Final Project of Distributed Systems WS22/23

1 Introduction

The objective of the project is to design and distributed chatting system. The system allows fundamental functions of private chatting. All servers and clients will passively or actively communicate with each other including Dynamic discovery, certain fault detection and solution, voting mechanism and causal multicasting functions.

2 Project Requirements analysis

The project achieves message-sending and receiving functions. The following functions, therefore, are required for the project:

2.1 Client perspective

Account Registration
Each client can create an account before being accessible to the messaging service, which consists of the following processes:

Register Process:
User -> (Request) -> Leader (elected from Leader election; if the Leader is dead, it will request the next server according to the election result until success.)

Login Process:
Step 1: User -> (ID : Login message) -> Leader
Step 2: Leader -> (Sub server IP) -> User
Step 3: Leader -> (User ID and User IP) -> Serve and update the group view
Step 4: If there are messages in the user's mailbox, the messages will be transmitted; the sent message is hung at the "sub server" side when there are still unsent messages in the FIFO queue in order to guarantee ordering.

User Online Process:
User -> (Get List message) -> Sub server
Sub server -> (Current Online Users message) -> User

User Offline Process:
User -> (logoff or crashes) -> Sub server
The sub-server tells other servers in the group, then the leader updates the group view. Other online Users get the offline notification.

Private Messaging:
Each client can message any online client and leave messages to a custom user, which consists of the following steps:
Step 1: User -> (target User ID) -> Sub Server
Step 2: Sub Server searches the responsible Sub server for the target ID, then forwards the message to the responsible Sub server.
If it is itself, then do step 3.
Step 3: The user reads these messages after it is online.

2.2 Server perspective

Account message storage: Each server stores the ID, address, and messages of each offline client. The client can directly chat with another online client by sending his/her unique ID to the server.
Leading server: The leader detects the state of each server to detect offline servers by Server online process and redistributes the clients connected to the offline servers to other live servers then updates the group view.

Server Liveness Process
Leader -> (Heartbeat message) -> Sub server
Sub server -> (Heartbeat message) -> Leader
Normal server (Sub server): Receive, transmit and forward messages; Detect the liveness between itself and leader. If the connection fails, the leader updates the new group view. If the leader is dead, take a new leader from the election result.
Election mechanism: Elect and vote for a leader by the LaLann-Chang-Roberts algorithm if the current leader fails.
Fault Tolerance during leader election: Restart the election algorithm if the leader node fails. In some cases, the heartbeat liveness detection from a node to the leader fails due to connection or network error and falsely triggers the node to another leader election. In that case, the node must confirm with each other nodes if they enter a new election.

2.3 Message perspectives

Message type: The system massaging application support text form and can be attached with some other information like vector clock, server address and so on.
Online messaging: the client can instantly receive the message if the client is online.
Offline messaging: The message will be stored temporarily in the FIFO order by the server group and will be transmitted once the client is online again.
Time stamps in transmission: The message includes a vector clock to clarify the chronological order.

3 Architecture design

3.1 Big picture

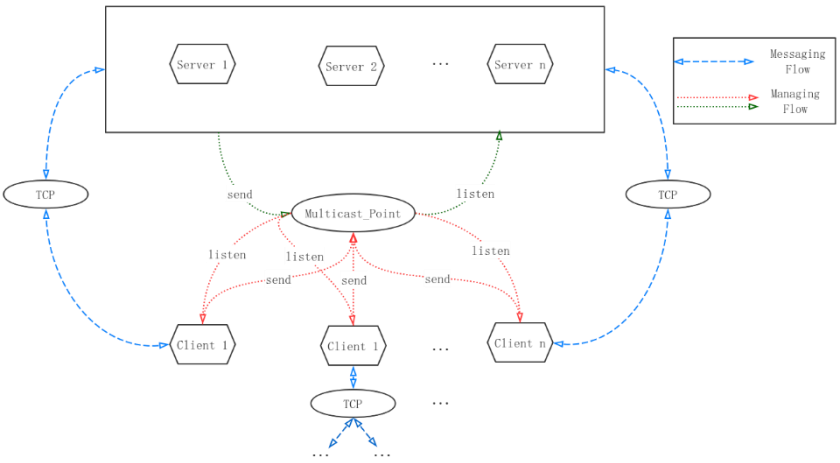

Fig. 1. General topology of the multi-server & multi-client network

The whole structure of the project is illustrated in the above figure. Information flows are divided into two general classes – Messaging flow and Managing flow. Messaging flow transfers client-to-client messages, i.e., user layer messages that any client wants to deliver to each other like chatting content. Managing flow – illustrated by red and green lines - consists of control messages of the network. The Multicast Point connects servers and clients, who send and listen to the instructions relative to them.

3.2 Client structure

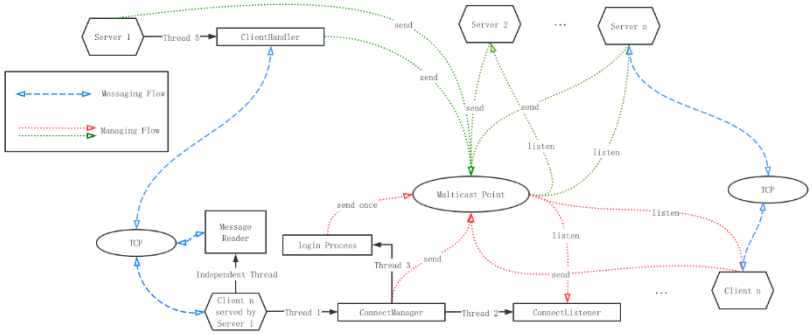

Fig. 2. Client Structure Showing Messaging flow

The single Client model contains 2 parts: A Client (Client) and a Connection manager(clientConnectManager).
Dynamic discovery: When a client is online a timer task (inside the clientconnectmanager) will start multicast until the client is discovered by a server, and the client will be informed which server it should communicate with. In this part, we apply UDP.
Fault tolerance: When a client is offline, its responsible server will discover this error by raising a connection error. The server will shut down the socket and close it. When a server is offline, its responsible client will also respond in the same way with a similar signal. In this part, we apply TCP.
Chatting: We transfer the message by TCP. Each Client has a responsible Server. A TCP is established between each client and its corresponding server. If two clients have different servers, their message will be multicast (UDP) within all servers, and the responsible server will pick up their request. This part will be illustrated in detail in 3.3.
The following section will introduce more details on each model.

Client
The client is responsible for interpreting the input of a user and transferring the message to the corresponding server. At the very beginning, the “name” of the client will be required, to generate. Then the name of the client will be served as a unique ID to identify the receiver of a message.
There are mainly two threads inside the client. The first one is the listener, which is made up of a TCP listener and a message handler; the second one is the reader, which is made up of a TCP sender and a user interface.

Users have 3 options during chatting. The first one is changing the chatting object, the second one is sending a message to the chatting object we have selected before, and the third one is exiting the current section (logging out).

Client connection manager

The client connection manager manages the connection between the Client and the server. The interface, with which the client connection manager could influence its corresponding client, is the socket of the client. In our code, by calling “setServer()”, we could choose the change the responsible server of the client.

In the beginning, this module will multicast in a certain interval(in our case this time interval is 3 seconds). Once this Signal is captured by the server, Server will establish a TCP connection with the Client.

Client-relevant Message

Our standard message contains 5 elements: source ID, target ID, message type, UUID(unique ID of a message, which is made up of the source ID and the current timestamp) and the content of the message. During the encoding(packing) process, all elements will be encoded into string type and then transfer into binary form. During the decoding(unpacking) process, the message will first be transferred into string type, and then extract the five attributes of the message.

There are four types of client-relevant messages.

1 online:

2 offline:

3 msg:

4 getlist: this message contains the current list of other online customers.

3.3 Server structure

The structure of a server is as follows:

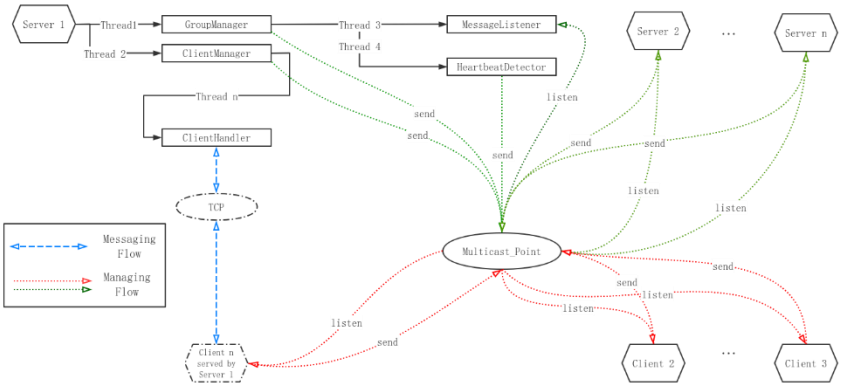

Fig. 3. Server Structure with thread creation, grouping and communication process

Each server has multiple threads, in which the thread ClientHandler is specifically used for messaging with a client, while the others are used for managing. Thread ClientHandler is invoked by an initial thread ClientManager. ClientHandler delivers messages directly to the client served by the server including message type, target id, content and uuid. ClientManager is responsible for managing the connection and disconnection requests between clients and this server, and at the same time constantly detects whether there are client requests for information. The ClientManager is also responsible for storing messages in a queue from the clients currently connected to it
Server

Group manager
This model consists of a message listener, a replication manager, a heartbeat detector and an election module.
The message listener will start first and set up a multicast socket. During the threading, this thread will constantly receive multicast messages. The listener will categorize messages into 3 classes and put them into the corresponding queue, based on the types of messages. They are the Heartbeat queue and membership queue (responsible for the group views of servers) and replication queue. Other models could have accessed those queues by calling the functions of the message listener.
After the listener got started, the heartbeat detector and election model will start running at the same time in parallel.
The heartbeat detector will multicast a heartbeat at a certain frequency. if a participant is not heard after a certain time (In our case, we choose 5 seconds), this participant will add to a list. A message of no connection will be sent by the message listener, so as to inform other servers about this disconnection. At the end of a heartbeat interval, all those failed participants will be removed.
Replication is designed for an online client. The message, which could not be sent properly, will be replicated to all servers, so as to retransfer this message when this

client is online again. This process will be evoked when the message fails to transmit a message to the client, which is an IO exception in reality.
The election process is evoked when the server could not receive a message from the leader. When a server is online, it will request the current group view. Based on this group view, the server will acquire a sequence, which would serve as a ring during the election. There are 2 election-relevant states for each server, one is inElection, which tells whether this server is still in the election process; the other one is isLeader, which is a mark of whether this server is the leader of the current group view.
Client manager. This module will check whether there is a new incoming client, if there is, this module will then generate a new threading to provide service for the client. This new threading is called “client handler”.
Basically, it will handle 4 types of messages.
Login message: after receiving the message, it will multicast the login information and start sending those messages, which have been captured by the server while this client is offline (replication). Then bind this handler with the ID of this client. Lastly, it will also multicast the online information to other clients in this network. This client will also be informed about another online client in the network.
Logout message: this socket will then shut down and this message will also be broadcasted.
Normal message: If the target of the message is being taken charge of by this server, this message will be transferred to the receiver directly via TCP. Otherwise, this message needs to be multicast to other servers and needed to be forwarded there.
Getlist message: After receiving this message, it will transmit a list which tells all the online clients.

3.4 Messaging structure

The structure of client messages is illustrated in the following figure.

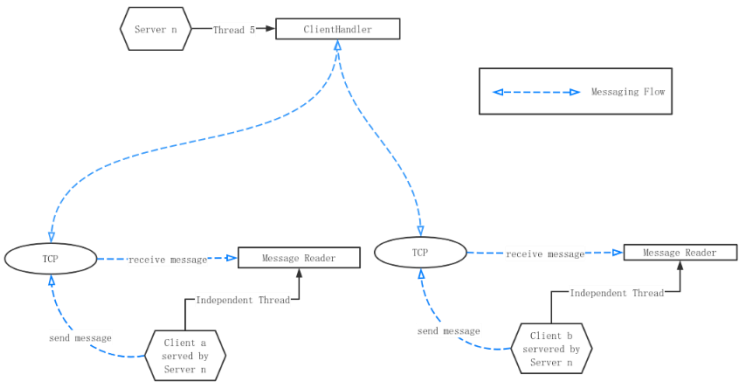

Fig. 4. Message flow when clients are on the same server

When two clients are connected through the same server, the messaging structure for both clients is relatively simple. When the process ClientHandler receives a transmission message from a client, the process will look for another client that corresponds to the id in the message, and then simply send the message over TCP. Since the client needs to display the read information in real-time, we include the reading of the information in a separate client thread to ensure that the client can display the received message in time.

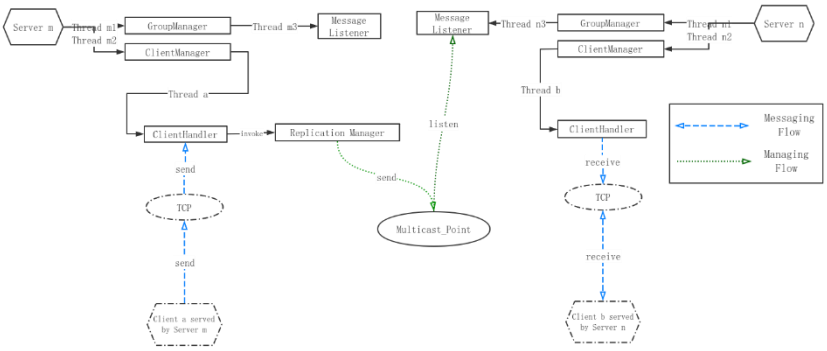

Fig. 5. Message flow when clients are in different servers

When two clients are connected to different servers, the messaging structure of the two clients is relatively complex. When the process ClientHandler receives a transmission message from a client, it looks for another client corresponding to the id in the message, and if it does not find a client with the corresponding id, it wakes up the Replication Manager, which sends this message to the Multicast Point, and the Message Listener process of other nodes periodically reads the message from Multicast Point and compares the consistency of the id in the message with the id of the client under this node when it recognizes the request to send the message from this node. If a matching id is found, the transmission message is sent to that client via TCP

4 Implementation

4.1 Messaging

The message flow test result is shown in the following graph.

Messaging in the same server

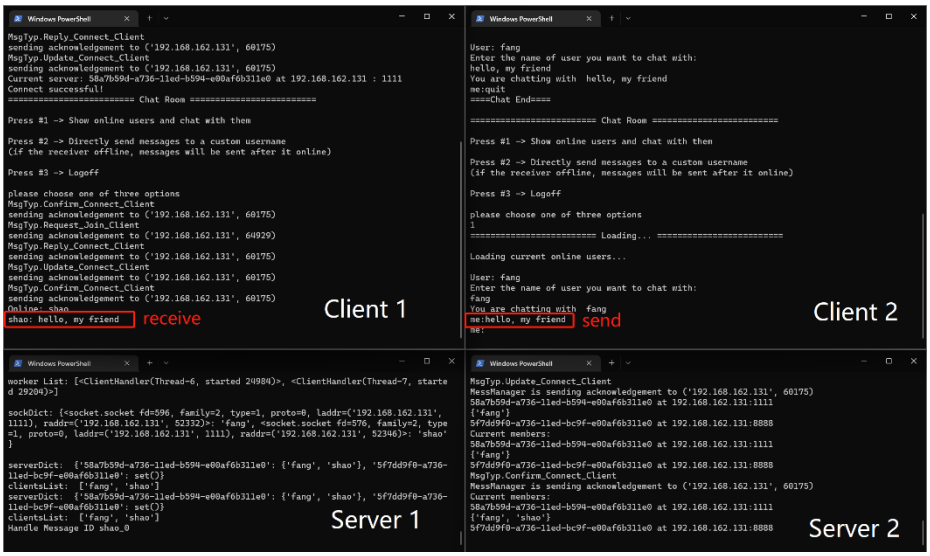

Fig. 6. Message sending in the same server test result

Test procedure:
Open two servers on different devices
Open two clients on the same device, select two clients connected in the same server
Set username for further connection use
Select the client that a client wants to chat with, type the username of the client
Tap the message and send the message to the respective client
See if the respective client receives the message and prints it out in the terminal

Messaging between different servers

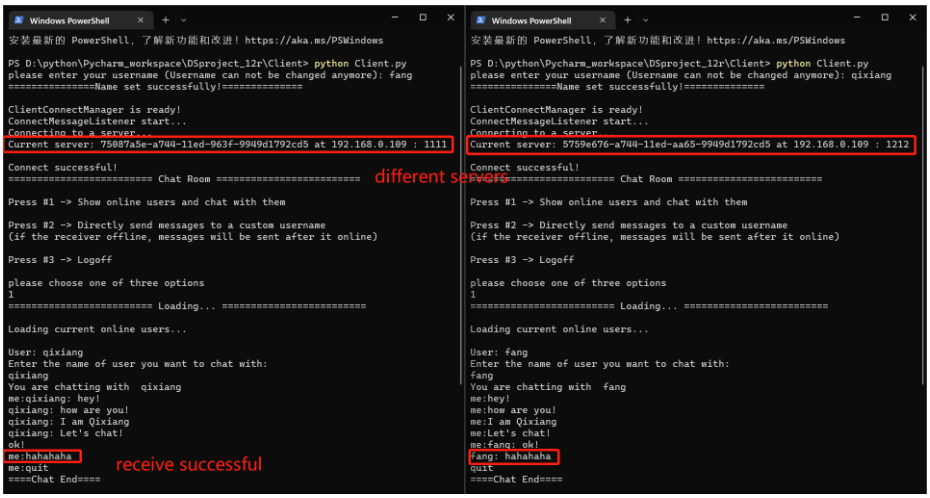

Fig. 7. Message sending to different server test result

Test procedure:
Open two servers on different devices
Open two clients on the same device, select two clients connected to different servers
Set username for further connection use
Select the client that a client wants to chat with, type the username of the client
Tap the message and send the message to the respective client
See if the respective client receives the message and prints it out in the terminal

4.2 Dynamic discovery

The core idea of dynamic discovery is that any node or client in a distributed system can join or leave the network at any time, while the system must update its current membership information - group view - and synchronize the information. The project's Dynamic discovery mechanism is implemented in the GroupManager, which makes decisions based on different message tags (join, leave, no connection, heartbeat detection)

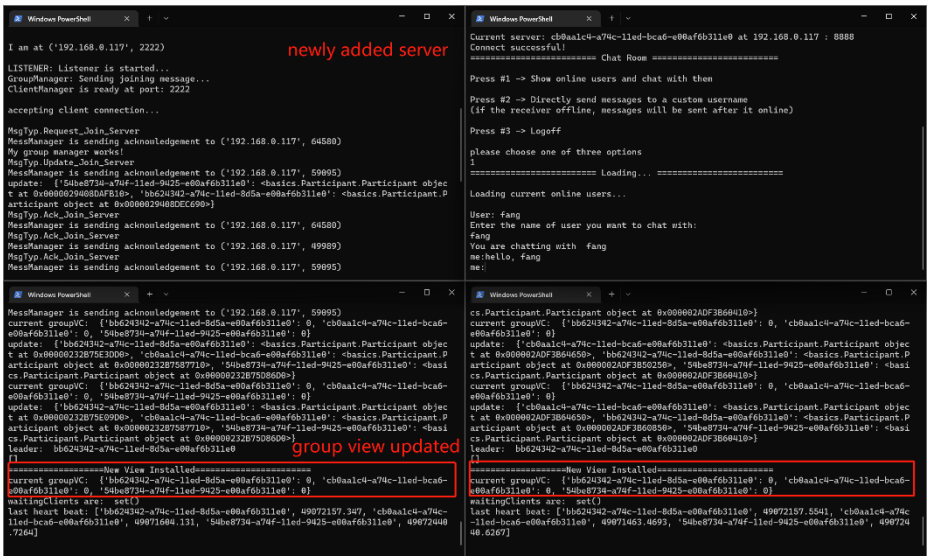

Fig. 8. Dynamic discovery test result

Test procedure:
Server-side: when there is a new server joining the group, this server could be successfully discovered by the other Servers. Here we set up 2 servers (A, B) first. Then we set up a new one (C). Then we will find A, and B can discover C, and C is also able to discover A and B.
Client-side: when a new client is joining the group, this client and server can discover each other.

Apply to join

The server that applies to join sends an application message. The leader node reads the application message in real-time and sends its server address, notifies the server to enter the group, records the server address, waits to receive the ack message from the server, i.e. the new server is ready to enter the group, and then updates the Group view.
For test results see Fig. 7

No connection detected

When one of the servers drops the connection and fails, it interrupts sending heartbeat messages. If the other server does not receive the heartbeat message, it triggers the "no connection" mechanism. In the no-connection mechanism, the leader node takes the same action in the request exit, removes the server's content from the Group view, updates the Group view, and then assigns the clients connected to that server to other servers. If the leader node loses its connection, other common nodes, after judging the lost node as the leader, then trigger the election mechanism to elect a new node as the leader node to take over the work of the lost node.

The main difference between applying to leave and no connection is, that in the application to leave, the node leaves initiatively from the group, which is not a failure to the system, while the node leaves the group passively due to failure in no connection mechanism.
Test result sees 4.3 Fault tolerance heartbeat detection.

Heartbeat Detection
See 4.3 Fault tolerance

4.3 Fault tolerance

Heartbeat detection

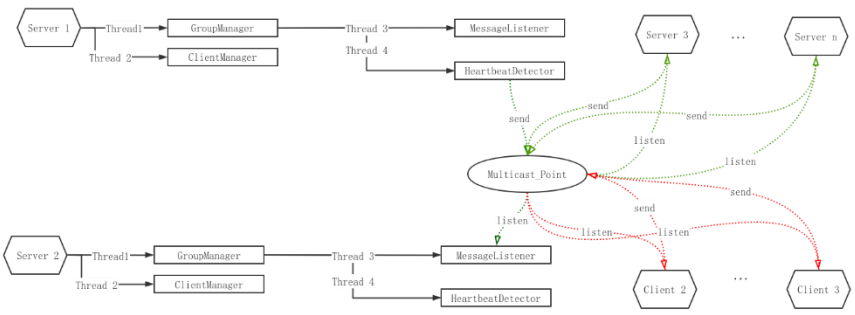

Fig. 6. Heartbeat detection mechanism

Heartbeat detection
Heartbeat detection is a common method to prevent system crashes due to errors in individual nodes. In the project, heartbeat detection is assigned to one of the processes. All nodes periodically send heartbeat detection messages to Multicast Point via HeartbeatDetector. The information is stored in a queue dedicated to heartbeat detection. At the same time, each node's MessageListener periodically reads the messages from this queue. If the number corresponding to a node does not change after several cycles, the node is considered to be in error and a message is sent via Multicast_Point to the clients connected to that node to connect them to other nodes that are still working properly. The failed node will afterwards be deleted in the group view and excluded from the group. If the failing node is accidentally the leader, the other node members in the group which are still functional will trigger another election procedure to select a new leader in the group, and simultaneously update the group view to all nodes.

Replication

When a client message is transmitted between the different servers (since the clients are connected by different servers), the replication manager replicates the message on all servers and delivers the messages to all other nodes. The function of replication in fault tolerance shows the offline after the client. When a client receives a message offline without acknowledgement to the server, the replication manager of the server will automatically store the transmitted data that the client does not receive in a FIFO queue. When the client is online again, the replication manager delivers the data again to the client.

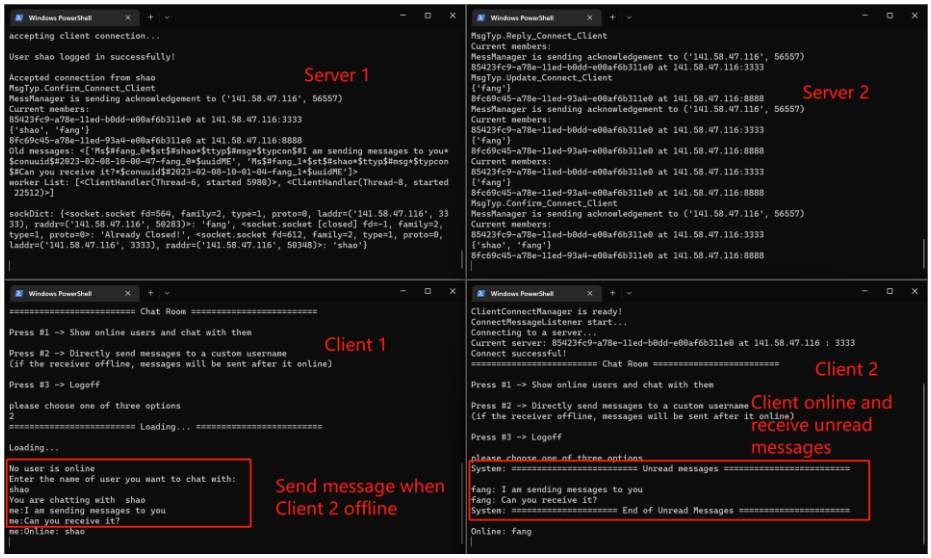

Fig. 9. Replication test justification result

4.4 Voting

When the system is powered on or because the leader node has dropped out due to failure, the election mechanism needs to be activated to elect a new leader node. The leader node can be selected based on the load size or the size of the id. Among them, the Bully algorithm and LaLann-Chang-Roberts algorithm select the node with the largest node id as the new leader node based on the size of the node id. The implementation process of the Bully algorithm is relatively simple. Each node has the right to initiate an election, while each node knows the id size of all other nodes. In the initial election, the node with the largest id is directly declared as the leader node; after the leader node is down, the node that finds the leader node fails first notifies the nodes with larger ids than itself, and then these nodes notify the nodes with larger ids than their own nodes, and until that there is only one living node, this node is elected as the leader node. The problem with the Bully algorithm is that when the leader node is unable to reply to heartbeat detection for a long time due to high load, it may trig-

ger the election mechanism without the leader node being failed, which leads to disordered triggering of an election. Therefore, the LaLann-Chang-Roberts algorithm is used in this project.

The algorithm in the project is the LaLann-Chang-Roberts algorithm with a little difference. In the original LaLann-Chang -Roberts algorithm, the live one or n more nodes would initiate an election when they find the leader does not deliver a new heartbeat signal. The messages – leader fails – are delivered by the live nodes who found the failure. In the project algorithm, all nodes need to detect the failure of the heartbeat from the node and transform the state to election ready. The election mechanism starts only when all live nodes have changed to and confirmed an election-Confirm state, which means the election member list is synchronized on each machine before the election to ensure that the results are the same on each machine

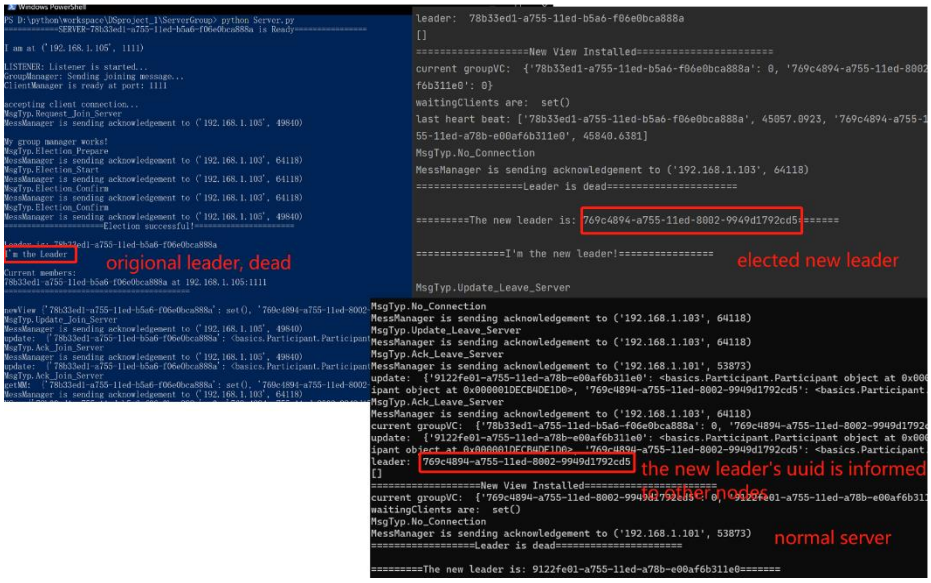

Fig. 10. Heartbeat detection and voting test result

Test of heartbeat detection and voting:

Construct a server network

Force closing the leader of the network, see if the leader election is triggered and a new leader is elected and whether the group view is updated.

4.5 Ordered reliable multicast

To realize the above-mentioned functions, the multicast provides an infrastructure for the following characteristics:

a) Dynamic discover services: multicast messages can be used by servers and clients to locate available discovery in order to register them in the

distributed system. In our project, for the same consideration, the messages used to manage the group members, clients try to discover the group, and the members' information, are sent by multicast.

b) Notifications: multicast to each group member when something happens. In our project, the server's health information is notified by multicast. The client's activities which are not on the same server are also known in this way.
c) Fault tolerance: The multicast synchronizes the identical operations between the members of the group. In this way even if some members fail or are unreachable, clients can still be served.
d) Replication Data: the data will be replicated in the group to increase the performance of the service. For our project, the offline message storing, the cross-server messaging, and a certain connection fault tolerance are realised in this way.

In our project, the multicast dedicates to group communication. Group communication is implemented over IP multicast, attaching some extra information to manage the group, which means our IP multicast allows joining and leaving dynamically and performs address expansion. As well as TCP information for client-to-client messaging is also sent by IP multicast.

For ordering guarantees for replication functions in the closed group after the group is stable, the reliable causal ordering multicast is applied, which indicates the cross-server messaging and offline messages storing. The order for these kinds of messages is relatively important.

In our project, every multicast message will be delivered to the message queue, by using a duplicated check, each message can only be delivered once, which implies reliable communication.

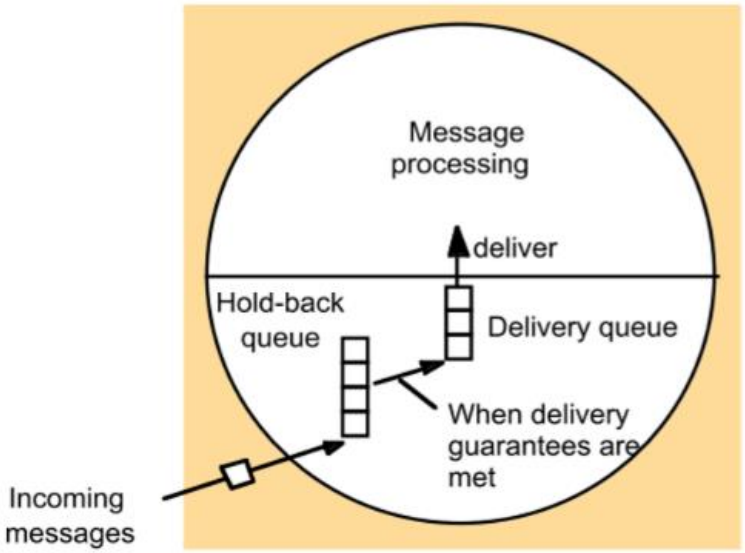

Fig. 11. The mechanism for receiving messages(from slides multicast)

Each incoming message will be first checked whether it is duplicated in the hold-back queue and delivery queue. And then it is checked whether meets the causal conditions: if it meets the two causal conditions, it would be finally appended into the delivery queue.

The conditions of causal ordering using vector timestamps are introduced in the lecture. In our project, each server has its own timestamps. To CO-multicast a message to the group, it adds its entry and packs the message content with vector timestamp then send it. The received message would be checked for two conditions, the first one is earlier messages from the sender have been delivered and any messages that the sender believed when it multicast have been delivered.

To test the reliable multicast, we opened three servers at the same time and logged one client, sending messages to an offline client, and checking the multicast messages on these three servers whether meet the reliable causal ordering.

```
Current members:
7070937f-a705-11ed-bafc-9949d1792cd5 at 141.58.42.240:1111
7c0a256a-a705-11ed-8906-9949d1792cd5 at 141.58.42.240:1212
92b68564-a705-11ed-8bf5-9949d1792cd5 at 141.58.42.240:2222
```

Fig. 12. The three servers we started

To illustrate the test results more clearly, the following figure shows the causal ordering between three servers. The port number is corresponding to the above address of servers.

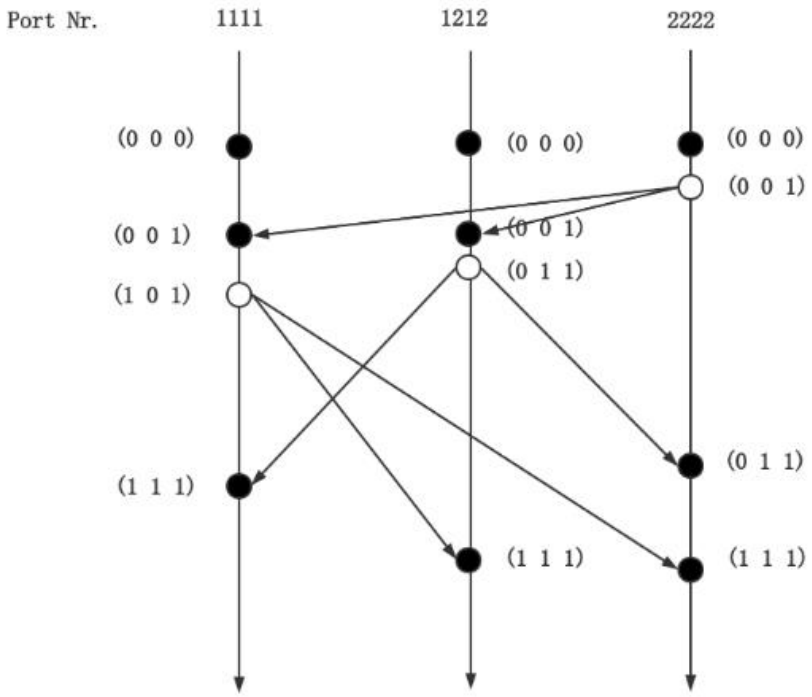

Fig. 13. Test ordering result

5 Discussion and conclusion

Through the development of this project, we have basically completed the simulation and validation of the characteristics and requirements of a distributed system with multiple servers and multiple clients, including the construction of the framework, the implementation of basic one-to-one communication, one-to-many multi-casting, and the verification and simulation of the distributed system Multicast reliability, causality guarantees, and system fault tolerance, which basically meet the requirements of the system characteristics. After discussion among the team members, our team believes that this project system can basically meet the communication needs between clients in the case of fewer clients and servers.

However, in a more complex communication environment, the project may not be able to work under high load because of its low efficiency, and some of the design has over-utilization of thread resources, and there is still a chance for optimization. For example, an optimized Lalann-Chang-Robert algorithm can be used in the election

system to optimize the election time, instead of triggering the election mechanism only after all nodes detect the leader node heartbeat stopped.
For communication, the IP multicast is not enough for the real-world group-manage application. Because it still suffers from some failure characteristics like Omission as UDP. And if the group has more servers at the same time, the causal ordering is much more important. In our project, we only consider the situation of a closed nonoverlapping group for simplification. But in the real world, open-group and closed-group ordering and overlapping members should be all provided.
In an applicable distributed system, not every node is connected to every other node. The communication of the system in this project is designed with Multicast_Point as the centre, where all the nodes read and write information. It is only suitable for use in application scenarios where the number of nodes is limited and the complexity of transmitting information is low. And if this structure is applied in a huge distributed system, it will certainly lead to the load of the central node being too large and cause the whole system to crash. Therefore, a possible future optimization direction is decentralization, de-composing multiple servers into groups, using high-speed communication between each group, and forming a network within the group. This will form a distributed system that takes into account both speed and stability. (4327 words)

Code：https://github.com/Qixiang-ycdi/DSproject_chat.git

References

1. Coulouris, George., Dollimore, Jean., Tim, Kindberg., Blair, Gordon.: DISTRIBUTED SYSTEMS Concepts and Design. 5th edt. Pearson Education, Boston (2012).
2. Python threading, https://docs.python.org/3/library/threading.html, last accessed 2023/01.
3. Python socket, https://docs.python.org/3/library/socket.html, last accessed 2023/01.