Distributed Systems, Project Description, Group 4

Kozheen Taher Esa
Joonas Ahovalli
Roni Tuohino

p2p-chat

We are developing a peer-to-peer (P2P) chat application. The main functionality is creating networks (Networks / Groups / Chats), which can be thought of as group chats. A client (Node) can join a network by contacting the leader of the network. The leaders of each network are elected from the nodes of the network using election algorithms. The leaders of all networks are stored in a central Node Discovery Service (NDS) through which a Node can discover existing networks and the addresses of the leaders.

The communication within a network happens in a P2P manner, so the NDS is only used to get the addresses of the leaders of different networks. With this architecture, the application can be considered to be a hybrid network overall.

The implementation requires the following services:

- A centralized Node Discovery Service (NDS)
- A client service (Node)

The NDS is quite straightforward to implement as a data store. However, the node service is more intricate. The node service needs several modules to fulfill the needs of the application, such as ui, messaging, election, and liveness. The next chapter “minimum functionality” describes how the system works as a whole, and the interactions between the Nodes and the NDS.

Minimum functionality

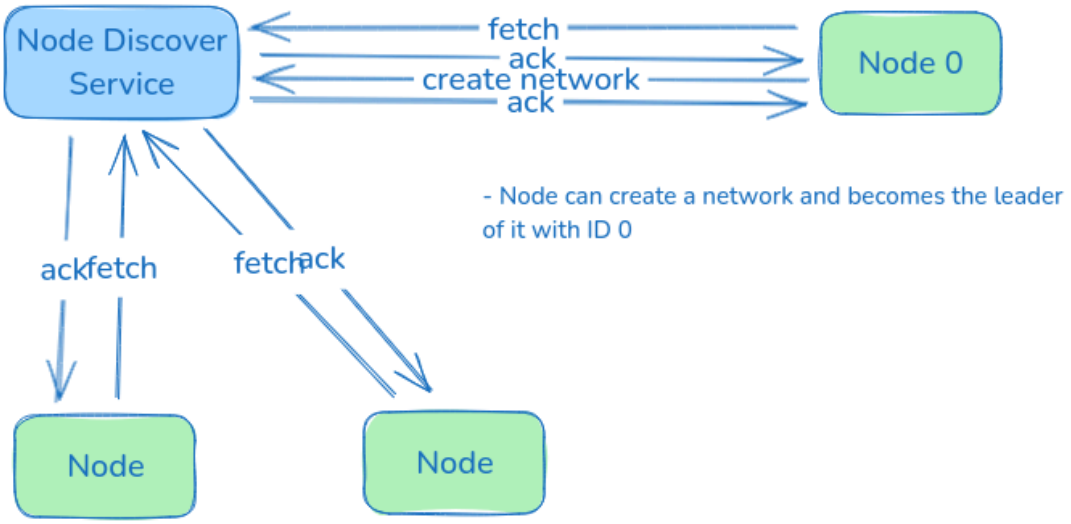
All messaging is synchronous unless specified otherwise.

Starting up the service

The p2p-chat service is started by starting the Node Discovery Service (NDS).

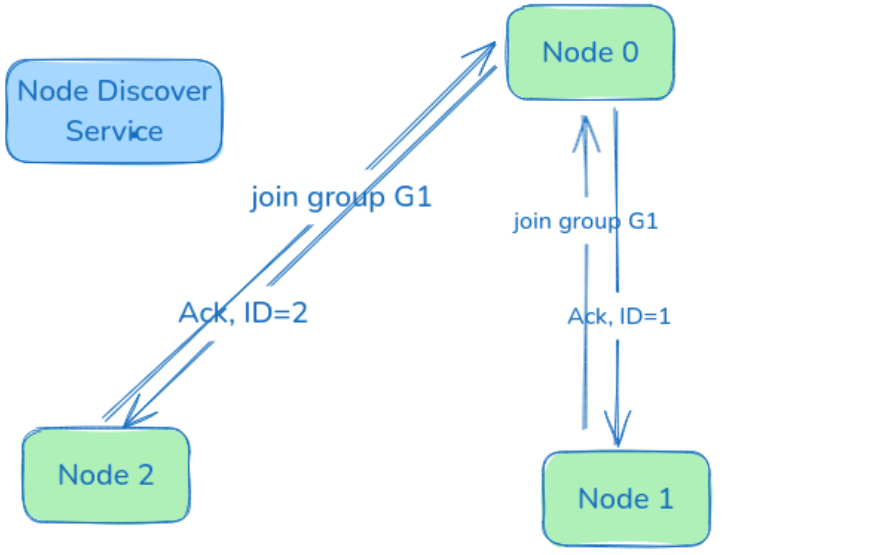
Creating a network

A new network G1 (group chat) can be created by a node (N1) by sending a request to the NDS. N1 becomes the leader of G1.



Joining the network

1. N2 pings available networks from NDS -> [G1: N1, …].
2. N2 joins the network G1 by sending a request to the leader of the network (N1).
3. N1 has to inform all other nodes in the network that N2 has joined.



Nodes request to join a group network. Leader responds by acknowledgement and giving the node ID, and incrementing ID by 1.

Leaving the network

1. N2 can leave the network G1 by sending a request to the leader of the network (N1).
2. N1 has to inform all other nodes in the network that N2 has left.

Liveness ping

When a node joins the network, it starts a timer with a random value in some range (t1, t2). If a node receives an ELECTION or COORDINATOR message, it resets its own timer to a new random value (t1, t2).
After the timer runs out, it pings the current leader of the network to check the connection. If the leader is not reached by some node, it starts a leader election.

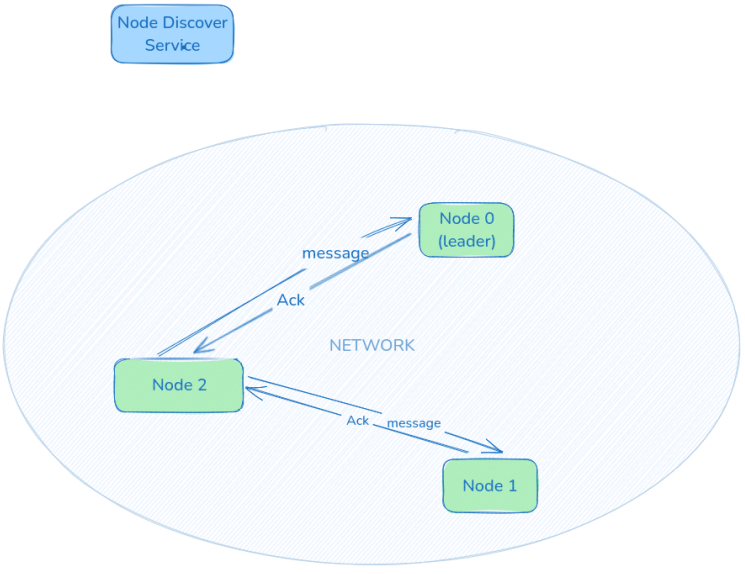
Leader election using Bully

1. Elect leader using Bully algorithm, but choose the leader with the LOWEST id.
2. The new leader requests a NDS if it can reach the previous leader. If not then is the new leader of the network. If the NDS can reach the previous leader, then the new one is not made leader.
3. Each node is assigned an id when they join the network, which is the highest of all in the network. This prioritizes nodes as leaders that have been active in the network the longest.

Chat messaging

Assume network G1, with nodes N1 (leader), N2, N3, N4.

1. N3 sends a message to the network by sending it to each node separately [N1, N2, N4]. If an ACK is not received from some node (N4), the message is sent to the leader of the network (N3 -> N1), which then passes it to the node (N1 -> N4).
2. If the node is not reached by the leader, then we assume that the node dropped out.
3. The leader then sends a message to all other nodes that N4 is no longer in the network.
4. Each node stores a copy of the message log locally.
5. The ordering of messages is not guaranteed, however if time allows we might implement it with vector clocks.



Additional features

- Using Distributed Hash Tables for messaging to achieve persistence. Message replication and synchronization?
- Admin rights?
- Message encryption?
- Message deletion?
- Authentication to a network?
- Could this be fully P2P without a centralized NDS?

Technologies

Python is a simple programming language which we can all program in. Using it results in faster production. It also provides a large amount of support for any use case. For example, the RPC protocol is already supported. It has one of the largest communities that keep it up to date with state-of-art.

The client's interface is going to be a CLI. From a design perspective CLIs are 🔥 Additionally, we want our chat to support even the most enthusiastic programmers that live only in Unix terminals. Using a CLI as the UI is also going to make debugging easier, when running the clients in remote virtual machines.

We have chosen to use RPC for messaging between the services, because it has been known to be suitable for microservice architecture due to being lightweight. Sockets for example keep a constant connection between clients, which is not ideal for our chat application. We have also never used RPC previously, so we want to learn how it works. gRPC is an implementation that we plan on using, because it is optimized for low-latency communication. Additionally, it supports both asynchronous and synchronous communication that we can leverage if required.

Messages

create_network(
    ip: str,
    chat_id: uuid,
    chat_name: string
) -> OK / FAIL

A client sends a request to create a new network to NDS, if FAIL return FAIL otherwise create a network with chat_id and set the ip as leader and return OK

join_network(
    leader[chat_id]
) -> OK [node list (ip+id)] / FAIL

leave_network(
leader[chat_id]
) -> OK / FAIL

election() -> OK / FAIL
coordinator() -> OK / FAIL

get_network_information() -> OK / FAIL
A client requests possible chats from NDS.

liveness_ping(
) -> OK / FAIL
If the liveness ping fails, a leader election is started by the node that made the ping.

nds_update_leader(
node_id: int
) -> OK / FAIL
NDS update leader checks if the leader is alive before updating the leader, if leader is alive, then return FAIL otherwise update the leader to be new leader and return OK

message(
chat_id : int,
msg_id : uuid,
source_id : int,
destination_id : int,
msg : Any
) -> OK / FAIL