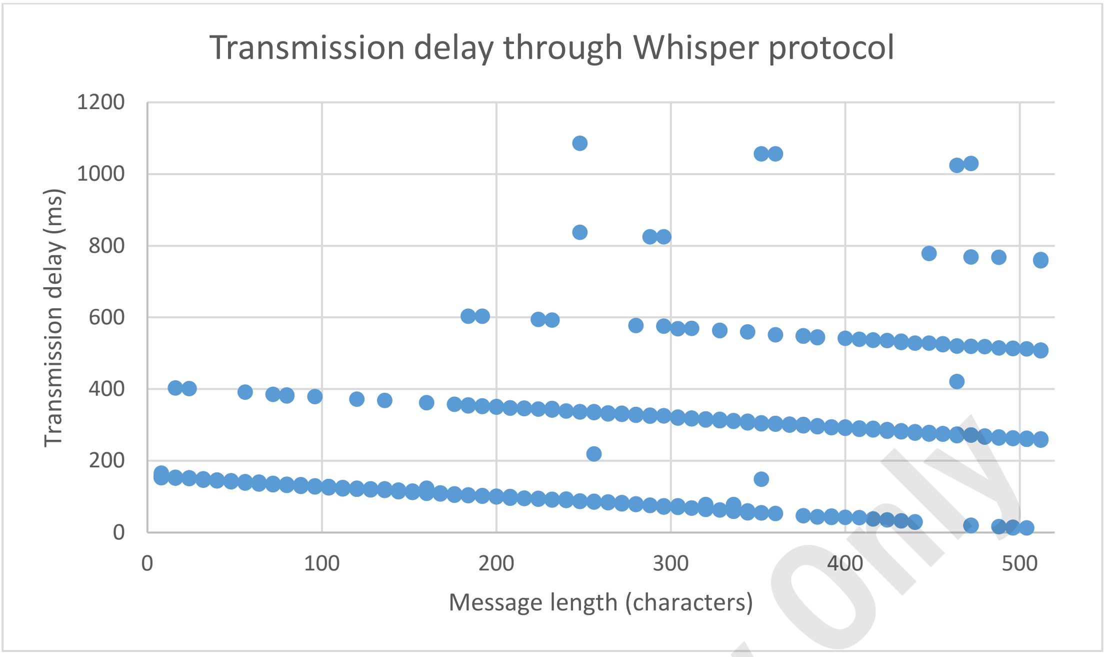


Chart 1: Transmission delay through Whisper protocol

The experiment result is shown in a scatter chart in Chart 1. For the complete set of raw data, please refer to Appendix 1. The author has several findings by observing this scatter chart:

a) As the message length increases, the maximum possible transmission delay is increasing, while the minimum possible transmission delay is decreasing.

b) Although the deviation of transmission delay increases as the message length increases, the possible values of the transmission delay do not distribute in a continuous fashion. Instead, they tend to distribute discretely with fixed gaps which is a difference of around 250 milliseconds. Table 2 shows a slice of the raw data shown in the scatter chart, which illustrates the discrete distribution of the transmission delay.

msg length	delay (ms)
296	71
296	72
296	72
296	74
296	325
296	325
296	575
296	825

Table 2: Excerpt of raw experiment data

c) Among the 512 tests, only 5 tests exceed the transmission delay of 1000ms. Therefore, the frequency of a message transmission which takes less than one second is 99% in this experiment when the length of message is no more than 512.

Chart 2 illustrates the trend of average transmission delay as the length of message increases. According to the linear regression, it is obvious that the average transmission delay is generally increasing as the length of message increases. However, the average transmission delay becomes more unpredictable as the length of message continues to grow.

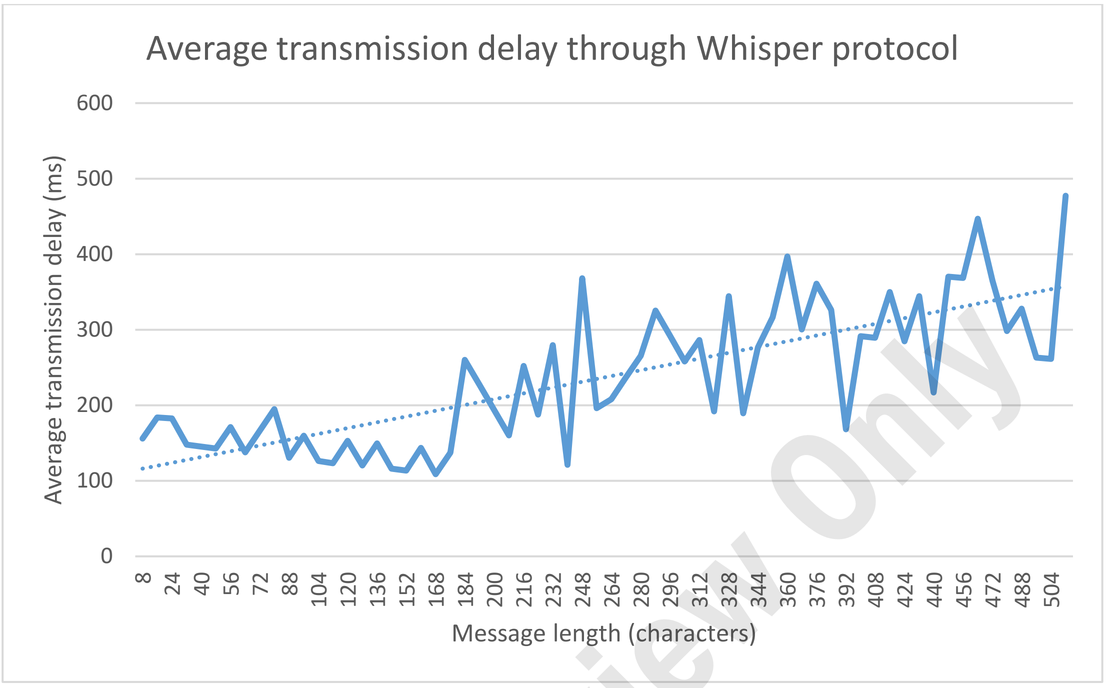


Chart 2: Average transmission delay through Whisper protocol

5. Conclusion:

The average transmission delay will increase in a linear fashion as the length of message increases, but the transmission delay will become more unpredictable if the length of message continues to grow.

Conclusion

This report introduced a decentralized web application which fully utilized the Whisper protocol of Ethereum to achieve real-time communications among users in a decentralized fashion. This application aims to help the university students to collaborate on their coursework by providing two main features: instant messaging and shared Kanban. Users can form a group and chat with each other in this application. Besides, the users can also use the shared Kanban to share their progress on the coursework with other users in the same group. The major challenge when implementing this web application is that the traditional client-server paradigm does not apply in this decentralized web application. To tackle this challenge, the author designed multiple approaches to make this application feel like a traditional client-server web application for end users so that they will still feel familiar with this application. These approaches include message-based peer-to-peer communication and state-driven UI rendering. The performance of this application was evaluated as well by conducting the experiment to discover the relationship between the length of messages and the transmission delay. The experiment result showed that the average transmission delay will increase in a linear fashion as the length of message increases.

Nowadays, many decentralized applications based on blockchain networks have been developed by numerous researchers and engineers. However, few decentralized applications could be found in the field of real-time communication. The author hopes this application could be a reference for other researchers and developers who are seeking for a working example of a decentralized application with the feature of real-time communication.

Acknowledgement

The author would like to thank the supervisor of this capstone project Dr Henry Chan. Dr Chan has helped the author to determine the main direction of this project during the early stage. He has given numerous precious advice and suggestions to improve the web application and this report. This project cannot be finished without the help of Dr Chan.

The author would also like to express the appreciation to the Whisper JavaScript example given in the official documentation of the Whisper API [23]. This example provides a minimum viable product for developers to build a complex decentralized web application based on Whisper protocol.

The following open-sourced JavaScript libraries are used to build this web application. Without the efforts of the open source contributors of these projects, this web application is not possible to be built in one academic year. Some libraries which have been mentioned above will not be introduced here.

- **web3:** https://github.com/ethereum/web3.js
  This library provides an official JavaScript API to access the Ethereum client. This web application is impossible to connect to Ethereum network without this library.
- **react:** https://github.com/facebook/react
- **redux:** https://github.com/reduxjs/redux
- **react-redux:** https://github.com/reduxjs/react-redux
- **redux-persist:** https://github.com/rt2zz/redux-persist
- **flatted:** https://github.com/WebReflection/flatted
  This library solved the issue where the Redux state objects failed to persist in localStorage due to some circular objects inside the states.
- **material-ui:** https://github.com/mui-org/material-ui
  This library provides beautiful and user-friendly UI components to use in React.
- **shortid:** https://github.com/dylang/shortid
- **formik:** https://github.com/jaredpalmer/formik
  This library provides convenient built-in handlers to deal with the states and events in the forms in this web application.
- **react-split-pane:** https://github.com/tomkp/react-split-pane
  This library provides an encapsulated UI component to build a split pane in this application.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system." [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, 25-30 June 2017 2017, pp. 557-564, doi: 10.1109/BigDataCongress.2017.85.

[3] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets." [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper,* vol. 151, no. 2014, pp. 1-32, 2014.

[5] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F. Wang, "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems,* vol. 49, no. 11, pp. 2266-2277, 2019, doi: 10.1109/TSMC.2019.2895123.

[6] J. Ray, M. Wilson, and G. Wood, "Whisper PoC 2 Protocol Spec," 22 Aug 2018. [Online]. Available: https://github.com/ethereum/wiki/wiki/Whisper-PoC-2-Protocol-Spec

[7] M. S. A. El-Seoud and I. A. Taj-Eddin, "Developing an android mobile bluetooth chat messenger as an interactive and collaborative learning aid," in *International Conference on Interactive Collaborative Learning*, 2016: Springer, pp. 3-15.

[8] A. Deb and S. Sinha, "Bluetooth Messenger: an Android Messenger app based on Bluetooth Connectivity," *IOSR Journal of Computer Engineering (IOSR-JCE),* vol. 16, no. 3, pp. 61-66, 2014.

[9] M. A. Gogy and J. Thomas, "Offline Messenger - For Devices Out of Direct Bluetooth Range," in *2017 International Conference on Recent Advances in Electronics and Communication Technology (ICRAECT)*, 16-17 March 2017 2017, pp. 109-113, doi: 10.1109/ICRAECT.2017.21.

[10] J. Partala, "Provably Secure Covert Communication on Blockchain," *Cryptography,* vol. 2, no. 3, p. 18, 2018.

[11] T. Gu, "OnionCoin: peer-to-peer anonymous messaging with incentive system," University of British Columbia, 2018.

[12] M. Abdulaziz, D. Çulha, and A. Yazici, "A Decentralized Application for Secure Messaging in a Trustless Environment," in *2018 International Congress on Big Data, Deep Learning and Fighting Cyber Terrorism (IBIGDELFT)*, 3-4 Dec. 2018 2018, pp. 1-5, doi: 10.1109/IBIGDELFT.2018.8625362.

[13] B. Lee, M. Y. Lee, H. S. Ko, S. H. Myung, M. O. Kim, and J. H. Lee, "A Secure Mobile Messenger Based on Ethereum Whisper " *The Journal of Korean Institute of Communications and Information Sciences,* vol. 42, no. 7, pp. 1477-1484, 2017, doi: 10.7840/kics.2017.42.7.1477.

[14] P. Mukherjee, C. Leng, and A. Schürr, "Piki - A Peer-to-Peer based Wiki Engine," in *2008 Eighth International Conference on Peer-to-Peer Computing*, 8-11 Sept. 2008 2008, pp. 185-186, doi: 10.1109/P2P.2008.33.

[15] A. Davoust, A. Craig, B. Esfandiari, and V. Kazmierski, "Decentralized collaboration with a peer-to-peer wiki," in *2012 International Conference on Collaboration Technologies and Systems (CTS)*, 21-25 May 2012 2012, pp. 286-293, doi: 10.1109/CTS.2012.6261064.

[16] "Most Loved, Dreaded, and Wanted Web Frameworks, Stack Overflow Developer Survey 2019." Stack Overflow. https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted (accessed 17-Oct, 2019).

[17] "react vs vue." npm trends. https://www.npmtrends.com/react-vs-vue (accessed 2019, 18-Oct).

[18] Ethereum. "Developer Resources." Ethereum. https://www.ethereum.org/developers/#smart-contract-languages (accessed 18-Oct, 2019).

[19] J. Ray. "Whisper." GitHub. https://github.com/ethereum/wiki/wiki/Whisper (accessed 18-Oct, 2019).

[20] Sia. "Sia's Technology." Sia. https://sia.tech/technology (accessed 18-Oct-2019, 2019).

[21] IPFS. "IPFS is the Distributed Web." IPFS. https://ipfs.io/ (accessed 18-Oct-2019, 2019).

[22] The go-ethereum Authors, "Whisper RPC API 6.0," 5 Nov 2019. [Online]. Available: https://geth.ethereum.org/docs/whisper/whisper-v6-rpc-api

[23] The go-ethereum Authors, "Whisper JavaScript example," 5 Nov 2019. [Online]. Available: https://geth.ethereum.org/docs/whisper/whisper-js-example