Group 8 - Project Final Report: Peer-to-Peer Chatroom

Project Team: Roman Hetzler 804143, Michael Lukic 763563, Luke Roy 806398

1 Introduction

For the programming project for Distributed System, we have implemented a Chatroom using advanced principles of distributed system development. For example, UDP and TCP for message transmission, fault tolerance with heartbeat and leader election Using the LaLann-Cheng-Roberts Algorithm. Participants are able to join the group conversation and send as well as receive messages from the other participants. The first Person to create the Chatroom is the leader and all the other people can join and communicate with each other. To better represent components from our code, functions are in italics and with round brackets at the end (*function( )*), variables are only in italics (*variable*) and commands are uppercase and in quotation marks ("COMMAND").

2 Project Requirements Analysis

2.1 Dynamic discovery of hosts

When the system is started, it sends a broadcast message to all participants and only the leader responds. This action is tried 5 times. If the current leader does not respond, the participant himself becomes the leader. Otherwise, the leader sends back a message with its response code and IP address. This functionality implements the dynamic discovery of the participants.

2.2 Crash fault tolerance

We have implemented a heartbeat to check if all participants are still in the chat. This is realised by sending a ping message to its neighbour (ring of participants) every second. The ping does not contain a message, we just want to use the TCP functionality to confirm that the message has been received. If the node doesn't get a reply after 5 messages, he assumes that the neighbour has left the chat. If this neighbour was the leader, he starts a new leader election and votes himself as the leader. This ensures that the system always has a leader and crashed participants are detected.

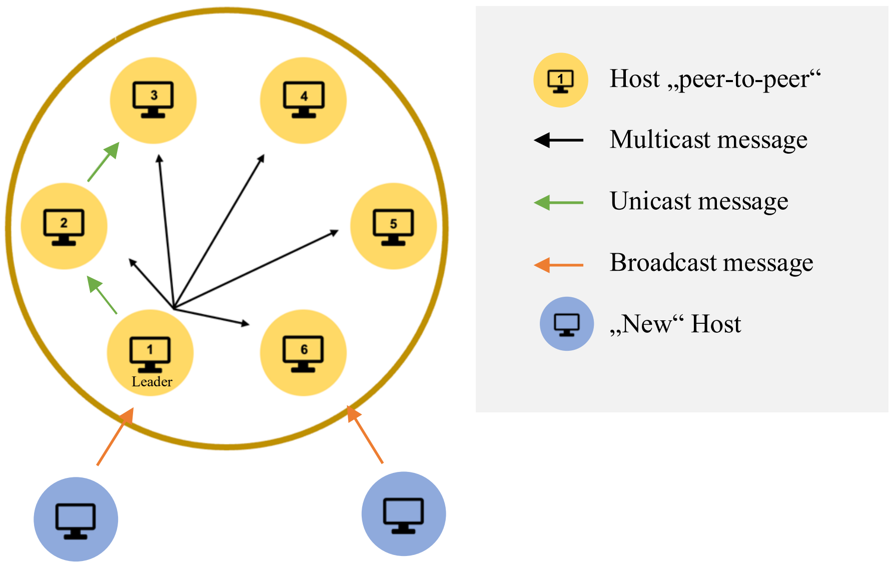
2.3 Voting

Initially, the first participant who starts the chat application is the leader. If the leader drops out, either due to network problems, crashes or leaves the chat, this is detected by the previous node in the list via the heartbeat. The neighbour is always the next higher node address stored in the peers list. The node which identified the missing leader starts the voting afterwards. If there originally were only two nodes in the chat, the remaining node automatically becomes the leader. If there is more than one participant left in the chat, a TCP message is sent through the ring to all participants performing the leader election. This is based on the LaLann-Cheng-Roberts algorithm and will be explained further in the implementation section.

2.4 Ordered reliable multicast

We did not implement ordered reliable multicast because it is not important in which order the messages are received. Why we have omitted this functionality will be further elaborated in our discussion and conclusion.

3 Architecture diagram



We use the peer-to-peer concept for our architecture and therefore we don’t need separate servers and clients because both components are contained in a peer. The information is then exchanged directly between the participants (peers) via broadcast (join chat), multicast (message) and unicast (heartbeat, voting). The peer-to-peer network is generated by a list of participant addresses, so each participant knows who else is participating in the chat.

This concept also offers advantages of fault tolerance because there is no single point of failure. Since all participants can perform the same functions. If the leader fails, another participant can take over directly and nobody notices the failure. The network is built up as a ring, where the participant with the next higher IP address receives from his predecessor and is monitored by the Heartbeat. The ring is used by the voting alg. of the leader as well as by the heartbeat. Participants broadcast a message to other participants that they want to join. But only the leader responds to it and informs the other participants about the new addition. Our system was set up as a monolith in order to keep operation and distribution as simple as possible.

4 Implementation

4.1 Set-up, Requirements and Testing

For the implementation of our chat application we used Python as the programming language. Only the following standard libraries were used: *Socket*, *Threading.thread*, *os*, *struct*, *time.sleep*, *sys* and *ast*. This has the advantage that no third-party modules need to be installed and the system can be used directly with the standard Python installation.

The application was tested under Python 3.10 on a Linux x86 and a Linux ARM device, whose distribution is based on Debian, as well as under macOS 13.1 Ventura with an x86 architecture. The functionality of the application was successfully tested with four separate devices.

Setup process under macOS or Linux:

1. We assume Python 3.10 has successfully been installed and the source code which is provided in the main.py file. In addition, at least two physical nodes are required for meaningful use.

2. Calling the main.py by running "Python3 main.py" (We assume that Python 3.10 is set as default).

3. Now the chat can be used and messages can be sent and received.

4.2 Internal system procedure

In our implementation, there are six core functionalities, as follows:

Join Chat:
Initially, when the program is started, the function *startup_broadcast( )* is called and a UDP broadcast socket is created. Then a broadcast message is sent with the intention to find the leader and join an existing chat. To ensure that the message arrives, up to 5 retries are performed. If no leader is found after the 5 retries, the user becomes the leader. If a leader is found, the user receives a UDP broadcast message back containing a response code, the own IP address and the port of the leader. These values are separated by an underscore. The user then sends a TCP message with the request "Join Chat" and sets the leader.

Add participants:
The *broadcast_listener( )* runs in its own thread and creates a UDP broadcast socket, listening continuously for broadcast messages from new participants. If the broadcast listener receives a message and is the leader, it sends back the response code with the new user's IP address and the listener port. If it is the leader and receives a message, it sends back the response code with the user's IP address and port. Otherwise the message is not processed.

The TCP Listener runs in its own thread and creates a TCP Unicast socket. It listens permanently for unicast messages from other nodes. When it receives a message, it responds depending on which command is contained in the message. A message always contains a coded dictionary, which contains at least the fields Command and Contents. The leader receives a message from the new participant with the request Command "JOIN". Then it sends a message via TCP to the new participant containing the current state, which is a list of all current participants. In addition, a multicast message with the address of the new participant is sent to all other participants. The peers receive the "JOIN" message and adds the address of the new participant to its list, if the address was not already in the list. They also call the *find_neighbour( )* function to get their new neighbour if the neighbour has changed.

Fault Tolerance by Heartbeat:
The *heartbeat( )* function runs in its own thread in a continuous loop. If the node has a neighbour, a TCP message with the command "PING" is sent to the neighbour every second. The neighbour receives the "PING" message with its TCP listener. The receipt of the "PING" message is confirmed by the TCP functionality of the sender. If no connection can be established or a timeout occurs, the counter for missed heartbeats (*missed_beats*) is incremented by one. If no error occurs, the counter is set back to 0. If five heartbeats are missed, the neighbour is removed from the list of participants and

the counter is reset. Additionally a TCP message with the command "QUIT" is sent to all remaining participants with the address of the former neighbour, which has left the chat. Participants who receive this message remove the unreachable participant from their peers list. Then the *find_neighbour( )* function is called to update their neighbour. The node that detected the departure of the former neighbour also calls the *find_neighbour( )* function to locate its new neighbour. If the departed node was not the leader, the detecting node starts sending the heartbeat to the new neighbour as if nothing ever happened.

Leader election:
If the current leader node becomes unavailable, the node that discovered the drop starts a new election process and votes for itself as the leader.

The *vote( )* function checks whether we have a neighbour or not. If we do not have a neighbour, we become the leader ourselves as we are the only chat participant left. Otherwise, we compare our address with the address we voted for. We then select the highest address. If the voted address is not equal to our address or the variable *is_voting* is not "True", a TCP message with the command "VOTE" is sent to the new neighbour with the suggested address and the info that no new leader has been appointed yet. After that the *is_voting* variable is set to "True" indicating that the system is currently in the voting process.

The neighbour now receives the request to participate in the election of a new leader via its TCP listener and processes the "VOTE" command. If no leader has been elected yet, the participant checks whether the received address is higher than its own address. If this is the case then the *vote()* functionality is executed with the received address just like the previous node. If the received address is lower, the own higher address is used for the *vote( )* function. If you receive your own address, you win the election and set yourself as the leader. In addition the *is_voting* variable is set to "False" indicating the election is completed. If the new leader has a neighbour a TCP message is sent to it containing the new leader address and the *leader_elected* state set to "True" informing them that the election is over. The neighbour receives this message. If the new leader address is not equal to the current node address, the received address is set as leader and the message is forwarded to the next neighbour via the TCP socket. This continues until the message arrives back at the leader. After that the leader does not send the message further, because all participants have already been informed. This completes the leader election process according to LaLann-Cheng-Roberts.

Send message:
The *transmit_messages( )* function runs permanently in its own thread and waits for user input. When a message is entered it is checked for its size in order not to exceed the buffer size of 4096 bytes. If the message starts with "cmd:", it is identified and evaluated as a local command to view system information and configure various

functions and variables. These commands will be further discussed in their own paragraph. Otherwise the message is sent as UDP multicast consisting of the command "CHAT" and the actual message that was initially entered to the other participants.

Receive chat message:
The *multicast_listener()* also runs continuously in its own thread like the other listener and sender functions. If a message is received, it first checks whether you are the sender. If this is the case, the own message is ignored. Otherwise the received message is decoded and the contents are displayed at the console output.

Command functions:
To help the user and debug during development, we have implemented the following Command functions. These can be used by entering "cmd:" followed by a keyword.

- cmd:clear → Clears the Console
- cmd:ip → Shows you your IP Address
- cmd:ports → Get Broadcast Port and Multicast Port
- cmd:peers → Shows the list of participating peers
- cmd:neighbour → Shows you your neighbour
- cmd:is_leader → Shows you if you are a leader or a participant
- cmd:toggle_debug → Activates the debug mode
- cmd:quit → Leave the chat

Helper functions:

- *debug( )*: toggleable yellow terminal output, which can be controlled by the *toggle_debug* variable.
- *out_cmd()*: Outputs the answers of the self-implemented command statements ("cmd:") in dark blue.
- *out_info()*: Outputs information for chat participants. For example info about joining, leaving and initial welcome message in light blue.
- *info_help(*): Outputs the list of executable commands with short description.
- *get_ip_address( )*: Uses a UDP socket to read the IP address of the host and returns it.
- *encode_message( )*: This function writes the command, the address of the sender and the content into a dictionary and encodes this as binary data. Afterwards the binary data is returned.
- *decode_message()*: Receives the encoded binary data and decodes it to a Python dictionary. Finally the dictionary is returned.
- *format_join_quit( )*: Writes the values *inform_others* and the target *address* into a dictionary and returns it.

5 Discussion and conclusion

In conclusion, we can say that we have accomplished the requirements and implemented our project as defined in the project proposal.

These implemented features include: Dynamic discovery of hosts, Crash fault tolerance and Voting.

In our project proposal we initially wanted to implement the ordered reliable multicast. We did not implement this in our application because we felt that the feature would not significantly improve the experience of using our chat.

For the implementation of our chat we have chosen python for the following reasons. Python is easy to understand and use for all of us, since we all have at least basic knowledge of this programming language. Since it is long established language with strong community it has good documentation and many other resources to help during development of the chat.

We have chosen the host address as node identification, because in our scenario it serves as a unique identifier since only one chat client will be used and it already contains all the important information needed to communicate.

We decided to implement all the functionalities of our chat application in a single file following a monolithic architecture.

Which provides the following advantages:

> It is less effort to test the application on different devices during the development and testing phase as only one file has to be provided to the test environment. In addition, the final distribution of the chat application to users is much easier, since only one file needs to be distributed just like during the development and testing phase.

On the other hand, there is a major disadvantage to this monolithic approach of our chat application:

> Since the complete code base is written in one file, it is considerably more difficult to keep track of its functionalities, which means that new developers who need to adapt the code will have a much harder time understanding and working with it.

Since our application is a mere 600 lines, it is still possible to understand the code with acceptable amount of effort, even for new developers with excising python knowledge. However, if the code had been significantly larger, it would not have been maintainable in one file and we would have had to be split up in several files.

In conclusion, we can say that there are many different ways to build a chat application. We are also aware that our application is not perfect and that there are some areas that could be improved, but in general, we as a team agree that our design, implementation and execution of our application has been a great success for the users of our application.

6 Source and Demo-Video

https://github.com/skywalkeretw/DBE-Distributed-Systems

Word count of the report:
2613 words