POLITECNICO
MILANO 1863

Causally-Ordered Message Delivery in a Highly-Available Distributed Group Chat

Distributed Systems A.A. 2023-2024

Matteo Bevacqua

July 9, 2024

Underlying Assumptions

Links

Unreliable links, messages are not guaranteed to reach the destination nor is multicast an atomic operation $\rightarrow$ ACKs [Unicast].

Clients

Clients are reliable (no crash nor Byzantine failures) but network partitions may happen or they might just leave **without notifying their peers**.

Timing

The room creation process is a timed procedure (around 60s in total), however, clients are not assumed to be synchronized in any manner. The only assumptions related to timing are $RTT << 1s, \rho << 1s$

Network Partitions

Edge cases due to partitions and causal delivery

Client c_i suffers a network partition while all the other $n - 1$ clients observe a given message m_j from client c_j.
Clearly all subsequent messages from all the other clients are causally dependent on m_j that c_i *has not observed*.
If c_j then leaves, it won't retransmit message m_j, as such c_i will be stuck forever, not able to deliver any messages from its inbound queue.

Solution

Solution $\rightarrow$ Request retransmission of past messages if after T_0 we aren't able to deliver any message.

Additional Issues

Expired Timeouts

A client comes back online after all the timeouts from his peers have expired, thus won't receive anything until they send a **new** message.
Solution $\rightarrow$ Ask to retransmit messages that come **after the last message acked by everyone** (if present and if we're able to detect packet loss).

Transient Short-Lived Network failure

Client c_i sends messages while c_j suffers transient **packet loss**, c_i then goes offline. **Unless new messages are sent by any of the other clients** (and received by c_j) c_j won't know that new messages have been sent.
Solution $\rightarrow$ Read-Repair, when c_k receives m_j from c_j and detects old timestamps it retransmits past messages.
$(old(m_j) = 1 \iff V_{tmp}[i] = min(V_j[i], V_k[i]) \wedge V_{tmp} \leq V_k \wedge V_{tmp} \neq V_k)$

Partition Issues

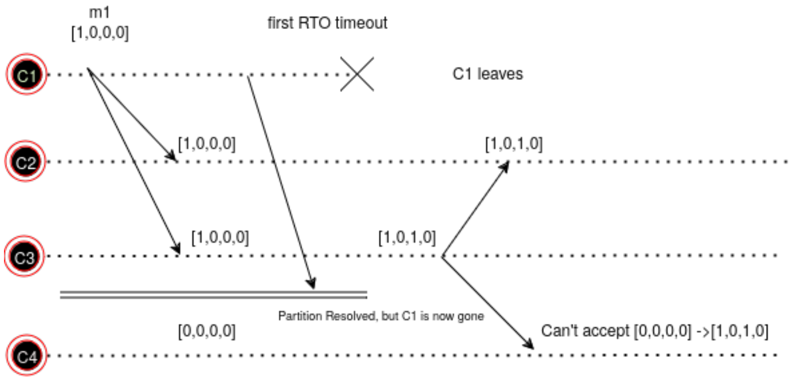

Figure: Edge case due to partitions

System Overlay

High Level Overview

Two main components

- Client
- QueueManager

Client

Handles user input, room management and chat browsing

QueueManager

Handles network communication, manages the sockets (1 per chat room, each room has its own multicast address)

Client Behavior

- **Peer discovery** each client sends an *HELLO* message on a predefined multicast channel (224.0.0.0), each online client will reply with a *WELCOME*
- **Heartbeats** Clients are assumed offline if after *5s* they haven't sent an heartbeat message over the common channel
- **Room Creation** Whoever wants to create a room will send a *ROOM_CREATE* command over the default channel, all the interested parties will reply with an *ROOM_JOIN_(N)ACK*. After 60s from sending the request the room will be finalized on the new multicast channel specified in the request, with whoever replied with an ack set as a participant.

Client-Queue Interaction

Thread-Based Asynchronous Model

Both the Client and Manager object are two separate threads that can be schematized as a producer-consumer pair.
The client adds outgoing messages to the queue of the room he wants to talk to, that will be in turn picked up and sent through the socket by the Manager.
The manager feeds back to the client generic *Events* whenever they need attention (e.g. a peer asked to participate in a room)
The interaction is non-blocking.
All shared data structures (residing in the *ChatRoom* object) are accessed through *synchronized* methods to avoid race conditions.

Message Ordering and Availability

Causal Delivery

Messages within each room (apart from the default common multicast one) are causally ordered, i.e. in order to accept a message at client k from j the following must hold:

$$V_k[j] = ts(m)[j] - 1$$

$$\forall i \neq j \quad V_k[i] \geq ts(m)[i]$$

The messages in each room (from each peer's perspective) are stored in a *linked list* based on their observed order.
Upon delivery of a message the recipient's vector clock is *merged* i.e.

$$V_k[i] = max(V_k[i], ts(m)[i])$$

Message Ordering and Availability

High Availability

Clients can go offline at any time. The queue thread detects network loss as soon as the socket throws an exception. Messages are queued (even without network problems) and are re-transmitted every 3 seconds until acked by every member of the room (or they've been retransmitted n_{max} times). Once a client gets back online it *might* receive missed messages.

Causality and Offline Queueing

Both outgoing and incoming messages are queued at the client. Messages stay in the inbound queue until they can be delivered according to their causal ordering.

Room Creation

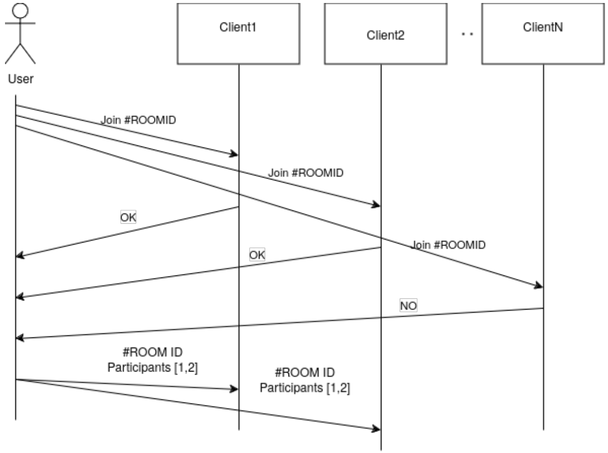

Figure: Room Creation Message Exchange

Communication

Messages

Text-Based, represented in JSON, belonging to two main categories

- **Multicast Ordered & Reliable** Room messages, timestamped with vector clocks and acked by all the recipients.
- **Multicast Non-Ordered & Not Reliable** Messages exchanged over the common multicast channel such as *HELLO/WELCOME/ROOM_CREATE_[REQUEST/REPLY]*

Main Loop

Client

The list of possible commands (numbered from 1 to 7) is shown to the user and taken as input throught a text prompt, then processed accordingly.

QueueThread

Outgoing messages are queued by the Client and consumed by the QueueThread, that sends them over the network throught the MulticastSocket of the specific room.
Events that require the user's attention are added to the client's event queue **only when it's in the main menu**, the client will ignore events when it's in a chat room for simplicity.