IT UNIVERSITY OF CPH

Mandatory Activity 3: Chit Chat

Distributed Systems, BSc (Autumn 2025)

BSDISYS1KU

Group:

Anders Grangaard Jensen	Agje@itu.dk
Mathias Vestergaard Djurhuus	mavd@itu.dk
Theodor Monberg	tmon@itu.dk

IT University of
Copenhagen
Fall 2025

Table of contents

0. Abstract

The purpose of this project was to design and implement a distributed chat service called ChitChat using the gRPC framework. The goal was to satisfy a set of functional and architectural requirements (S1–S7), including support for multiple concurrent clients, ordered message delivery, and a reliable broadcast mechanism for all chat events. The system implements Lamport logical clocks to ensure the consistent ordering of events across distributed processes, demonstrating core principles of distributed systems, including synchronization, concurrency, and event ordering.

1. Streaming Model

ChitChat uses server-side streaming as its primary communication model. Each client connects to the server through a persistent streaming RPC call (Subscribe), which allows the server to continuously push messages to all connected participants. This approach ensures real-time message delivery without requiring clients to repeatedly poll the server.

Alternative models such as client-side or bidirectional streaming were considered but ultimately rejected. Client-side streaming would not fit the system's communication pattern, as clients only need to send individual messages occasionally. Bidirectional streaming would add unnecessary complexity, since this system requires a single, authoritative message hub that fans out broadcasts to many listeners. The server-side streaming model, therefore, represents the most natural fit for a chat service that emphasizes broadcast communication and event ordering.

2. System Architecture

The ChitChat system follows a centralized client–server architecture. A single server acts as the central coordinator, managing all active participants, maintaining the global Lamport clock, and broadcasting messages to clients. Each client runs as an independent process that communicates exclusively with the server through gRPC.

When a client starts, it sends a Join request to the server, receives an acknowledgment containing its assigned ID and initial logical time, and then opens a persistent Subscribe stream to receive future broadcasts. Whenever a user types a message, the client invokes the unary Publish RPC to send it to the server. The server increments its logical clock, updates the message timestamp, and broadcasts it to all connected participants through their active streams. Finally, when a user leaves, the client sends a Leave request, prompting the server to remove the participant and broadcast a "left" message to all remaining clients.

This architecture guarantees that all message is coordinated through the server, while each client maintains a local logical clock that synchronizes with the server's global state through Lamport's clock rules. It is therefore a classic centralized topology rather than a peer-to-peer network.

3. RPC Methods and Message Types

The system defines four core RPC methods in its gRPC API.

Join (JoinRequest) → JoinAck

A unary RPC where a client registers with the server by sending a *`JoinRequest`*.

The server assigns a participant ID and returns a *`JoinAck`* containing both the name, ID, and current logical time.

Subscribe (SubscribeRequest) → stream Broadcast

A server-side streaming RPC. Once invoked, this call keeps the connection open, allowing the server to send all future broadcast messages (*`JOIN|MESSAGE|LEAVE)`* to the client in real time.

Publish (PublishRequest) → PublishAck

A unary RPC used by clients to send new chat messages. The server timestamps and broadcasts the message to all active participants.

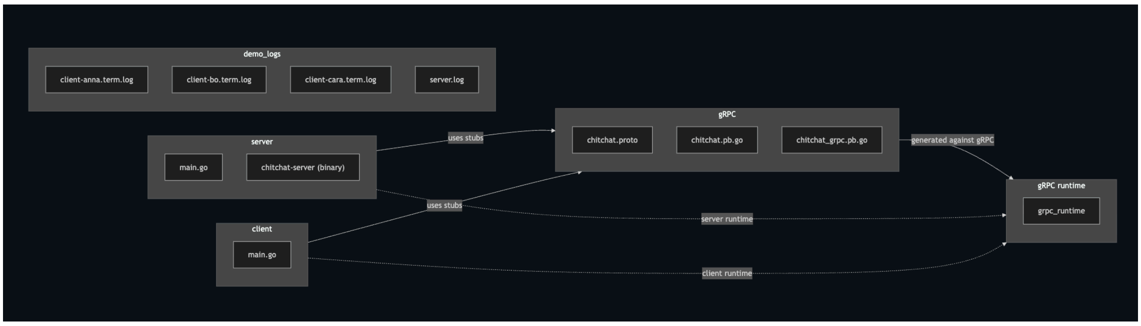
Leave (LeaveRequest) → LeaveAck

A unary RPC that informs the server when a client disconnects, triggering a broadcast to all remaining clients.

All communication relies on the message types defined in chitchat.proto, including `JoinRequest`, `JoinAck`, `SubscribeRequest`, `Broadcast`, `PublishRequest`, `PublishAck`, and `LeaveRequest`. Each message includes the participant's ID, name, content, and a logical timestamp. The Broadcast type includes an enumerated field describing the kind of event *(JOIN|MESSAGE|LEAVE)*.

4. Repository Structure (Using Mermaid)

Link to respository: https://github.com/AGJ2000/LeGroup/tree/main



5. Lamport Timestamp Implementation

ChitChat implements Lamport logical clocks to maintain a consistent ordering of messages across distributed clients. Each process. Both server and clients maintain its own local logical clock variable.

The clocks start at zero. Before sending a message, the process increments it's clock by one. The server receives a message from a client, it then compares its own clock with the timestamp included in the message and updates it using the Lamport's rule:

$$L\_server \;=\; max(L\_server, L\_client) + \;1$$

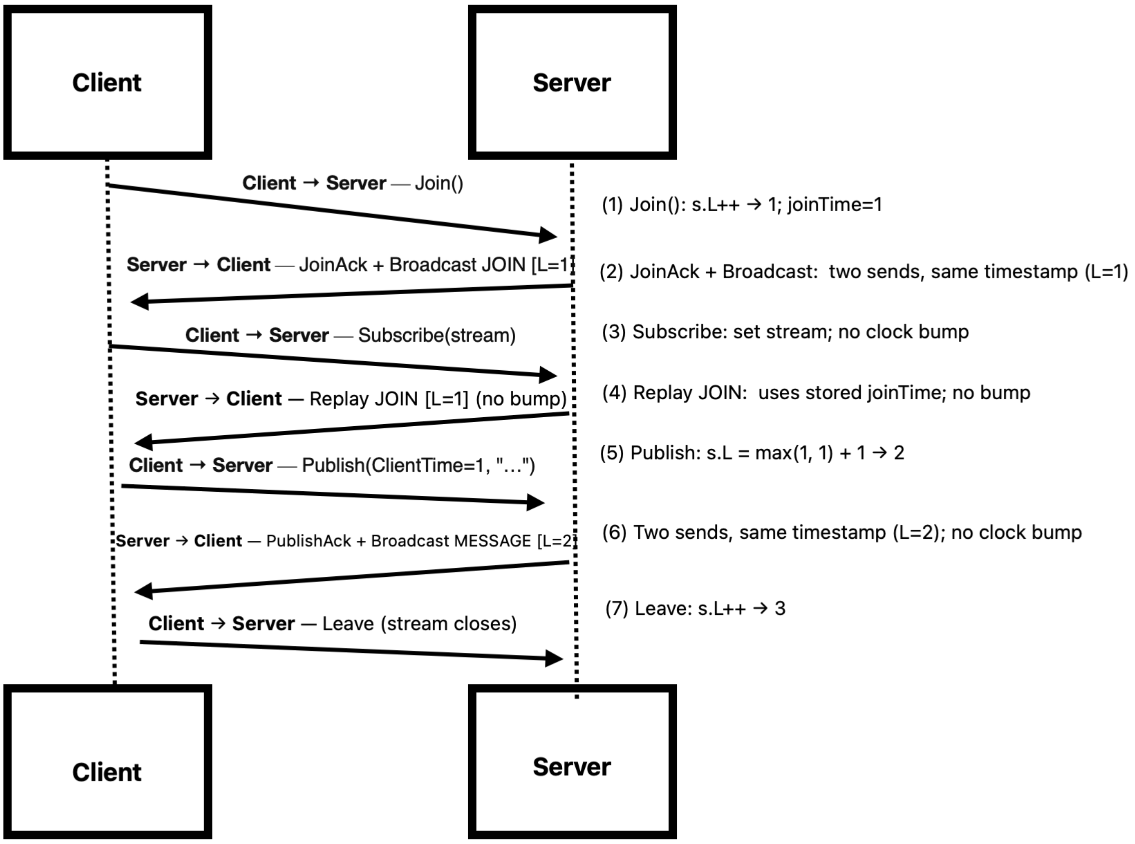
The resulting value is then included in the broadcast that goes out to all active clients. Upon receiving a broadcast message, each client updates it's local clock according to same rule.

$$L\_server \; = max(\, L\_client \,, msg.LogicalTime) + 1$$

Approaching it like this ensures that all participants agree on a global order of events, even though there is no shared physical clock. The Lamport clock does not measure real time but defines a causal ordering between messages such that if one event 'happened before' another, it will always have a smaller logical timestamp.

6. Sequence of RPC Calls with Lamport Timestamps (Made in Freeform)

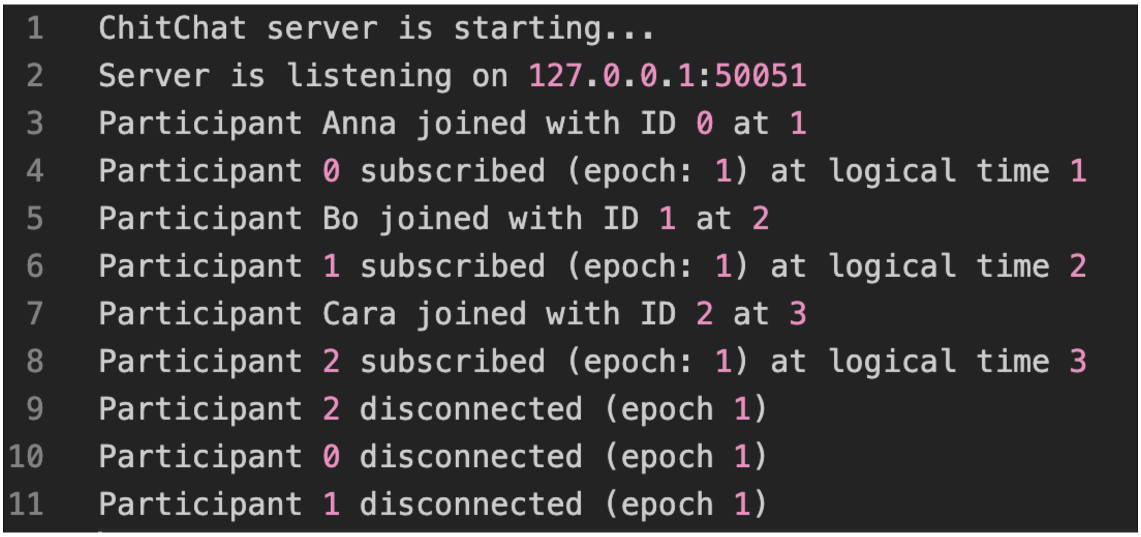
The diagram illustrates the communication flow between a client and a server using gRPC.


Client
Server
Client → Server — Join()
(1) Join(): s.L++ → 1; joinTime=1
Server → Client — JoinAck + Broadcast JOIN [L=1]
(2) JoinAck + Broadcast: two sends, same timestamp (L=1)
Client → Server — Subscribe(stream)
(3) Subscribe: set stream; no clock bump
Server → Client — Replay JOIN [L=1] (no bump)
(4) Replay JOIN: uses stored joinTime; no bump
Client → Server — Publish(ClientTime=1, "…")
(5) Publish: s.L = max(1, 1) + 1 → 2
Server → Client — PublishAck + Broadcast MESSAGE [L=2]
(6) Two sends, same timestamp (L=2); no clock bump
(7) Leave: s.L++ → 3
Client → Server — Leave (stream closes)
Client
Server

7. Appendix

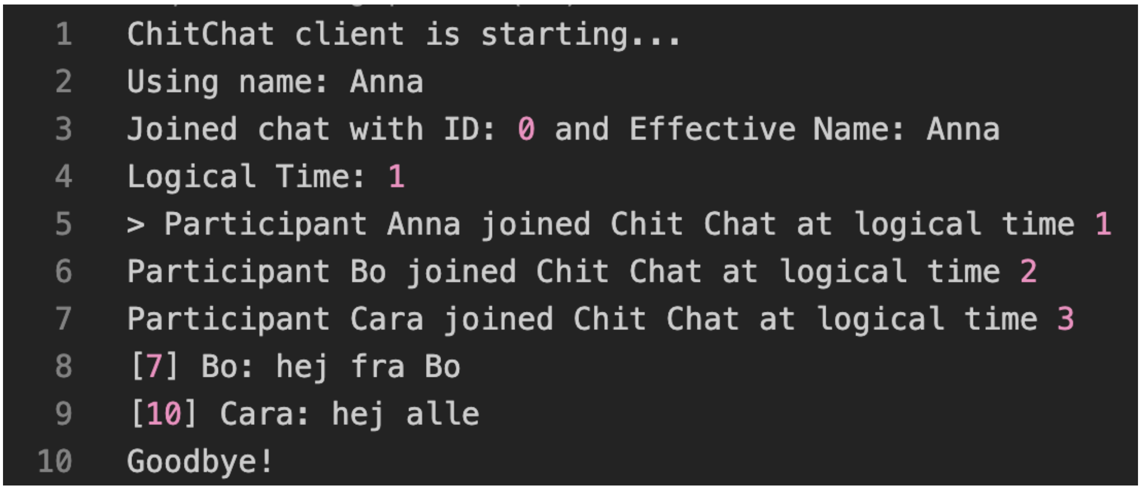
The accompanying repository includes:

- client-anna.log, client-bo.log, client-cara.log, server.log
- logs/0.log, logs/1.log, logs/2.log
- README.md with build and run instructions
- gRPC/chitchat.proto

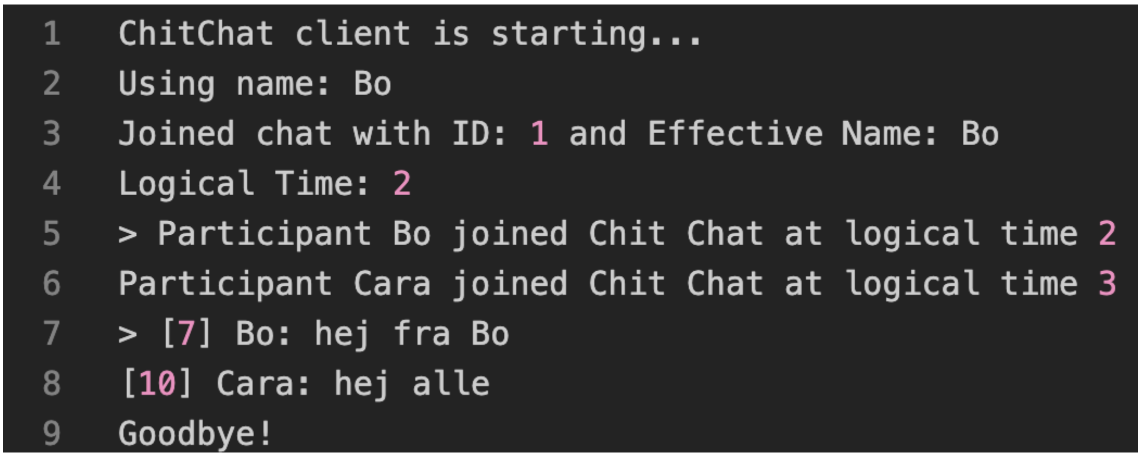


```
ChitChat server is starting...
Server is listening on 127.0.0.1:50051
Participant Anna joined with ID 0 at 1
Participant 0 subscribed (epoch: 1) at logical time 1
Participant Bo joined with ID 1 at 2
Participant 1 subscribed (epoch: 1) at logical time 2
Participant Cara joined with ID 2 at 3
Participant 2 subscribed (epoch: 1) at logical time 3
Participant 2 disconnected (epoch 1)
Participant 0 disconnected (epoch 1)
Participant 1 disconnected (epoch 1)
```

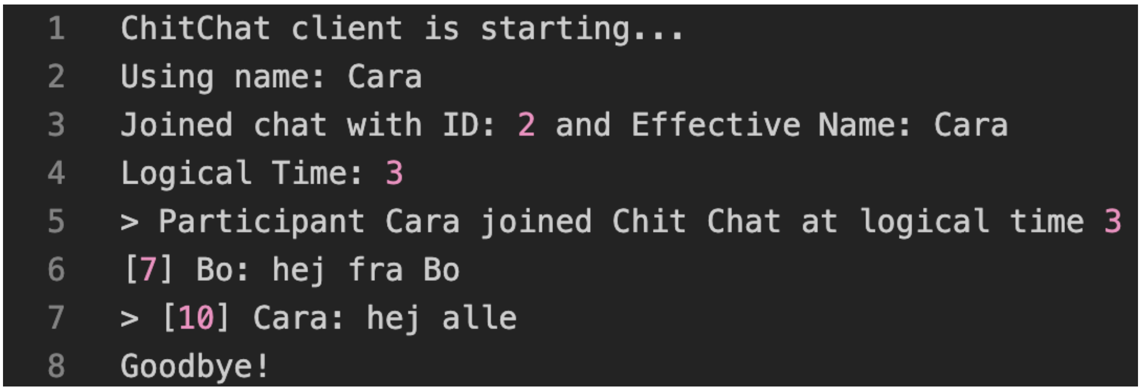
The server log above demonstrates that all participants can join, publish messages, and leave the system dynamically. Each client log confirms that broadcasts are correctly received and displayed with Lamport timestamps.



```
ChitChat client is starting...
Using name: Anna
Joined chat with ID: 0 and Effective Name: Anna
Logical Time: 1
> Participant Anna joined Chit Chat at logical time 1
Participant Bo joined Chit Chat at logical time 2
Participant Cara joined Chit Chat at logical time 3
[7] Bo: hej fra Bo
[10] Cara: hej alle
Goodbye!
```



```
ChitChat client is starting...
Using name: Bo
Joined chat with ID: 1 and Effective Name: Bo
Logical Time: 2
> Participant Bo joined Chit Chat at logical time 2
Participant Cara joined Chit Chat at logical time 3
> [7] Bo: hej fra Bo
[10] Cara: hej alle
Goodbye!
```



```
ChitChat client is starting...
Using name: Cara
Joined chat with ID: 2 and Effective Name: Cara
Logical Time: 3
> Participant Cara joined Chit Chat at logical time 3
[7] Bo: hej fra Bo
> [10] Cara: hej alle
Goodbye!
```

Every client maintains a local log file (logs/0.log, logs/1.log, logs/2.log), which contains the same sequence of JOIN, MESSAGE, and LEAVE events with their corresponding logical times. This demonstrates that all clients not only display broadcasts but also persist them locally as required by specification S7.

8. References

- Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. Link: https://lamport.azurewebsites.net/pubs/time-clocks.pdf
- The Go Programming Language. https://go.dev
- RPC Documentation. https://grpc.io/docs/