Distributed Chat System - final report

Course: CSM13001 Distributed Systems, University of Helsinki
Members: Juha Väisänen, Pekka Prökkinen, Ville Hänninen, Heidi Holappa
Group number: 9

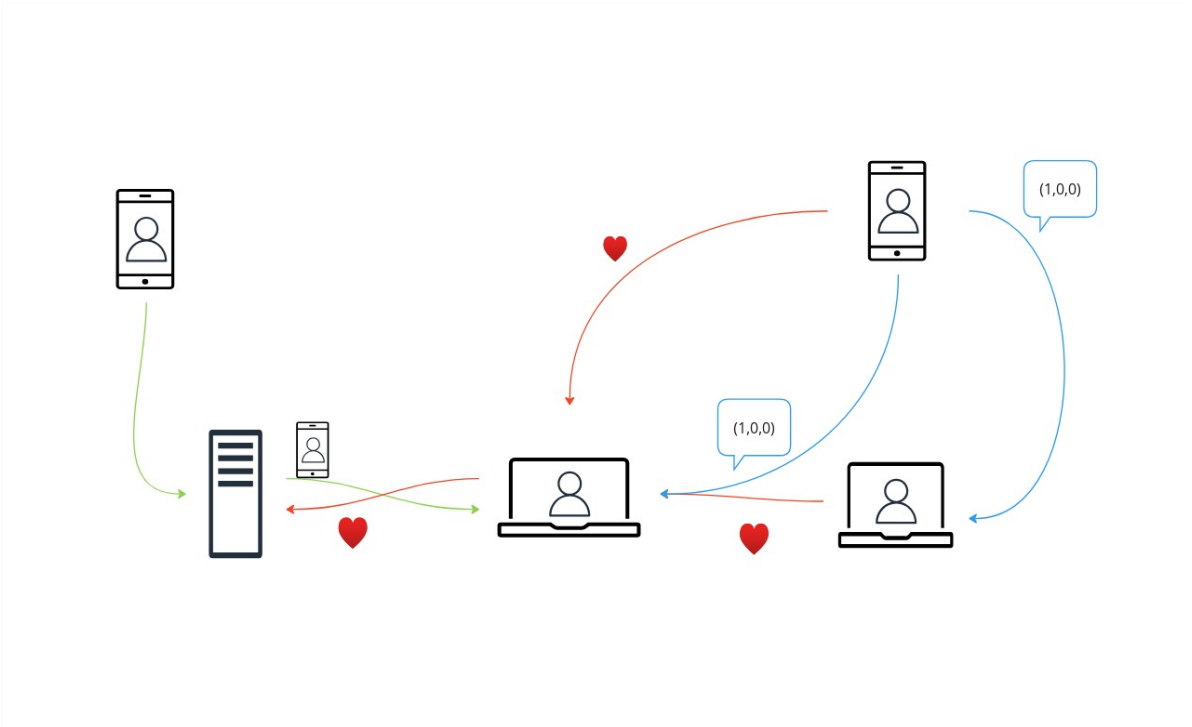


Figure 1: Image of the Distributed Chat System with a Node Director, a Coordinator, and Chat Nodes. A new Chat Node is contacting the Director to join the network and the Node Director is passing its information to the Coordinator. The Coordinator is sending a heartbeat to the Node Director. Chat Nodes are sending a heartbeat to the Coordinator. One Chat Node is sending a new chat message to other Chat Nodes.

Table of Contents

1 Introduction

The purpose of this project is to research distributed chat systems and to implement a working prototype. As a novel approach, this system does not contain persistent storage. Instead, chat discussions are transient, like phone calls, and remain active only until all nodes leave the communication. At that point, the discussion is gone and cannot be retrieved.

The justification for such an implementation is that in today's world, we all live with an ever-growing digital footprint. At the same time, many of us have a need for private communication. Most social media services maintain communication history, making it difficult to clear previous discussions. This project introduces an approach in which each discussion is transient by default, promoting a new type of real-time, text-based communication.

The prototype implements the following key functionalities: - a single chat room for group discussion - node discovery through a central director node - eventually consistent chat messaging - coordinator election with tailored implementation of Bully algorithm - coordinator tasks to onboard new nodes and to keep the list of active nodes up to date

This report first reviews the functionalities implemented in the prototype. After this, some key steps to needed to proceed from a prototype to an MVP (minimum viable product) are discussed. The report concludes with lessons learned, reflections on group participation, a description of how LLMs (large language models) were used during the project and a short reference list.

The project's remote repository (with source code and documentation) can be found at Github: https://github.com/distro2024/distributed-chat-system

2 Techical overview

The system under design provides a chat application for end users. When users launch the application to initiate a chat discussion, the client application becomes one of the **chat-nodes** (later nodes) in the distributed chat system. The system includes a **node director** responsible for connecting nodes to each other. In this proof-of-concept version all nodes participate in the same discussion.

Terminology:

- **node:** chat-node sends and receives messages from other chat-nodes. Chat-node can also serve as a coordinator for other chat-nodes (detailed later in the document)

- **node director:** node director connects new chat-nodes into the group discussion. Node director does not participate in chat discussion, but communicates with the coordinator node (detailed later in the document)

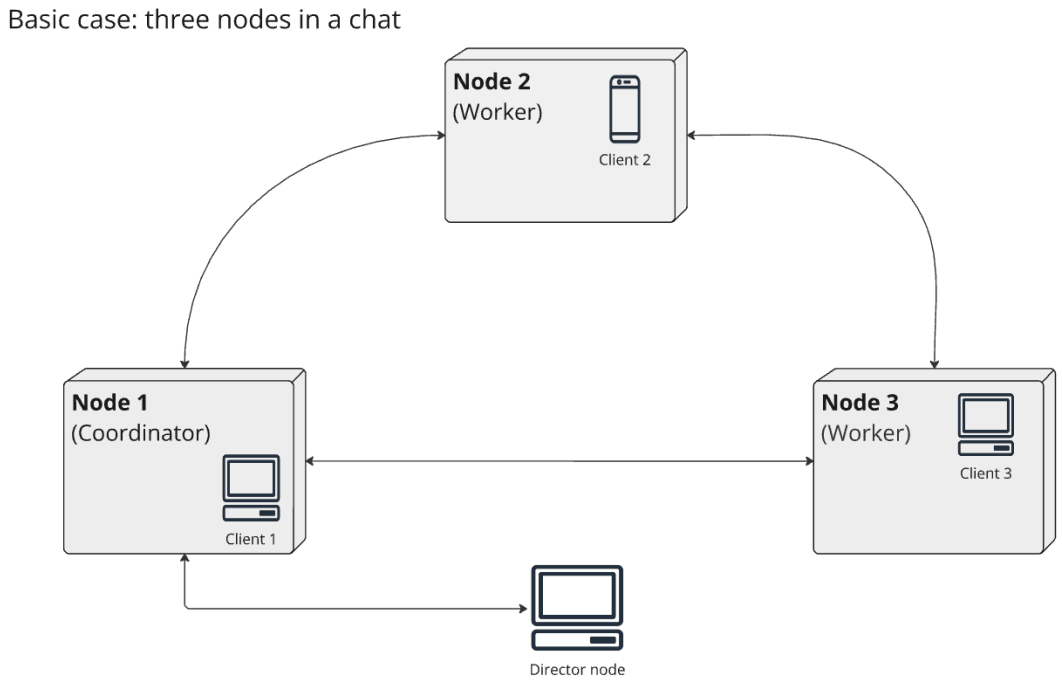


Figure 2: Basic case

2.1 System architecture

The architecture of this Distributed Chat System is centered around a network of nodes, each representing a client participating in the discussion. These nodes communicate directly with each other, forming a decentralized and fault-tolerant approach. One key component is the coordinator node, responsible for onboarding new nodes, maintaining an up-to-date list of active nodes. The system also incorporates a director node, which acts as a central point for node discovery, connecting new nodes with the current coordinator node. This architecture prioritizes providing users an experience of real-time interaction, employing websockets for node-to-node communication. The system is designed to be transient, meaning that chat discussions are not persistently stored and disappear when all nodes leave the communication.

The prototype provides distributed system transparency on some key areas (Steen & Tanenbaum 2024, 11):

- **location transparency:** the user is unaware of the physical locations of other nodes participating in the chat. The system hides this and provides an appearance of a single chat room

- **coordinator node management:** the selection, monitoring and eventual replacement of coordinator are all handled by the system. The users are unaware of which node (or if their node) serves currently as the coordinator.

- **message routing:** the system uses web sockets and HTTP to pass messages between nodes and ensures the eventual consistency of the chat messages. These details are hidden from the end user to enable them to simply focus on sending messages without being conserned about the actual implementation of messaging.

- **node failure and system recovery:** the system is designed to handle node failure gracefully. If required, an election to choose a new coordinator. Additionally the coordiantor removes unresponsive nodes from the discussion.

2.2 Nodes

Each node (aka chat-node) participating in the discussion is a chat-node. Nodes have an identical structure and should share an identical state of the discussion and of nodes in the network.

2.2.1 Node state

Each node stores information of all nodes in the network. For each node, the id of the node, the node's address and an instance of `socket.io-client` is stored into the object. Addition-

ally each node has a field lastHeartbeat, which is used by the Coordinator node (see section Coordinator node for more details):

```
[
    {
        "nodeId": "uuid",
        "nodeAddress": "URI",
        "socket": "instance of socker.io-client object",
        "lastHeartbeat": "timestamp"
    }
]
```

Storing `socket.io-client` to each node helps us streamline the communication, as now we can send a message to all nodes with:

```
nodes.foreach(nodes) => {
    // check that the node.id is not the id of the sender
    // carry out an operation
    node.address.emit('path', body)
}
```

Each node should also share an identical state of the chat-discussion. This functionality is detailed in section Sending and receiving messages

2.3 Coordinator node

The coordinator node is an elected role responsible for onboarding new nodes into the chat and maintaining an up-to-date record of the nodes participating in the discussion. During the onboarding process, the new chat node receives a list of its neighbours and the existing chat history. This list of neighbours is updated and sent to all chat nodes whenever a new node joins the chat. Each chat always has one coordinator node, and if the current coordinator fails, the remaining nodes initiate an election process to select a new one. Thus, any node is eligible to become the coordinator at any point during the chat. See coordinator election section for more spesific details of the election.

The coordinator node sends heartbeats to the director node, allowing the director to identify the current coordinator for each chat and to direct new nodes to the appropriate chat. Similarly, chat nodes send heartbeats to the coordinator node, enabling it to monitor which nodes are active and participating in the chat. If the coordinator does not receive a heartbeat from a node, it runs a regular task to identify and remove potential zombie nodes from the list.

The coordinator node has two endpoints:

POST `/onboard_node`: Handles the onboarding of a new node into the chat by providing the list of neighbours and the chat history to the joining node. The payload structures are:

```
{
    "nodeId": "uuid",
    "nodeAddress": "URI"
}
```

for a request and

```
{
    "neighbours": [
        {
            "nodeId": "uuid",
            "nodeAddress": "URI"
        }
    ],
    "discussion": [
        {
            "id": "uuid",
            "nodeId": "uuid",
            "nodeHost": "URI",
            "vectorClock": { ["nodeId"]: "int" },
            "timestamp": "timestamp",
            "message": "string"
        }
    ]
}
```

for a response.

POST `/heartbeat`: Handles incoming heartbeats from chat nodes. The request payload structure is:

```
{
    "nodeId": "uuid",
}
```

and the response is only a status code “200”.

2.4 Node director

The director node is a separate node (see Figure 2) whose main role is to manage node discovery. The director node has a known name or address, which is provided to chat nodes as a starting parameter. It offers two endpoints:

POST `/join_chat`: Handles requests to join the chat. Both the request and response payloads have the following structure:

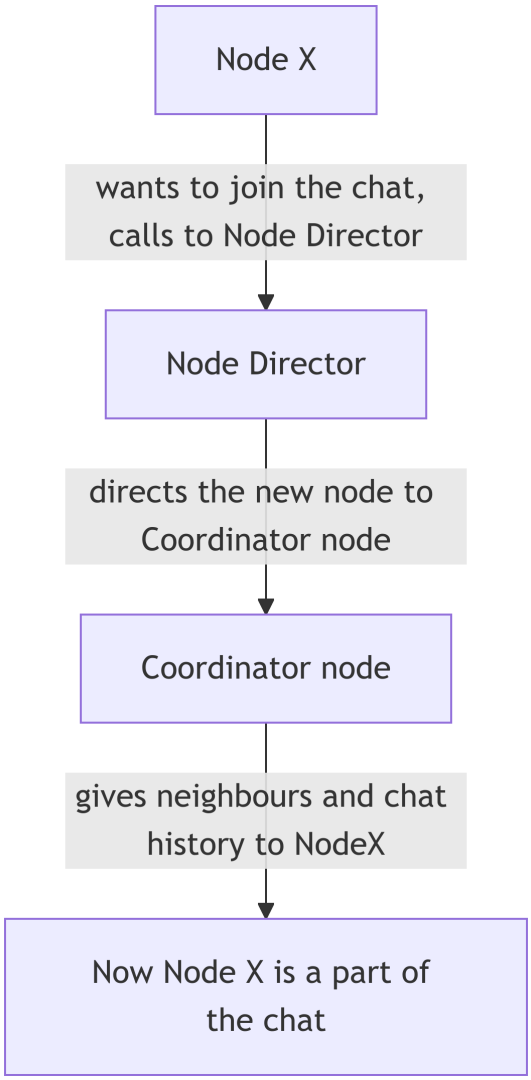
```
{
    "nodeId": "uuid",
    "nodeAddress": "URI"
}
```

In the request payload, the nodeId and nodeAddress represent the joining chat node's ID and address. In the response payload, these fields represent the current coordinator's ID and address. If the joining chat node is the first node in the chat, the response payload contains its own details.// MAYBE NEW PAGE FOR ALL LEVEL TWO (##) SECTIONS?

POST `/update_coordinator`: Updates the information about the current coordinator in the chat. Both the request and response payloads follow the same structure as above.

2.5 Joining group discussion

When a new node wants to join the chat, it connects first to the Node Director with a known name/address. The Node Director directs it to a coordinator node, which gives neighbour nodes and the existing chat history. Now the node is part of the chat.



2.6 Coordinator election

> This subsection contains references to the prototype module ./node/election.js. Each referenced method can be found in this module.

After the distributed system is initialized and Director is running, the first node to join the network will be made the Coordinator by the Director. After this, every Coordinator is decided with an election process. In the prototype the election is initiated when the Coordinator node becomes unresponsive. The prototype implements an election process based on the well established Bully-algorithm (Garcia-Molina 1982). Some inspiration has been taken from the Raft-algorithm as well, which was introduced later in the course.

In this implementation all other nodes send a heartbeat to the Coordinator node at a pseudo-randomized interval between $[T, 2T]$ seconds. When a node does not receive a reponse to two consecutive heartbeats, it initiates an election by sending an election message to all nodes with an id higher than its id. In the prototype ids are UUIDs and the comparison is an ascending String comparison (i.e. ($aaaa < aaab < aabb < ...$)). For implementation see method `initiateElection`.

After initiating the election, the node waits for a specified time (in the prototype three seconds) for responses. Then, if: - no responses arrive, it announces itself as a Coordinator to all the nodes in its node list (see method `determineVotingOutcome`). - atleast one response from a higher Id node arrived, it denounces its candidacy (module-scoped global variable `isCandidate` is set to false). For details see method `handleIncomingVote`.

Upon receiving an election request, each node initiates its own elections and sends a response `vote` (OK) to the election message (see method `sendElectionResponse`). The idea, as in Bully algorithm, is that the node with the highest id is the only one that should not receive any responses, given that the assumptions set by Garcia-Molina hold (1982). That is, for instance, that no messages are lost during election.

2.7 Sending and receiving messages

The nodes will send chat messages to all other nodes in the group discussion. The team will investigate a mechanism for transferring chat messages to each other efficiently. The idea is to simulate multicast functionality that works also when the nodes are in different networks. As a starting point, the team investigates using websockets or HTTP/2-protocol for inter-node communication. As communication between nodes is direct node-to-node discussion, no middleware is required in this proof-of-concept phase.

The team has outlined the necessary messages for the system currently under design. While the communication technology and protocol are still being evaluated, the final format of these messages has not yet been determined. For clarity and readability, the proposed message

content is presented in JSON format. Regardless of the chosen technology, these examples will illustrate the intended structure and content.

2.7.1 Chat messages

Any node (i.e. chat-node) can send chat-messages to other nodes: POST `/message`

```
{
    "id": "uuid",
    "nodeId": "uuid",
    "nodeHost": "URI",
    "vectorClock": { ["nodeId"]: "int" },
    "timestamp": "timestamp",
    "message": "string"
}
```

2.7.2 Communication structure

The project utilizes the Socket.IO library to implement communication in both client-to-node and node-to-node scenarios. Additionally, the socket.io-client library is used for inter-node communication. This approach ensures real-time interaction.

Client-to-node communication Clients (e.g., browsers) connect to the server via Socket.IO. When a client establishes a connection, the server listens for the connection event and receives a new socket connection.

Messages are sent by the client to the server using the established socket connection and the `client_message` event. A new client retrieves the chat history from the server using the `discussion` event.

Node-to-node communication Node-to-node communication is implemented in a separate Socket.IO namespace, /nodes. This separation isolates inter-node communication from client communication and helps minimize conflicts between client and node events.

Nodes can send various messages to each other, such as `node_message`, `update_nodes`, `vote`, `update-coordinator`, and `election`.

2.7.3 Election process

Messages related to the election process with the implementation of Bully-algorithm. The messages are represented here in JSON format for clarity. The actual prototype implementation can be reviewed in module `./node/sockets.js`:

Any node can initiate election process by emitting a request to nodes with higher id to route `/election`

```
{
    "nodeId": "uuid"
}
```

Nodes respond to an election request by emitting a request to the sender of election request to route `/vote`

```
{
    "nodeId": "uuid"
}
```

The node with the highest id will not receive any vote requests, and determines it to be the new Coordinator. It then notifies other nodes that they are the new coordinator by emitting their id to route `/update-coordinator`

```
{
    "nodeId": "uuid"
}
```

2.8 Eventual Consistency in discussion

- **Direct Communication:** Nodes communicate directly with each other, reducing reliance on a single point of failure.
- **Vector Clocks, Timestamps, and Priority Queue:**
  - Each node tracks events using a vector clock, with counters representing the state of each node.
  - The local counter is **incremented only when the node sends a message**, accurately reflecting local events.
  - Upon receiving a message, the local vector clock is updated by merging with the incoming clock, taking the maximum value for each node.
  - **Timestamps are used to break ties when vector clocks are identical.**
  - **A priority queue (implemented through the `discussion` array) stores messages, ordered by their vector clocks and timestamps.** This queue uses the `compareMessages` function to determine the order of messages during insertion.

    - **Inserting a new message into this queue has a time complexity of O(n) in the worst case**, where n is the number of messages in the queue. This is because the `handleNewMessage` function (which utilizes the `compareMessages` function) potentially needs to iterate through the entire array to find the correct position for the new message.
    - **Retrieving messages from the queue (when sending the history to clients or new nodes) has a time complexity of O(1)** since the messages are already stored in order.

- **Catch-up Mechanism:** New nodes receive the full chat history (`discussion` array), allowing them to become consistent with the rest of the network.
- **Points to consider for future development:**
    - **Vector Clock size**: Vector clocks grow in size as the number of nodes increases. This can lead to increased storage and communication overhead. Consider strategies to manage clock size, such as pruning old entries.
    - **Catch-up Optimization**: Sending the entire discussion array to new nodes can be inefficient if the chat history is very large. Consider optimizing the catch-up mechanism, perhaps by sending only a recent portion of the history.

2.9 Security

Our prototype prioritizes secure communication. When connecting nodes, use **wss://** addresses for WebSocket connections and **https://** addresses for standard HTTP requests ensuring all data exchanged is encrypted using TLS. However, beyond this basic transport encryption, our initial prototype does not include additional security measures. In the MVP we will address this by incorporating the following security considerations:

- **Input Sanization :**
    - Input sanitization
        - All user inputs should be sanitized by escaping or encoding characters that could potentially cause harm to the system. For example, by properly escaping the following characters, the risk of Cross-Site Scripting (XSS) attacks can be mitigated: `<, >, &, "`.
    - Validation
        - Enforce strict rules for data types, formats, and allowed values to prevent invalid data from causing issues or being exploited.
- **User authentication**
    - Despite the absence of user accounts in the current prototype, we recognize the critical importance of addressing user authentication early in the development process. We will address the following key aspects of user authentication:

    - **Strong passwords**
        - Enforce password complexity and potentially use password hashing.
    - **Multi-factor authentication (MFA)**
        - We will evaluate the feasibility of implementing MFA, such as OTP or authenticator apps, to provide an extra layer of security.
    - **Session management**
        - Secure session management practices will be implemented, including the use of secure cookies and session timeouts, to protect user accounts.

- **Integrity of messages**
    - To ensure message integrity, the MVP will incorporate measures to protect against tampering during transit and storage. The specific techniques for ensuring integrity are still under investigation and will be finalized during further development. We are currently evaluating several approaches to ensure message integrity, including:
        - **Hashing algorithms (e.g., SHA-256)**
            - Generate a unique hash of each message. This hash is sent along with the message. Upon receipt, the recipient independently calculates the hash of the message and compares it to the received hash. If they match, the message integrity is confirmed.
        - **Digital signatures (public-key cryptography)**
            - The sender signs the message with their private key, and the recipient uses the sender's public key to verify the signature. This method ensures integrity, authentication, and non-repudiation. Examples include RSA and ECDSA.
        - **Message Authentication Codes (MACs)**
            - Use a shared secret key to generate a MAC, such as HMAC, for each message. The MAC is transmitted with the message, and the recipient, possessing the same secret key, can verify it, ensuring both integrity and authentication.

- **Security Testing**
    - To ensure the effectiveness of our security measures, we will conduct regular security testing throughout the development process. This will include:
        - **Penetration testing:** Simulating attacks to identify vulnerabilities.
        - **Vulnerability scanning:** Using automated tools to detect known weaknesses.
        - **Code reviews:** Manually inspecting code for security flaws
            - Security testing will be integrated into our CI/CD pipeline to ensure that tests are run automatically with every code change.

- **Additional note on security measures**

– When choosing the appropriate dependency for a certain use case, the necessary precautions should always be taken and it should be ensured that chosen dependencies are from trusted sources.

2.10 Language selection

The chat nodes and the node director have been implemented using Node.js. Node.js has a strong reputation in handling asynchronous calls, which the team considers to be a critical functionality for the system under design.

2.11 No persistent storage

As the motivation is to built a system for group discussions without any persistent memory, no database is necessary for this prototype proof-of-concept (PoC). Once all client applications shut down, the discussion is permanently lost.

2.12 Container technology

All nodes may also reside within containers, which may or may not be located on the same physical machine. This containerized approach ensures flexibility, scalability, and ease of deployment across different environments. The protype provides a demonstration of Docker compose implementation. The system can also be used without containers. For instructions, review the README.md.

3 From prototype to MVP

3.1 Potential applications of the project

This project provides a foundation for building diverse applications that demand real-time communication capabilities. Its potential extends to social media platforms, collaborative tools, and specialized services like emergency response and IoT systems—any application where speed, scale, and reliability in messaging are critical can leverage this technology.

This simple prototype demonstrates some key functionalities in the scope of one shared group discussion. In an MVP the system should provide multiple discussions and provide the end-users the control to select, which discussion they participate in (or to create a new discussion room).

3.2 Scaling

The system can be easily scaled by simply adding new nodes. Client communication and inter-node messaging are separated, which helps distribute the load efficiently.

3.3 Transparency considerations

In an MVP Chat system closer attention should be placed on system transparency. For instance, it would be beneficial to provide the end-users the option to be aware of when a node joins or exits the discussion. Currently this is hidden from the user. Additionally, it should be considered whether the user should be notified of missing chat messages.

The system guarantees eventually consistent ordered chat message history and is aware when messages are missing. In the prototype this is hidden from the user. As the system grows from a prototype to an MVP, the transparency should be reviewed in a large perspective. As new functionalities are introduced, new considerations for transparency will also arise.

3.4 Optimization of data structures

In the prototype each node stores information on nodes in the network in an array. It should be considered, whether some other data structure would be more efficient. For instance, after Zombie hunt the Coordinator node iterates through the whole array to find and remove unresponsive nodes. It should be considered whether other data structures, such as map, could provide better time and space complexity results.

3.5 Election process

The election process in the prototype is based on the well-established Bully algorithm and took additionally loosely some inspiration from the Raft-algorithm (heartbeats, randomized time-intervals for sending heartbeats). In an MVP the implementation should be refined and fault-tolerance should be improved. Also naming conventions should be reconsidered.

In the current implementation nodes response to an election request with a `vote`. This naming may lead to confusion as it can be interpreted to mean that the sender of `vote` is giving the other node their vote, when actually they are "giving" themselves the vote and the purpose of the response is to let the other node to know, that a higher id node is ready to become the Coordinator. To improve readability and maintainability naming conventions should be as clear as possible and this should be clarified.

3.6 Improving fault tolerance

3.6.1 Messaging

In the prototype messaging is kept simple to purely demonstrate some key features of a Distributed Chat System. For instance, heartbeats are sent as HTTP-requests without fine-grained response handling. In an MVP the exception handling should have a higher maturity level and if HTTP requests and responses are used for some communication, the status codes of the responses should be also verified.

3.6.2 Election process

The current implementation is based on the assumptions presented by Garcia-Molina (1982). For instance, it is assumed that no messages are lost. In reality, messages can be lost and this could lead into a situation, in which multiple nodes declare themselves the new Coordinator. In the current implementation this could lead to multiple nodes continuously declaring themselves the new Coordinator to other nodes and to the node Director. In an MVP election process should be resilient and handle such scenarios.

The optimal solution should be carefully researched, but as an initial thought the Director node could be used to detect a situation in which two or more nodes keep constantly updating states (resembling a livelock) and to intervene in some way. It is worth noting that in such scenario the Coordinators may have different lists of nodes and this should also be fixed.

> Side-note: it is possible to demonstrate this issue with the prototype by (a) disconnecting the Coordinator node, (b) waiting just long enough for the first election to start and (c) quickly bringing up the current Coordinator again. In this case a new Coordinator is elected but the current Coordinator get the old state from the Director and sets itself as the Coordinator (leading to two Coordinators)

3.6.3 Director node

In the current implementation, there is only one Director node. Therefore, if it fails, new nodes cannot join the chat. To improve fault tolerance, the Director could be modified to store information about chats and their coordinators in a more persistent and centralized manner. This would allow for a pool of Director nodes, with an election process among them to select one responsible for listening to the known name or address for connections.

3.7 Benchmarking

The report outlines, that simple messaging was used to demonstrate key features of a Distributed Chat System. For example, heartbeats were sent as HTTP requests without fine-grained response handling. Neither the communication with web sockets or with HTTP was benchmarked. For a Minimum Viable Product (MVP), a more mature level of communication exception handling and verificatoin is necessary. To achieve this, benchmarking is essential for an MVP to evaluate performance and optimize key parameters.

The system has has been demonstrated to function with four nodes. For an MVP, the scalability should also be benchmarked to find limitations for the maximum number of nodes the system can support in a single discussion. One possible bottleneck will be the leader election, which will create more overhead as the network grows in size.

The heartbeat frequencies and coordinator task frequencies, such as the frequency of removing zombie nodes, should also be benchmarked and optimized. Benchmarking will play a crucial role in determining the optimal frequencies for these tasks.

By systematically testing different frequency values and measuring their impact on system performance, developers can identify the settings that strike the best balance between responsiveness, resource consumption, and fault tolerance. Without benchmarking, arbitrarily set values might lead to either excessive overhead from frequent tasks or delayed responses due to infrequent checks, or failures due to unsufficient exception handling and verification.

3.8 Performance

The performance is significantly affected by the number of nodes. While the system can handle a small number of nodes adequately, increasing the number of nodes substantially leads to added network and processing load due to broadcast-type message distribution, node-to-node coordination, sharing of historical data, and the election process. Latency grows in line with increasing network latency and the degree of network distribution.

This problem could be addressed, for example, by forming clusters or using intermediary servers, ensuring that not all nodes communicate directly with every other node.

4 Lessons learned

Working on this small prototype gave the participants an opportunity to study hands-on some course concepts. The team had at least one meeting each week, during which we discussed the themes related both to the group project and to the course in general. During the process the team got a glimpse into the challenges that manifest when designing a Distributed System.

The scope of this project was quite limited and a lot of constraints had to be placed on which functionalities to implement. As a team we are in consensus in considering our prototype to be a good and coherent demonstration of some key features that would exists in some form in an actual distributed chat system. In it's constrained scale the system is functional and provides selected core features required from a Distributed Chat System.

5 Group participation

```
FINAL REPORT INSTRUCTION
7. Notes about the group members and their participation, work task division, etc. Here you a
```

- How did everyone participate
- How did we work
  - meetings
  - how we kept in touch
  - pair coding etc.

The team maintains active communication through a group discussion on Telegram. Each weekend, they hold an online call to plan the upcoming week and discuss any current issues. Work items are coordinated using a Kanban-style project board on GitHub, where tasks are tracked as issues. This approach helps the team better estimate workloads, coordinate active tasks, and plan the project's timeline effectively.

6 Use of LLMs

University of Helsinki Large Language Model Curre, Microsoft Co-pilot and Chat-GPT have been used to help improve the grammar of the documentation and to polish the text. Separate LLMs have been prompted to provide feedback on text and to suggest improvements. The response has then been evaluated and with consideration applied to improve the text.

7 References

Garcia-Molina H. 1982. Elections in a Distributed Computing System. IEEE Transactions on Computers, 31(1):48–59, Jan. 1982.

Steen, M. van & Tanenbaum, A. S. 2024. Distributed systems / Maarten van Steen, Andrew S. Tanenbaum. Fourth edition, Version 4.02 (2022). Published at unknown.