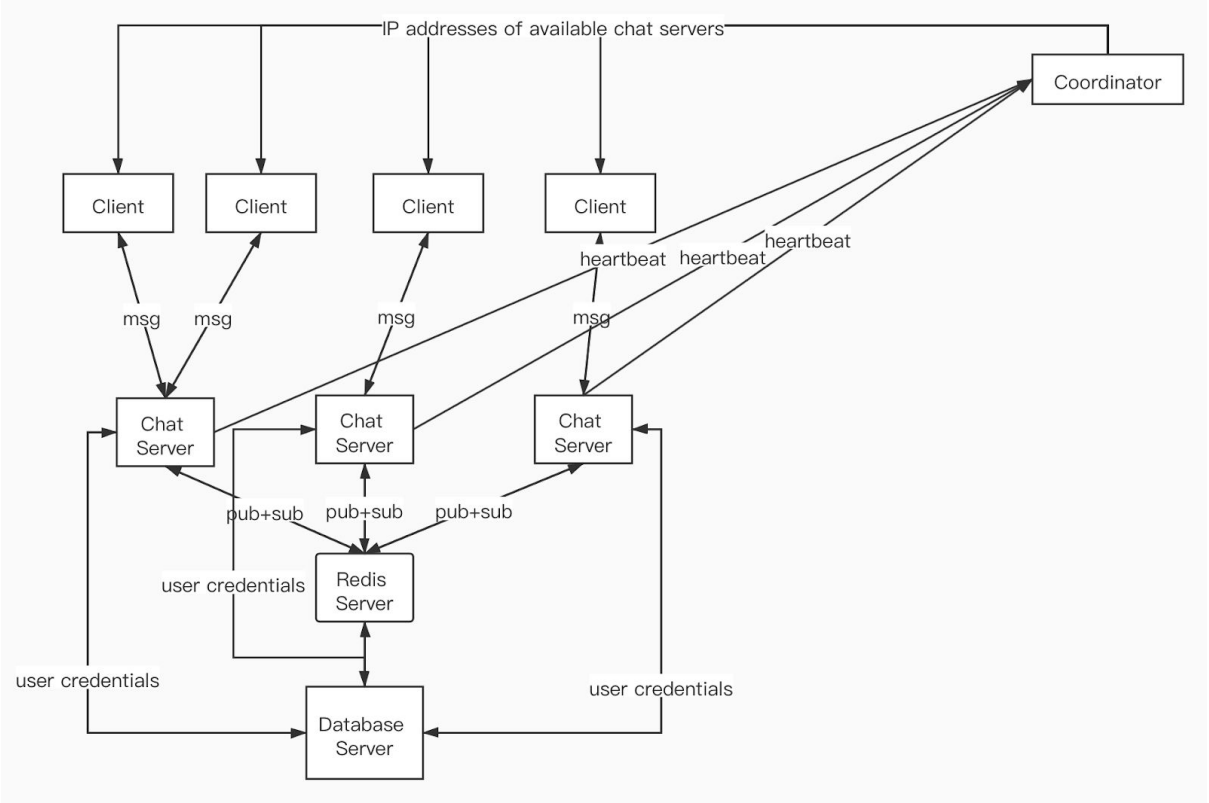
Project title: distributed chatroom

1. The project's goal(s) and core functionality. Identifying the applications / services that can build on your project.

The project is developed to serve as an instant messaging application. Its core functionality is to allow users to send text messages in a chatroom.

2. The design principles (architecture, process, communication) techniques.

❖ Architecture



❖ Process

- From the client side:
  - A client logs in the app.
  - A client sends text messages in a chatroom and all the users in the same room would receive the messages at the same time.
  - A client logs out the app.
- From the chat server side:
  - A chat server becomes a publisher and a subscriber of a channel of a Redis server.
  - A chat server is assigned to serve a client by a coordinator.

- A chat server publishes the messages it receives from the clients to the channel.
- A chat server forwards the messages it receives from the subscribed channel to the clients.
- A chat server periodically sends a heartbeat to the coordinator to indicate its existence.

 From the coordinator side:

- A coordinator maintains a list of IP addresses of the available chat servers.
- A coordinator responds to a client's login request with the IP address of a currently available chat server.
- A coordinator receives periodical heartbeats from all chat servers.
- A coordinator removes the IP address of a dead chat server from the list.

 From the Redis server side:

- It serves as a message broker.
- It temporarily saves messages in a Redis list so that chat servers could fetch them for new coming users.
- It persists the chat messages into the database.

 From the database server side:

- It contains and updates the user information.
- It periodically backs up the chat messages locally.

❖ Communication techniques

 Nodes communicate with each other using TCP socket.

 pub/sub feature provided by Redis.

3. The key enablers and the lessons learned during the development of the project.

The key enablers: a Redis server as a message broker, a coordinator to assign available chat servers;

Lessons learned:

1) There are better alternatives than Redis that could serve as message brokers. It is of great importance to understand the underlying working principle of a component before applying it. For example, Redis pub/sub feature does not provide data persistency functionality, which took us some time to deal with, and consistency of messages could not be guaranteed because of the at-most-once delivery nature of Redis pub/sub.
2) Understand the core requirements before actually coding.
3) Single point of failure should be seriously taken care of in a distributed system.

4. How do you show that your system can scale to support the increased number of nodes?

Our system should be able to scale horizontally because a chat server could be set up easily (codes of servers are the same) and therefore, all chat servers here are homogenous and are acting the same. The scalability is demonstrated in the demo.

5. How do you quantify the performance of the system and what did you do (can do) to improve the performance of the system (for instance reduce the latency or improve the throughput)? (Extra points for improvement).

We are able to quantify the performance of our system by throughput, which measures the number of messages that could be sent per second.

We quantify the throughput of our system through the following two experiments:
Experiment 1:

- Publish 100, 500, 1000, 5000, and 10000 messages of 100 bytes to the Redis server in every 200 milliseconds.
- One server subscribes to the Redis channel and redirects subscribed messages to 10, 50, 100 clients.
- Measure the average time that clients need to receive all messages
- Result:

client_num / msg_num	10	50	100
100	2.17s	2.17s	2.22s
500	11.05s	11.07s	11.20s
1000	22.33s	22.15s	22.25s
5000	110.57s	111.04s	110.14s
10000	220.91s	221.55s	220.32s

- Conclusion: Server throughput seems to be quite stable, the increased number of clients does not affect the throughput a lot.

Experiment 2:

- Publish 1000, 5000, and 10000 messages of 100 bytes to the Redis server in every 200 milliseconds.

- Run 1000, 5000, and 10000 (using Python threading) to subscribe to the Redis channel.
- Measure the average time that servers need to consume all messages.
- Result:

server_num / msg_num	50	500	1000
1000	21.88s	33.97s	90.30s
5000	111.63s	213.99s	433.80s
10000	238.05s	436.49s	827.00

- Conclusion: The increased number of servers (i.e. subscribers) significantly degrades the throughput of the Redis server.

For improvement: we could manage to do this by developing a single Redis server to a Redis cluster which provides more capability to forward the messages. By applying a Redis cluster, the workload of a single Redis server could be distributed into multiple ones (Redis cluster bus is used to propagate pub/sub messages across the cluster), which would increase the throughput of our system.

6. What functionalities does your system provide? For instance, naming and node discovery, consistency and synchronization, fault tolerance and recovery, etc? For instance, fault tolerance and consensus when a node goes down.

Functionalities of our system:

1) Naming and Node discovery: Each server has a unique ip. The coordinator is responsible for node discovery by receiving heartbeat messages from chat servers.

2) Synchronization: The host machine is used as an NTP server, the clocks of the virtual machines are synchronized with the host machine. Chat servers use the NTP time to add timestamps to messages before publishing them to the Redis server. In this way, users can determine the order of messages even if they are delivered out of order for some reason.

   We are using Redis server to serve as a message broker and therefore consistency is somehow not guaranteed here by Redis because of its at-most-once delivery nature. We are planning to replace Redis to another message broker or to introduce some mechanisms which provide at-least-once delivery.

3) Fault tolerance and recovery: all nodes of our systems could possibly go down.

For chat servers, if a chat server goes down before a connection is established with a client, we are able to make sure clients only access the available chat servers by asking the coordinator to collect the heartbeats of all the chat servers so that if a chat server goes down, the coordinator would soon know and respond. If a chat server goes down during the connection with a client, the client will detect it itself and request a new valid chat server ip from the coordinator.

For coordinator and Redis server, they remain as single points of failure in our system. For the database server, there is a periodic backup mechanism in our system to make sure the recovery.