DBE - Distributed Systems

Group 07: Kevin Ross, Simon Haupt, Zacharias Sitter

1 Project Proposal

1.1 Problem Statement

How to design and implement a chat application using central and advanced principles of distributed systems development?

What we need to develop is a heterogenous but also open system which has to comply the security requirements of availability, confidentiality and integrity. The distributed system we create must be fulfill transparency and concurrency. Furthermore, it must be resistant to faults and has to be scalable. While building a distributed system it is a difficult task to fulfill all named requirements as you can see from other examples like WhatsApp.
In the following report, our task was to describe in detail how we worked around these problems and how we rebuilt our chat application.

1.2 Project Description

Our idea was to develop an exchange possibility for students in form of a chat application. The focus should be mainly on the exchange of important data and less on the entertainment possibilities. For this reason, it is important that this application works and is available without complications.
The application should allow users to communicate via text messages. The idea is that the users register themselves. If someone new joins the conversation, this is written in lists and shared with all servers. Afterwards the communication runs over the main server, which receives the messages and transmits them to the addressee. For the protection of the data further backup servers are installed, which get their information over the main server.
In the next parts of this report the technical requirements will be explained. The decision of the specific characteristics and other possible options will be focused. The literary foundation of this report is based on the material from the distributed systems course in the SS2022 including the lectures and the book “Distributed systems: concepts and design” by Coulouris Dollimore and Kindberg.

1.3 Project Requirements Analysis

1.3.1 Dynamic Discovery

- Client registers itself by sending username and IP to server
- Server receives various incoming client's requests
- It stores the information of the clients IP address and name in a list
- This List is **broadcasted** to all servers
- It also keeps track of various chat rooms, providing a group IP to each multicast group

1.3.2 Crash-Fault-Tolerance

- The failure of a participant must not cause a crash of the entire system
- In the failure case a new leader election should be triggered
- Heartbeat messages sent by the leader as a failure detector when a participant crash
- Backup server which replicates the data

1.3.3 Voting

- Based on the LCR (LeLann-Chang-Roberts) algorithms
- Every server must have a unique ID (UID)
- If the leader server goes down a new leader will be elected
- Clients can trigger election if the server does not respond
- After trigger neighbor will be searched and pinged
- Voting message will go around until a server get its UID back
- After election leader announces itself as a leader

1.4 Architecture Design

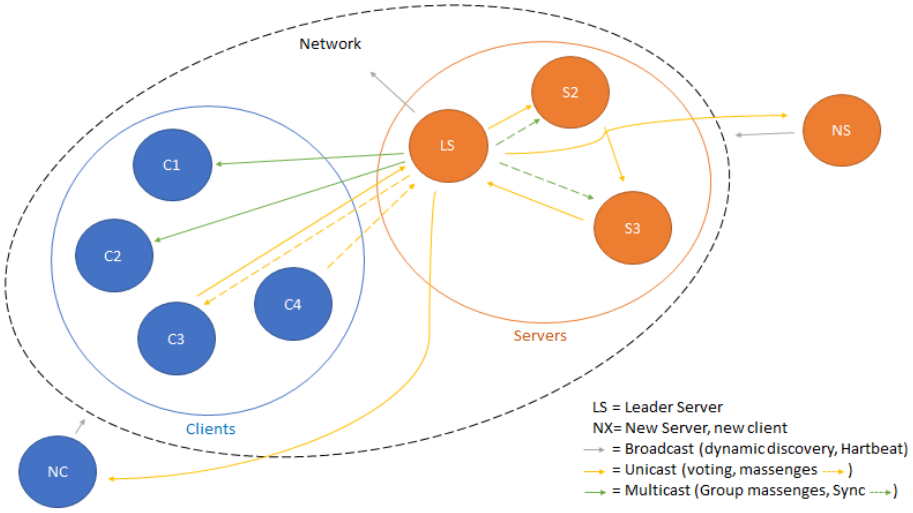

Fig. 1. Architecture design draft.

For the architecture design we decided to use a server-client network, for this reason we need to make a server instance available. All the important chat information includes the coordination code are distributed among the server and clients.

To build the server-client network, the server holds a list of all the UUIDs of the chat participants. In this way, the server knows currently active nodes and the clients can communicate with each other.

This not only provides communication capabilities, but also helps with crash fault tolerance. Because even if the leader crashes, all servers can replace the leader.
To meet the requirement of chat, we also must build a network ring. Every server searches his neighbor by sending his UUID to the next if his UUID is bigger than the previous one, if not he sends the UUID of his neighbor to the next server.
The ring is the basis for the voting algorithm, the voting algorithm send its voting messages to its neighbor.

New clients can join the network by sending out a broadcast message to all the servers and clients in the network on a specific port. The leader server always listens to the joining messages and is responsible for providing all the information a new client needs to join the chat.
Servers can join in the same way, if a server does not get a join message back it is automatically the leader server and waits for other join messages.

2 Implementation

2.1 Python Socket Module

The socket library is available for the Python programming language and is used to send messages and data in a network between endpoints. There are different network structures in which sockets can be used. For example, a small implementation can take place within a local network between two computers, but also a very large implementation such as functions within the internet.
Socket is a low-level networking interface. It is available for all modern operating systems, such as Unix, Windows and MacOS. The socket library can be used for different structures. For instance, client-server architectures or multicast applications can be created.
For our distributed systems project we will use this module and build a client-server architecture.

2.2 Implementation Overview

We wrote the whole project in Python 3 programming language. The whole project is embedded in GitHub for collaboration. The repository can be found under this url: *https://github.com/hackschnitzel09/Github_DS_Project*. Through this opportunity, each group member can contribute on their own project topic as they prefer, and we will have a version documentation. Therefore, any programming environment can be used.

As of today, the system is able to send messages from one client via the leader-server to another client via UDP sockets based on human readable names like *Simon*. To implement this a *users.json* and a *servers.json* file provides the needed information and are maintained by the leader server. The main code can be found in the utility.py which is included in the server and client for accessing the needed code. In the *server.py* and *client.py* are only variables found which the processes need to be specified before running.
The voting process can identify the neighbor server according to the ID which should be given when joining the network and send a voting message. In the following will be some explanations of the code by functions (Refer to the screenshots below):

- get_ip(): Reads the IP based on the system hostname
- get_brodcast_ip: Extracts the IP of the local machine and changes the last three digits to 255
- udp_listener(): Opens a socket for receiving UDP messages and calls msg_split() for further steps
- msg_split(): Divides the received message to extract information. The syntax of messages is which kind @ to whom @ the message @ from whom. The kind is important for the server to know what to do with it. If kind is message the leader will send the message to the defined receiver. Therefore he uses usr_ip() to get

the latest IP to the name and sends it via send_msg(). If the kind is voting the message will for now only be forwarded to the neighbor.

- neighbor(): Gives the server with the next higher ID back by checking in the list for the next higher ID. Afterwards it pings the server to check if he is alive. Then gives the IP back.
- send_msg() and send_brodcast(): Sending messages either to a fixed IP or the broadcastip
- usr_ip(), usr_name(), sever_ip() and server_name(): Are translating either names to ID or IP or the otherway around based on the information given in the .json files

```
#create udp socket
def create_socket():
    udp_socket= socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    return(udp_socket)

def get_broadcast_ip():
    ip = get_ip()
    broadcast_ip = ip[:-3] + "255"
    return(broadcast_ip)

#get my ip
def get_ip():
    hostname = socket.gethostname()
    myip = socket.gethostbyname(hostname + ".local")
    return(myip)

#listen for msg on udp
def udp_listener(leader):
    s = create_socket()
    myip = get_ip()
    # Set the socket to broadcast and enable reusing addresses
    s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # Bind socket to address and port
    s.bind((get_ip(), port))
    print("Listening to broadcast messages")
    while True:
        data, addr = s.recvfrom(1024)
        if data:
            print("Message:", data.decode())
            #check if Iam leader
            msg_split(data.decode(), leader)

```

Fig. 2. Code screenshot 1.

```python
#Split mgs into receiver and msg and forward it to receiver
def msg_split(rec_msg, leader):
    msg_obj = rec_msg.split("@")
    kind= msg_obj[0]
    to = msg_obj[1]
    msg = msg_obj[2]
    sender = msg_obj[3]
    #print("send to: " + to)
    print("msg: " + msg)
    print("from: " + sender)


    if kind == "voting":
        msg = "voting@" + str(neighbor()) + "@" + "@" + get_ip()
        send_msg(msg, server_ip(str(neighbor())))

    if leader == True:
        if kind == "msg":
            msg = "msg@" + to + "@" + msg + "@" + usr_name(sender)
            print(msg)
            send_msg(msg, to)


#find neighbor
def neighbor():
    myip = get_ip()

    with open("servers.json","r") as servers:
        server_list = json.load(servers)

    my_id = server_name(myip)
    print(my_id)
    print(len(server_list))

    if len(server_list) == int(my_id):
        neighbor = 1
    else:
        neighbor = int(my_id) + 1
    print("my neighbor is: ", neighbor)
```

Fig. 3. Code screenshot 2.

```python
#Send UDP msg
def send_msg(msg, rec_ip):
    s = create_socket()
    s.sendto(msg.encode(),(rec_ip, port))
    print("msg send to: " + rec_ip)

#send broadcast
def send_broadcast(msg):
    s = create_socket()
    s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    s.sendto(str.encode(msg), (get_broadcast_ip(), port))
    print("broadcast msg send")

#get ip of user by name
def usr_ip(name):
    with open("users.json","r") as users:
        user_list = json.load(users)
    usr_ip = user_list[name]
    print("User ip is: " + usr_ip)
    return(usr_ip)

#get name by ip
def usr_name(ip):
    with open("users.json","r") as users:
        user_list = json.load(users)
    names = list(user_list.keys())
    ips = list(user_list.values())
    return(names[ips.index(ip)])

#get server by ip
def server_name(ip):
    with open("servers.json","r") as servers:
        server_list = json.load(servers)
    ids = list(server_list.keys())
    ips = list(server_list.values())
    return(ids[ips.index(ip)])
```

Fig. 4. Code screenshot 3.

- The function utility.heartbeat() is used by the leader to send out the heartbeat. Therefore he acquires the server addresses from the list and sends out the heartbeat via the send_msgp(). The difference between send_msg() and send_msgp() is the specification of the port.
- The server.heartbeat_rec() is the counterparty to the utility.heartbeat(). It is run by servers which are not voted as leaders. It generates a socket to receive UDP messages bind to the heartbeat port. If the beat is received it sets the global heart variable to true.
- The function server.heartbeat_reset() is the actor of the heartbeat functionality. If the heart variable is set to false it will trigger a voting. If the variable is true, it will reset it to false and wait for the next interval to recheck.
- To handle multiple processes at the same time the threading method was imported and used to start the services.
- The *voting.py* was included into utility.voting() the functionality remains the same. It generates the voting message and sends it via neighbor() to the next available server.

Broadcast Handler

There is one socket for sending and one socket for listening. The listening socket is in a separate thread in a while loop, always listening for broadcasts. The broadcast IP is dynamically calculated, based on the users IP address and subnet mask. The broadcast port is chosen by the user.

UDP Unicast

There is one socket for sending and listening. A random available port is selected. The socket is in a separate thread in a while loop, always listening for UDP unicasts.

Dynamic Discovery of Hosts

The dynamic discovery is intended to enable hosts to join the chat without having specific knowledge of the other participants. The broadcast IP address should be calculated from the own IP address and the subnet mask (from the DHCP server or own generation). Therefore, the only condition for joining is that the host is in the same local subnet. This dynamic discovery of hosts allows new hosts to easily connect to the running chat application on the standard broadcast port.

However, we had serious difficulties with the concrete implementation of dynamic discovery. We first put our focus on the creation of the whole chat application to get a result. Consequently, dynamic discovery is currently not implemented as mentioned in chapter 1.3.1.

Crash Fault Tolerance
To keep the distributed system online even after a leader server is crashed, an algorithm needs to be implemented, that detects and handles such an event adequately.

Description
The implemented algorithm is composed of three parts:
- Detection of crashes
- Start voting
- Distribution of information

When a leader server crashes (no response or heartbeat) voting is automatically started, each server sends its UUID to its neighbor if its UUID is greater than the one from the previous server. If not, it sends the previous server's UUID. This goes on until a server get its own UUID back - this server is then the leader.
Messages that are used to survey a neighbor's server state are sent using UDP unicasts. In comparison to TCP unicast messages, this allows for a reduction of message-related overhead. However, to reduce the risk of missed response messages leading to a false judgement, the counter allows for a maximum of three consecutive "hbping"-messages to be unanswered before declaring the neighbor server as offline.

Voting
Why do we use LCR?
For our chat Application, we considered two voting algorithms. The LeLann-Chang-Roberts Algorithm (LCR) and the Bully algorithm.

The LeLann-Chang-Roberts algorithm only uses unicast messages for voting messages. The Bully algorithm on the other hand uses broadcast messages to send its election messages to all other processes with higher IDs.
We choose to use the LCR algorithm because the LCR only requires messages whereas the Bully algorithm needs broadcast messages. A lot of broadcast messages could lead to a network overload, which could result in a package loss, what we must to avoid.

How does LCR work?
LCR works by sending voting messages in the network. The voting message gets passed through the ring until a new leader gets elected. We achieve this by following the instructions below:

a. One server starts the election with its own identifier
b. Every server receives the message and checks whether the identifier is bigger, smaller, or same than the own identifier:
    - If the received identifier is bigger than the own identifier, the server just passes the message to its next neighbor
    - If the received identifier is smaller than the own identifier, the server exchanges the identifier in the message with its own identifier and sends to its neighbor

- If the received identifier is the same as the own identifier, the message passed the whole ring and the server got elected as the new leader.

How do we implement LCR?
We implement the LCR by using every server UUID as an identifier and sending it to its neighbor. We get the neighbor by sorting all server dictionary and then picking the server with the next lower index.

The functionality is implemented in the middleware functions:

- When incoming voting announcements, it subscribes to all incoming unicast messages and checks for the following conditions:
  - If the command is 'voting' it either declares itself as leader, sends the message to its neighbor, or replaces the message UUID with its own UUID and then sends it to its neighbor. (Based on, whether the incoming UUID is equal, bigger, or smaller than the servers own UUID). When a server declares himself as new leader, he sends a reliable unicast message to all servers, containing the 'leaderElected' command and its own UUID and sets the leaderUUID variable as its own UUID.
  - If the command is 'leaderElected' the server replaces its leaderUUID value with the incoming UUID of the message.

How do we prove that the voting is correct?
We proved that the voting is correct by performing a simple voting test. We implemented a user input that manually triggers the voting function so that we can test it atomically.
We tested the voting functionality with four processes running on two different devices. Furthermore, we tested the whole scenario two several times, including restarting all the processes. All three times the server with the highest UUID got elected as new leader, which proofs that the implementation of the voting algorithm works correct.
We also tested to trigger the voting function through the heartbeat function, by killing the current leader server. We tested this scenario, with the same setup, and can confirm that the voting algorithm worked correct, one new leader got elected and was accepted by all other participants.

2.3 Conclusion

Throughout the implementation of this project, we have gained a deep insight into the functioning and complexity of a distributed system. In particular, the methods for crash fault tolerance and voting are extremely relevant to the correct functioning.

The biggest challenge accruing is the ability to receive broadcast messages because the messages send to the broadcast IP are not transmitted further to the UDP listening sockets. There were several tries in different environments like hosting the machines in VirtualBox with several different network configurations as well as hosted machines on a NAS system. Therefore, it was decided to focus on the other tasks and lower the priority of dynamic discovery.

2.4 Outlook

The next steps in this project will be the implementation of the voting system. Now messages are forwarded to the neighbor, but he is also forwarding the message without interaction. The finding of the neighbor needs to be optimized for the case of no direct follower ID. Moreover, now only one port is in place the idea is to put voting messages for instance on another port.
Afterwards new servers and clients should be able to be recorded by the leader server in the .json files if no leader is available a voting will be triggered. Syncing this list to other servers would be the next step. In addition, a heartbeat between the leader and the other servers and maybe clients as well, needs to be implemented along with the voting trigger if a beat is not received.
For getting the system running, multithreading is needed to be implemented for clients to send and receive messages at the same time as well as servers for forwarding, listening heartbeat among other things.
Lastly the broadcast issue will be taken into elaboration.

References

1. Aiello M..: Lecture notes Distributed Systems (Summer term 2022)
2. https://docs.python.org/3/library/socket.html (Last checked, 22.06.2022)