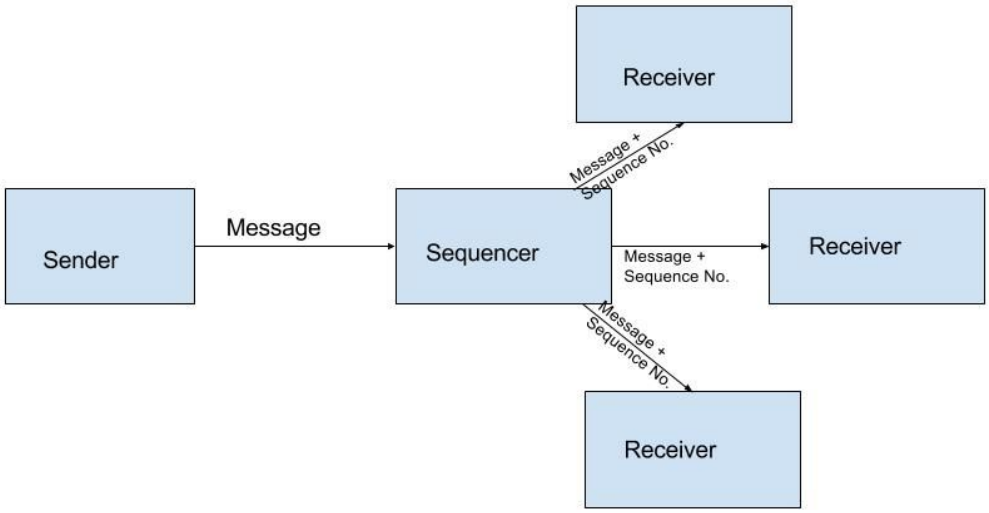
Report: Lab 1, Distributed systems advanced

- by Kruthika Suresh Ved and Romi Zaragatzky, Group 6

In this lab, we have implemented a reliable broadcast protocol that supports total and FIFO ordering. To do this we used the algorithms described in the paper *An efficient reliable broadcast protocol* [1]. We also added the handling of a sequencer crash.

The protocol consists of three parts that exist on each node: the sending, receiving and sequencer parts. There is always exactly one sequencer in a broadcast group at any point in time; the node knows that it is the sequencer when it's flag *isSeq* is set.



Sender

When a node needs to broadcast, it enters the *cast* method. The sender sets the MessageNr, which along with SenderID, will uniquely identify the message. The message is then added to the senderBuffer. The senderBuffer stores the messages which were sent to the sequencer, but not yet broadcasted. In the event of sequencer failure, sender will send all the messages in senderBuffer to the new sequencer. The sequencer takes care of ordering and broadcasting to everyone.

When sequencer completes the broadcasting procedure, it sends 'SUCCESS' message to the sender. The sender, upon receiving success message, removes the message from the senderBuffer.

ExampleMessage and MyMessage

In our implementation, we have 2 classes for messages → ExampleMessage and MyMessage. The former contains the header and data separately. This is the one which is

used extensively in our protocol. Header contains all the info required to run the protocol, like Sequence number for ordering. The data is the text to be delivered to higher level application.
However the lower layer takes in only Sender ID and text in string format. That's why we have implemented another class named MyMessage, which will be only used for sending and receiving purpose.

Sequencer

The sequencer, in this protocol, takes care of assigning Sequence numbers and broadcasting to everyone. When a Receiver detects that it has missed some messages, it requests sequencer to retransmit. The sequencer, then checks the history and retransmits the requested message.
Using the history buffer has a major disadvantage of being limited in size. We need to clear the buffer when history is full. To do this, whenever a sender sends a broadcast request, it also attaches the sequence number of last message received (piggybacked acknowledgement). The sequencer stores this information in SequenceStat array. When history is full, it clears all those messages from the history that has been already received by everyone. Now in some cases history still remains full, then the protocol starts two phase commit protocol. The protocol exchange is given in the Figure 1 below.

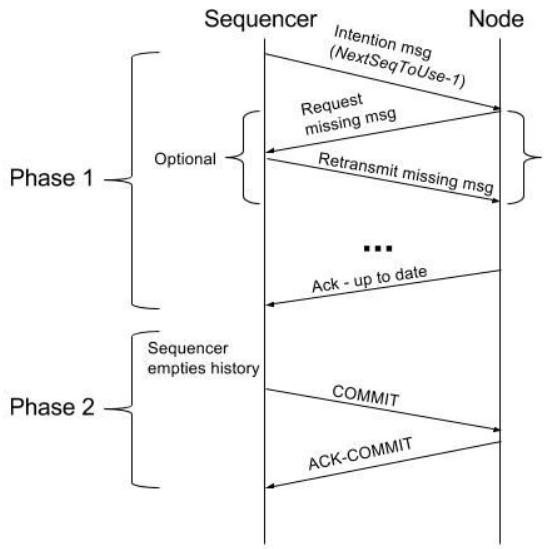


Figure 1: Two Phase Commit Protocol

During the two phase commit protocol exchange, any broadcast request received by the sequencer is put in SyncPhaseBuffer. After the sequencer receives ACK-COMMIT from all the members, it clears the complete history and then starts processing data from SyncPhaseBuffer.

Receiver

Receiver normally receives messages with type BROADCAST. It first checks whether it has missed some messages in between. For example: If the receiver expects a message with SequenceNr 4 but it received message with SequenceNr 6, implies that it has missed the packets with SequenceNr 4 and 5. It then requests the sequencer to retransmit those packets and adds the packet with SequenceNr 6 to the receiveBuffer. Every time it receives a packet, it checks the receiveBuffer and delivers the next expected packet to the higher application layer if available. The delivered packets are removed from the receiveBuffer.

Crash of Sequencer

Every processor maintains the list of the peers who are up and running. When the sequencer crashes, the process finds the live process with highest id and identifies it as the new sequencer. When the process figures out that no process with higher id than his is alive, it initializes the objects associated with the sequencer and sets its *isSeq* flag to true.

Requirements:

Integrity Requirements:

Integrity means *'A correct process delivers a message at most once.'*
The sender sends the message to sequencer only one single time. The sequencer broadcasts to the receiver only one time, unless receiver requests for retransmission. The receiver requests for retransmission only when it knows it has missed a particular message. Therefore, no one delivers a duplicate message.
However, at the time of sequencer failure, the sender sends the message, which still did not get broadcast success message, to the new sequencer for broadcasting it. This does not violate integrity requirements as broadcast happens only once, even with the sequencer crash.

Validity Requirements:

Validity means '*A message from a correct process will be delivered by the process eventually.'*
When there is no sequencer crash, everything works out fine.
When there is a crash, the sender has the senderBuffer in which he stores all the messages which have been sent but not yet broadcasted. It resends all those messages to the new sequencer. Thus, the message is broadcasted eventually.

Agreement Requirements:

Agreement means *'A message delivered by a correct process will be delivered by all other correct processes in the group.'*

Every receiver receives the broadcast sent by the sequencer. If it misses a packet, it asks the sequencer for retransmission. Every broadcast the receiver receives is delivered to the higher level application eventually.

Ordering Requirements:

This protocol satisfies Total Ordering using SequenceNr and FIFO ordering using MessageNr assigned by the sender. The central entity - sequencer - is in charge of the ordering.

References:

[1] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, Henri E. Bal, “An efficient reliable broadcast protocol”, Dept. of Mathematics and Computer Science, Vrije Universiteit Amsterdam, The Netherlands