University of St Andrews

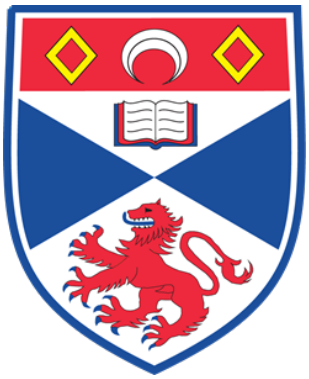
Distributed Systems

CS4103

Ring-Based Distributed System

Author:
150008022

April 23, 2019



Goal

To demonstrate an understanding of leader election and mutual exclusion in distributed systems by developing a ring-based distributed social media application.

Contents

1 Initial Set-up

Java 8 was chosen for this project due to it's friendly socket API, and Maven [1] was used as the build tool.

Since each node would be running on an isolated machine, the planned testing environment would involve using *ssh* to start the nodes remotely. Providing the configuration as command line arguments was considered simpler than other methods, and was implemented using the Apache Commons CLI library [2].

In order to run the program, the arguments shown in table 1 had to be specified. The purpose of each argument is explained later.

usage: java -jar ⟨program⟩.jar	
-d,–drop	Include if this node should trigger a database refresh.
-e,–election ⟨arg⟩	Election method to use.
-f,–list ⟨arg⟩	Path to file containing list of nodes (resource P).
-i,–id ⟨arg⟩	ID of this node.

Table 1: Arguments for running application.

2 Ring Formation

2.1 Messages

In order for nodes to understand and parse the messages that were sent between them, a message protocol was designed. Figure 1 shows the header that would be part of every message sent. The possible types of messages are summarised in table 2.

The messages consist of serialized Java objects, which requires that they all implement the *Serializable* interface. A more versatile format such as JSON would be preferred, but since all nodes would be using Java this method of serializing messages was kept. Figure 1 shows the header of every message, and figure 2 shows the header for an election message.

Message Type	Usage	Payload
JOIN	Sent by new node to coordinator when wanting to join ring.	None
SUCCESSOR	Sent by coordinator to inform a node to connect to a new successor.	New Successor ID
SUCCESSOR_REQUEST	Sent by a node to the coordinator after its successor has failed in order to be reassigned a successor.	None
TOKEN	Sent by a node to its successor.	None
TOKEN_ACK	Sent by a node to its predecessor after receiving the token to confirm its reception.	None
COORDINATOR_ELECTION	Message payload contains an election message.	Election Message

Table 2: Possible values for the type field in the message header, with their usage and payload.

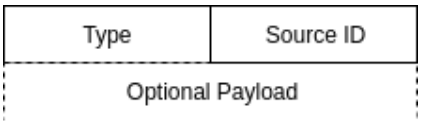


Figure 1: Main message header.

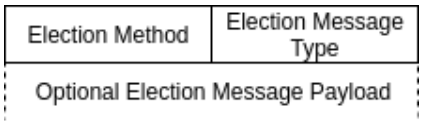


Figure 2: Election message header.

The election message header would be in the payload of the main message. This allowed multiple election methods to be used and easy routing of election related messages to an election handler class.

2.2 Communication

Due to different patterns of communication during recovery and regular message passing phases both TCP and UDP were used for this project, as shown in Figure **??**.
TCP was used for communication **around** the ring:

1. Reliable communication avoids token being lost.
2. Connection is reused frequently between predecessors and successors, justifying handshake overhead.

UDP was used for communication **across** the ring:

1. Low communication overhead allows for messages to be sent to multiple nodes quickly, improving recovery time from node failure.
2. No session maintenance allows coordinator to handle more members in the ring.
3. Multicast communication possible which would greatly simplify broadcasting.

2.3 Initialization

On initialization, each node sends out join messages to the coordinator. The joining protocol is similiar to that described in [3], and is shown in Figure 3. Figure 4 shows how the state of each connection changes over the course of the joining procedure.

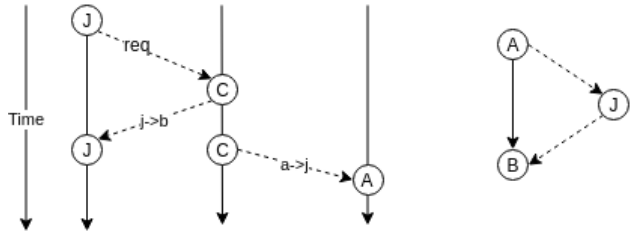


Figure 3: Joining protocol over time, with topology shown on right.

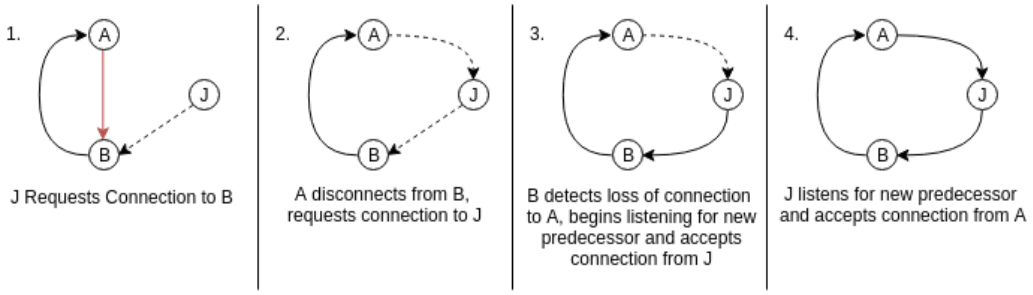


Figure 4: State of each connection as a new node joins the ring. The red arrow shows the edge that is replaced with the new node, and solid and dashed arrows represent ongoing and pending connections respectively. Either A or B can be the coordinator in this scenario.

When node J wants to join the ring:

1. J sends join request to coordinator C.
2. C sends successor to J, telling it to connect to B.
3. C sends successor to A, telling it to connect to J.
4. A disconnects from B, B begins listening for new predecessor.

5. J connects to B, A connects to J.

In order for this process to work for a ring network with a single node, it required another thread to wait on the connection, and the main thread would then request the connection to itself. For two nodes and more the joining node would be able to first connect to its new successor once its predecessor had disconnected from its old successor as shown in figure 4.

2.4 General Node Recovery

When designing a distributed system, it should be assumed that failure will occur, and so as an extension this implementation included a mechanism for recovering from failure. Node failure is detected by either:

1. a node attempting to read the socket connected to its predecessor.

2. a node forwarding the token to its successor and not receiving an acknowledgement within a given timeframe.

When a failure is detected by the successor of a failing node, it will simply begin listening for a new predecessor. Once the predecessor of the failing node detects the loss of connection, it will request a new successor from the coordinator node. The coordinator will then reply with the ID of the node after the failed node for the original predeccessor to connect to, at which point normal network behaviour can resume. Figure 5 shows the topology change when a node fails.

The token acknowledgement message is the only occurence of communication in the reverse direction of the ring topology. The predecessor of the failing node will hold onto the token until it is assigned a new successor, and only releases the token once it has received the acknowlegement from it. This limits the number of situtations that can result in the loss of the token. If somehow the token is acknowledged but the acknowledgement message does not reach the predecessor in time, it could be possible for two tokens to then be in circulation.

During testing, an edge case was realised with this recovery mechanism. If the disconnect is discovered when reading from the predecessor socket, there is a case where just waiting for a new predecessor does not work. Figure 6 shows two scenarios where the disconnect is realised while reading from the predecessor socket.

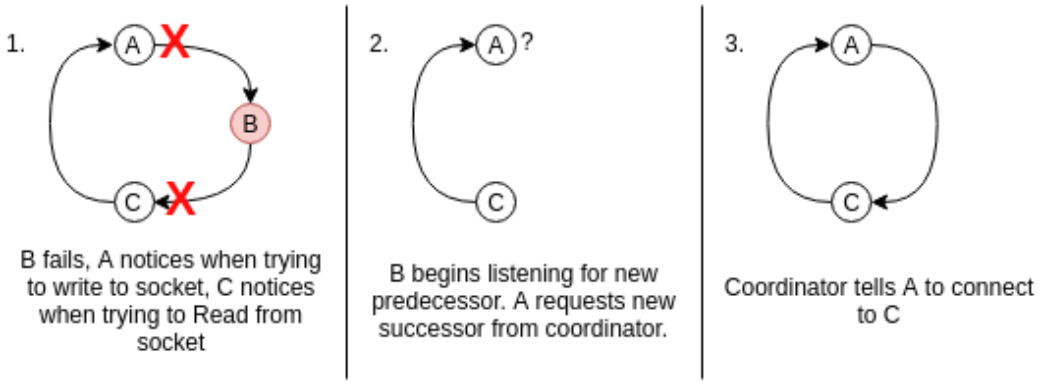


Figure 5: Network topology during the recovery from node B failing. Either A or C are the coordinator in this scenario.

The scenario on the left would require A to first designate itself as its new successor before performing the self connection procedure described in the initialization protocol. This would only be necessary if the network consisted of two nodes, otherwise another node would eventually connect to A. The scenario on the right however is not one where an actual failure has occurred, and instead A is just changing its predecessor from itself to B. In order to check that the disconnect had occurred not because of failure, the token ring socket class (*RingCommunicationHandler*) provided a boolean flag that would be true if the most recent disconnect had occurred while connected to itself.

Knowing this meant that it could be assumed if there were only two nodes in the network that the other node has failed, and so a self connection would have to be performed as described in the previous section.

2.5 Ring Topology Persistence

Initially, the coordinator node was always the node with the ID 6. This node would maintain an in-memory virtual representation of the network topology using a cyclic linked list of node IDs. This obviously would be useless if the coordinator was to change, and so it was necessary to persist the network topology somehow.

This was achieved by adding a database which would behave like resource P in the practical specification (the list of nodes). Only the coordinator

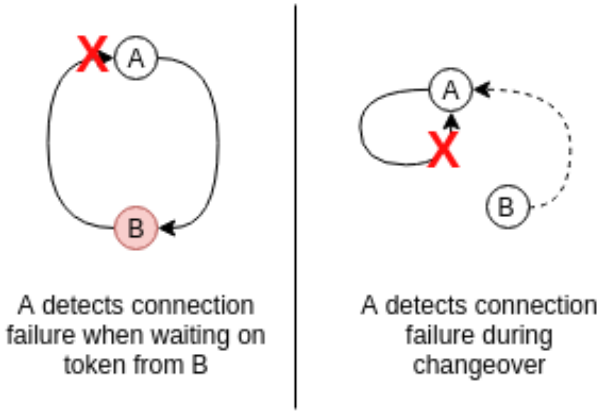


Figure 6: Two predecessor disconnect scenarios that are identical to the thread polling the predecessor socket.

would be able to write and update the database, and all other nodes would read from it. The database allowed operations to be carried out as transactions, meaning that node failure during an update could be rolled back. The database is initialized using a csv file containing the ID, address, and port of each node that *may* join the network at some point. Figure 7 shows an ER diagram of the database schema. The successorId is nullable and refers to another row in the node table. If a value is present in that column then the node ID on that row would be assumed to be part of the ring.

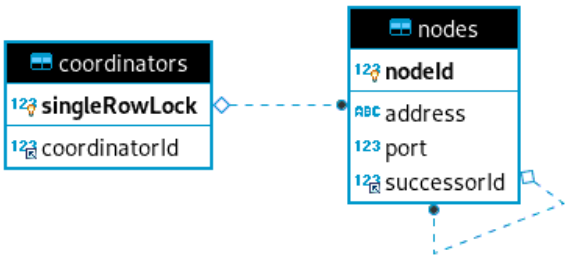


Figure 7: ER diagram of the database schema for persisting the network topology.

When a new node wants to join the network, it first queries the database to learn who the current coordinator is. If no coordinator exists and no

nodes have successors set, then this node will assume coordinator status. A limitation of this functionality is that if another node (B) queries for the current coordinator before the first node (A) has had the chance to update the database, then B could potentially also designate itself as coordinator. Since B would overwrite A as coordinator, later joining nodes would always make requests to B, and A would remain connected to itself.

The database was also essential for recovering the network topology quickly when a coordinator failed. If a node detected the failure of its successor and knew that its successor was currently the coordinator, it would temporarily act as a coordinator until a new one was elected. This would allow it to update the database and recover the ring topology by connecting to the successor of the failed coordinator. Immediately after recovering this connection, an election would be triggered by the acting coordinator in order to designate a permanent coordinator.

In order to ensure that only one coordinator existed at any time, the coordinators table included a boolean column *singleRowLock* that would always be true. By setting this boolean column as the primary key, insert statements could then be constructed to either

1. Insert a new row if no coordinator exists with the value TRUE for *singleRowLock*.

2. Change the value of the coordinator ID if a row already exists by using the 'ON DUPLICATE KEY UPDATE' statement in the INSERT statement.

Whilst another row could be added with the value FALSE for *singleRowLock*, permissions would be configured such that only the hosts and a system administrator would be able to perform updates so it could be assumed that as long as the application code always used TRUE for inserts, only one row would ever exist.

3 Leader/Coordinator Election

Firstly the ring-based election algorithm was implemented, as required for the basic specification. An interface was used for providing a generic 'election algorithm' API, allowing different algorithms to be switched out easily. Any messages that arrived from either the UDP socket or the ring sockets that were related

4 Receive/Send Posts

References

[1] Apache. Apache maven project. `https://maven.apache.org/`.

[2] Apache Commons. Apache commons cli. `https://commons.apache.org/proper/commons-cli/`.

[3] D. Lee, A. Puri, P. Varaiya, R. Sengupta, R. Attias, and S. Tripakis. A wireless token ring protocol for ad-hoc networks. In *Proceedings, IEEE Aerospace Conference*, volume 3, pages 3–3, March 2002.