Final Report
Distributed systems course
Master's Programme in Computer Science

Norris Chat System

Niko Häppölä
Kristian Krok
Aksel Linros
Laura North

22.12.2021

Table of contents

1. **Goals and core functionality** **3**
2. **Design principles** **3**
3. **Functionalities** **5**
4. **Scaling** **6**
5. **Performance** **6**
6. **Lessons learned** **7**
7. **Appendix: Advancing from the design plan** **7**
8. **Links** **7**

1. Goals and core functionality

Norris Chat System is an instant messaging system for *Distributed Systems* -course in University of Helsinki. The goal of the project was to build a distributed chat system, where users can broadcast messages to every other user in the network. All users are free to join and leave at any time, and will see the messages sent in the same order as everyone else. Receiving messages happens only when the user is online, and the system does not save any previous messages.

In a way, the system works similarly to the legendary IRC-system. The main differences are that IRC consisted of multiple servers that were online simultaneously and handled all the messaging, but in Norris Chat System each node works on its own home server and there is no centralized system to work as a host or group of hosts.

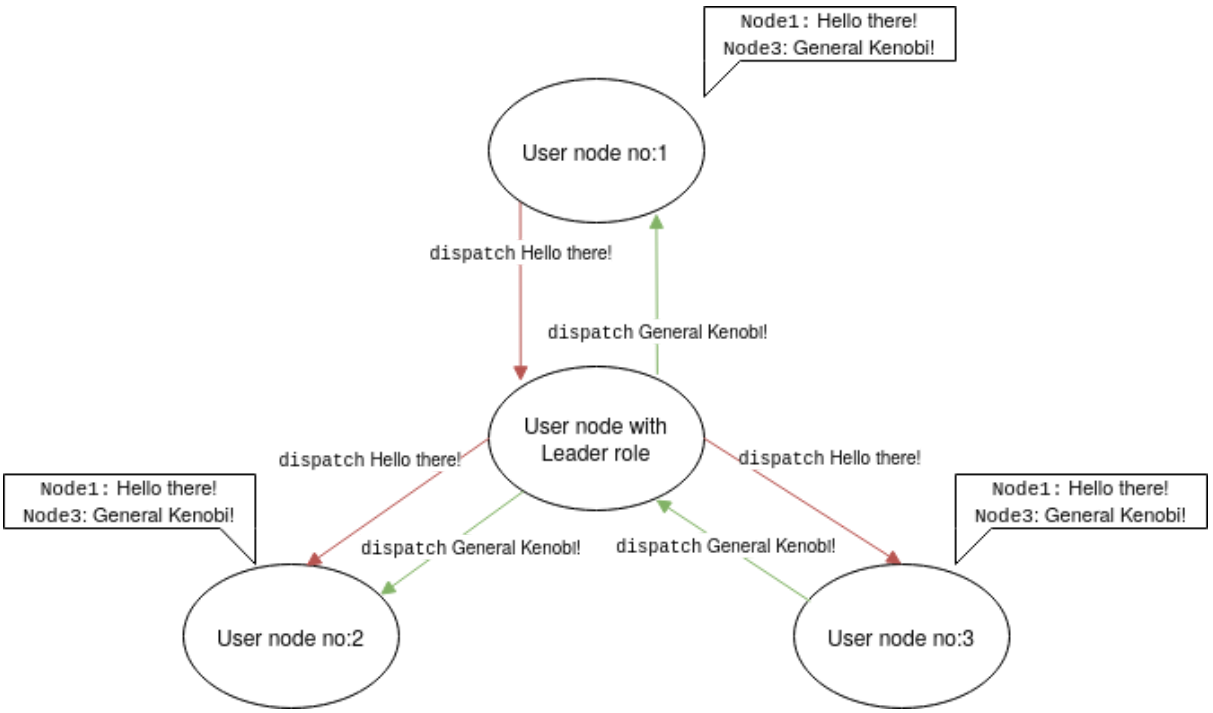
For the system to go online, a first node must join. It assumes a role as the leader (host) node. A leader node is responsible for maintaining information about other nodes present in the system. It also handles all the messaging and the status of the system. Each node has the functionality to be the leader node if the current leader node becomes unresponsive for reason.

2. Design principles

Architecture

The application builds on different nodes, the number of which depends on the number of users online.

The project is implemented using Python and its native socket library. The source code and subsequently an arbitrary host on which a system node for the project is run is runnable on any OS capable of running python 3.8 interpreter and providing a full TCP/IP stack.



Nodes

A user is a single node in the system. There are no centralized servers, so the whole network consists of user nodes. All the nodes are pluripotent, so depending on the situation they can take either of these roles:

1. User node. The normal node type that can send and receive broadcasts from the network. When a new user node connects to the network, it will be given a unique identifier by the host, which is shown to all other nodes when a message is sent.

2. Leader node. This is a special role for one user node that is needed for initializing connection of a new node. When a request to join the chat system comes, the leader node will respond with information about the network, so that the new node can initialize communication paths. There must always be at least one leader node, and the first node to start the network automatically gets assigned this role. In addition, the leader node has all the user node capabilities, and from the usage point of view does not differ from a user node.

Messages and communication

In a chat message, a node sends a message containing its identifier and the message. The system uses the user node's IP address as identifier and labours under the assumption that no two nodes originate from a shared IP address. The messages are sent to the leader node, which delivers them forward to the whole system.

The system aims to deliver all chat messages in chronological order. However, the system assumes that the arrival order of the messages is the correct order and delivers them accordingly.

The messages are delivered using TCP/IP and TCP/UDP protocols. End-user messages from client to leader and thereon from leader to the rest of the clients are transmitted over TCP protocol. Under the hood system communication, periodic ping messages from current leader to client nodes, utilizes UDP protocol.

Limitations

The number of users isn't limited. The system aims to allow an arbitrary number of simultaneous users. However, a significant number of simultaneous messages may prove to be difficult for the system to handle. This can be something that could be looked into further in the development.

The system does not provide a feature for the user to delete or edit messages that have already been sent. The system does not persist a message log, hence a fresh user will only see messages that have been sent after they have joined the system.

Process

The design process was done in an iterative manner and due to limitations in the time available and our know-how, we kept revising the design plan and what to implement up to the very end. Conjuring the initial design plan was a tad bit challenging as the requested system requirements were not fully comprehended by the team. After the theory side of the course progressed past the first few weeks, our understanding of distributed systems in general, and how to iterate the design plan towards a more reachable goal, increased.

Technical limitations, due to lack of know-how and lack of past experiences with the domain, affected the state of the design plan throughout the project. Threading and socket based communication proved to be a pickle to tackle and this was reflected in the design plan’s lifecycle in such a way that there were no fixed system functionalities - everything was extended with a “if we’re able to implement” -clause.

3. Functionalities

Naming

The leader node is in charge of naming. Whenever a new node joins the chat, the leader includes it in the list of nodes. The leader maintains a list of nodes currently active in the system. The list contains a user nickname - IP address tuples. The leader node is capable of transmitting the node list to all

connected notes whenever a new node joins or leaves the system. Having an up-to-date node listing on every node in the system facilitates, in its part, leader election that should be held if the active leader becomes unreachable.

Node discovery

When a node joins the chat system, it contacts the presumed active leader. If no response is received the node has the capability to start functioning as a leader. In other cases, there is already a leader and the node assumes a user role.

The system relies in its current state on a hand-appointed leader. As the first leader is hardcoded, there should not occur a case where no response is received.

Consistency and synchronization

All chat messages sent by the end-users go through the leader node and the leader delivers the received messages to all other nodes. The messages are sent through a TCP connection and the system utilizes TCP protocol's built in message delivery and acknowledgement properties to ensure that all chat messages get delivered.

Fault tolerance

When the node that has the leader role fails for some reason, another node must take the role of the leader node for the messages to be distributed and new nodes to be accepted.

The current leader node sends pings every 15 seconds (can be adjusted if needed) for all other nodes. The pings are sent via UDP connection and no response is needed for now. All other nodes are given a timeout window and if they do not receive a ping during that window, they will assume that the leader is faulty. The system has nearly finalized leader switch functionality. However, as there are no implemented consensus mechanisms, a vote for a new leader and automatic leader switching is currently not supported.

Consensus

In a fault-tolerant process group, each non-faulty process executes the same commands, in the same order, as every other non-faulty process. Formally, this means that the group members need to reach consensus on which command to execute.

In the chat system this would mean that each message is timestamped and that they are organized based on the timestamp. If a message with an earlier timestamp arrives at a node after another message with a later timestamp, the system will organize the messages based on the timestamps. The

system, however, lacks the timestamp-based message ordering and thus all chat messages are perceived as concurrent.

4. Scaling

The chat system uses the Python socket library and its BSD socket implementation. It enables a server to accept connections. The function also specifies the number of unaccepted connections that the system will allow before refusing new connections.

At its current state the system is able to support up to five user nodes. Five here means that five connections are kept waiting if the server is busy and if a 6th socket tries to connect then the connection is refused.

The number can be scaled up to 128 or even 4096 nodes. However, with using only one thread, such a large backlog isn't recommended. The code could be developed to spawn extra threads for more connections to be able to connect without building such a large backlog.

5. Performance

For the performance we considered the eight network fallacies[1]. Network problems form an issue that we cannot tackle in this project. A little latency is tolerable in our project and it is handled in the consistency and synchronization section as well as in the consensus section. Same applies to bandwidth limitations.

We cannot do much for the performance in this project. We can adjust the ping interval as well as the number of nodes able to join. The system does not produce any considerable overhead and thus it performs reasonably well for its core functionality, a chat system.

6. Lessons learned

Distributed systems means distributed problems.

We were too optimistic of the tools and the amount of time we had to finish the project. In the beginning we struggled to understand what a distributed system was and what kind of project could be made as a distributed system. We had multiple good ideas that we did not have time to finish.

We used a lot of time for planning and discussing. We should have started coding earlier. However, the course progressed more leisurely in the

[1] https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained

beginning and thus we wouldn't have had enough knowledge to make a distributed system at that point.

We discussed the programming languages we would like to use and ended up with Python. This did cause some challenges and maybe another language could have been chosen. However, in the end we didn't figure out any better languages either.

It can be said that this course and the project did require A LOT of time. We did enjoy our discussion and coding sessions, but they did consume a huge part of the week. A key takeaway from this project as lessons learned is that in future projects, do start working on proof-of-concepts/exploratory code snippets as early as possible. This enables the group to identify possible know-how caveats earlier in the process and subsequently saving suboptimally spent time in the later stages of the project.

7. Appendix: Advancing from the design plan

The main architecture of the Chat System did not change during the development. However, in the beginning of the course we weren't aware of the possible ways the communication should be organized between the nodes, and what would be feasible to implement. The plan was that all nodes would message all nodes. However, we ended up giving the host node more responsibility and all the messaging goes through the host node. This is the main difference from the design plan to the end report, though maybe this plan was not that visible in the beginning.

8. Links

Repository:

https://github.com/kriskrok/norris-chat-system

Video:

https://helsinkifi-my.sharepoint.com/:v:/g/personal/linaksel_ad_helsinki_fi/EZEQqgL1KyJMtVmnUOl9-AUBMUi8mi2ONPccE7c8zdZj-g?e=HJoFIP