University of St Andrews

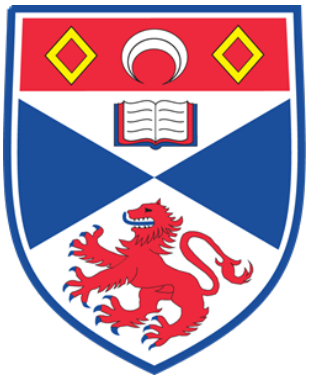
Distributed Systems

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 **1**

2 Ring Formation **1**
- 2.1 Messages . . . 1
- 2.2 Transport Protocol . . . 2
- 2.3 Initialization and Joining . . . 3
- 2.4 Topology Recovery and Maintenance . . . 5
- 2.5 Topology Persistence . . . 7

3 Leader/Coordinator Election **10**
- 3.1 Election API . . . 10
- 3.2 Triggering Elections . . . 10
- 3.3 Ring-Based . . . 11
- 3.4 Chang Roberts . . . 11
- 3.5 Bully . . . 12

4 Receive/Send Posts **13**
- 4.1 Mutual Exclusion . . . 13
- 4.2 Web Server . . . 14

5 Testing **15**

6 Evaluation **15**

7 Summary **15**

8 How to Run **15**

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 are sent as serialized Java objects, which requires that all message classes implement the *Serializable* interface. A more platform independent format such as JSON would be preferred, but since all nodes would be using Java this method of serializing messages was kept. Figure 2 shows the header for an election message. The election message header would be in the payload of the main message. This allowed multiple election

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.

methods to be used and easy routing of election related messages to an election handler class.

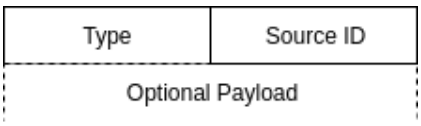


Figure 1: Main message header.

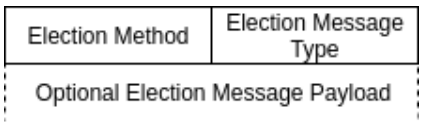


Figure 2: Election message header.

2.2 Transport Protocol

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 3.

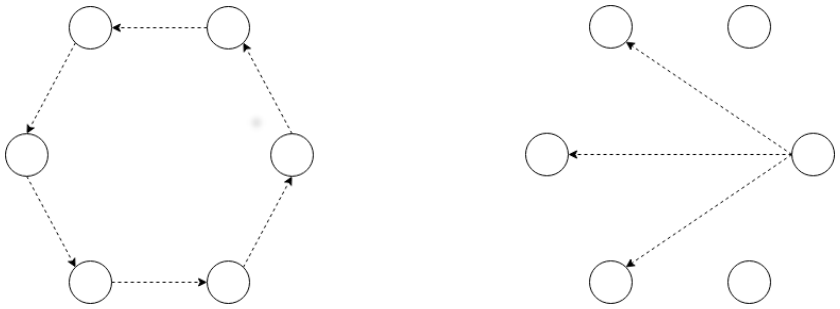
TCP was used for communication **around** the ring:

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.



(a) TCP Around Ring (b) UDP Across Ring

Figure 3: The different types of communication used

2.3 Initialization and Joining

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 4. Figure 5 shows how the state of each connection changes over the course of the joining procedure.

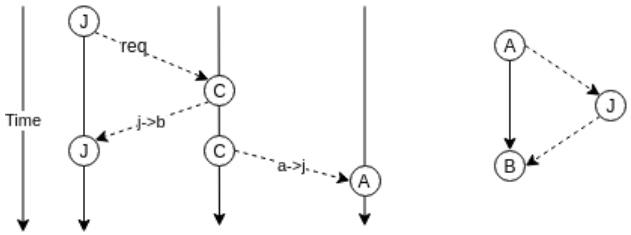


Figure 4: Joining protocol over time, with topology shown on right.

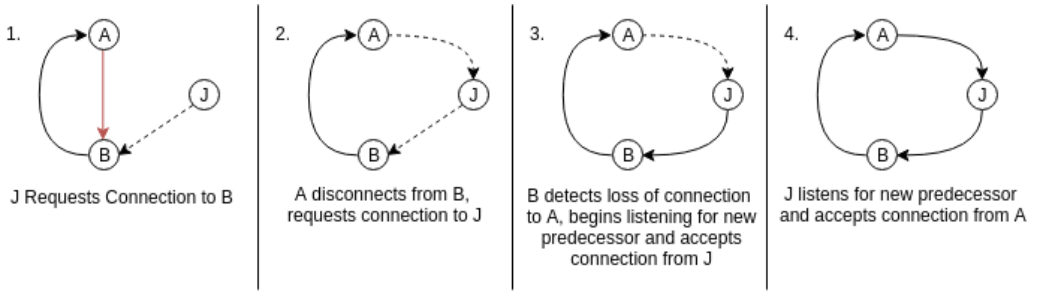


Figure 5: State of each connection as a new node joins the ring. The red arrow shows the edge that is replaced with the new node, and solid and dashed arrows represent ongoing and pending connections respectively. Either A or B can be the coordinator in this scenario.

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 5.

Figure 6 shows the logs from node 2 as it joins the ring, and figure 7 shows the logs from the coordinator as it handles the request. Figure 8 shows the logs from node 6 when it initializes the ring network by connecting to itself.

```
[2019-04-23 17:17:50 | 2 ] [INFO   ] Requesting successor from coordinator 8
[2019-04-23 17:17:50 | 2 ] [INFO   ] Sending message to 8 : Message{type=JOIN, srcId=2, payload=null}
[2019-04-23 17:17:50 | 2 ] [INFO   ] Waiting for coordinator to tell me my successor
[2019-04-23 17:17:50 | 2 ] [INFO   ] Received message: Message{type=SUCCESSOR, srcId=8, payload=SuccessorMessage{successorId=5}}
[2019-04-23 17:17:50 | 2 ] [INFO   ] Waiting on predecessor connection
[2019-04-23 17:17:50 | 2 ] [INFO   ] Predecessor connected from address to /127.0.0.1:58214
[2019-04-23 17:17:50 | 2 ] [INFO   ] Updating successor to localhost/127.0.0.1:5005
```

Figure 6: Logs from node 2 when joining the ring with coordinator ID 8.

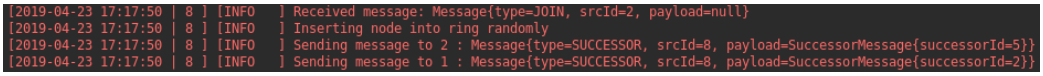


Figure 7: Logs from coordinator node 8 when node 2 wishes to join.

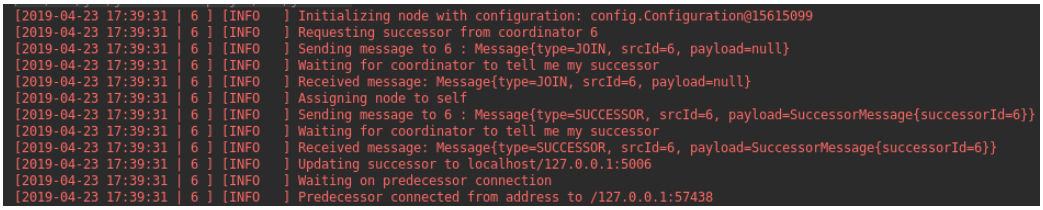


Figure 8: Logs from node 6 when it initializes the ring network and connects to itself.

2.4 Topology Recovery and Maintenance

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 9 shows the topology change when a node fails. Since the failure is detected by the ring handling thread, and the new successor message arrives on the UDPSocket handling thread, the ring handling thread simply blocks until notified by the UDPSocket thread that a new successor has now been connected to.

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

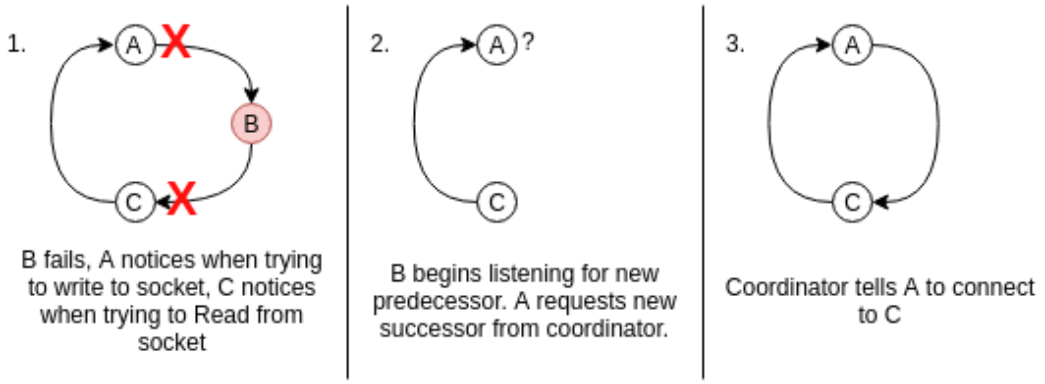


Figure 9: Network topology during the recovery from node B failing. Either A or C are the coordinator in this scenario.

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. Figure 10 shows logs from a node that has experienced successor failure.

```
[2019-04-23 17:44:11 | 6 ] [WARNING] Lost connection to successor
[2019-04-23 17:44:11 | 6 ] [INFO   ] Requesting successor from coordinator 8
[2019-04-23 17:44:11 | 6 ] [INFO   ] Sending message to 8 : Message{type=SUCCESSOR_REQUEST, srcId=6, payload=null}
[2019-04-23 17:44:11 | 6 ] [INFO   ] Received message: Message{type=SUCCESSOR, srcId=8, payload=SuccessorMessage{successorId=5}}
[2019-04-23 17:44:11 | 6 ] [INFO   ] Updating successor to localhost/127.0.0.1:5005
[2019-04-23 17:44:11 | 6 ] [INFO   ] Returning token to forwarding queue
[2019-04-23 17:44:11 | 6 ] [INFO   ] Retrying token send
[2019-04-23 17:44:11 | 6 ] [INFO   ] Forwarding token
[2019-04-23 17:44:11 | 6 ] [INFO   ] Waiting on token ACK
[2019-04-23 17:44:11 | 6 ] [INFO   ] Token forwarded
```

Figure 10: Logs from node 6 when it's successor fails to acknowledge it's sent token.

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 11 shows two scenarios where the disconnect is realised while reading from the predecessor socket.

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.

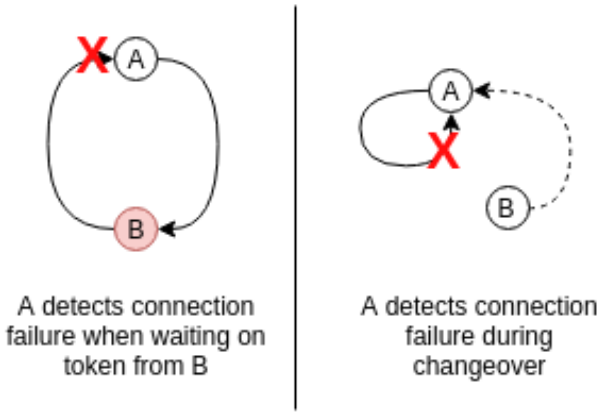


Figure 11: Two predecessor disconnect scenarios that are identical to the thread polling the predecessor socket.

2.5 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 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 12 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.

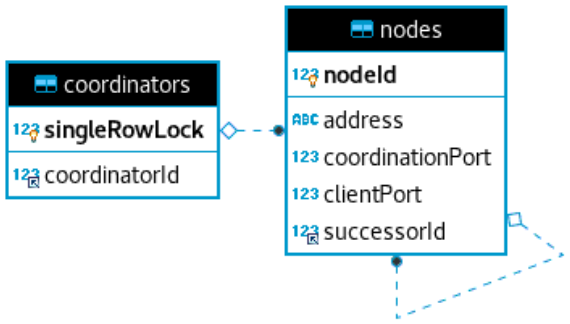


Figure 12: ER diagram of the database schema for persisting the network topology.

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.

```
MariaDB [jm354_distsys]> select * from nodes;
+--------+-----------+-----------------+------------+-------------+
| nodeId | address   | coordinationPort | clientPort | successorId |
+--------+-----------+-----------------+------------+-------------+
|      1 | 127.0.0.1 |            5001 |       8001 |           3 |
|      2 | 127.0.0.1 |            5002 |       8002 |           1 |
|      3 | 127.0.0.1 |            5003 |       8003 |           4 |
|      4 | 127.0.0.1 |            5004 |       8004 |           6 |
|      5 | 127.0.0.1 |            5005 |       8005 |           7 |
|      6 | 127.0.0.1 |            5006 |       8006 |           8 |
|      7 | 127.0.0.1 |            5007 |       8007 |           2 |
|      8 | 127.0.0.1 |            5008 |       8008 |           5 |
+--------+-----------+-----------------+------------+-------------+
```

(a) Node database table.

```
MariaDB [jm354_distsys]> select * from coordinators;
+---------------+---------------+
| coordinatorId | singleRowLock |
+---------------+---------------+
|             8 |             1 |
+---------------+---------------+
```

(b) Coordinator database table.

Figure 13: The state of the database after nodes join in the order 6, 5, 8, 3, 4, 7, 1, 2.

```
MariaDB [jm354_distsys]> select * from nodes;
+--------+-----------+------+-------------+
| nodeId | address   | port | successorId |
+--------+-----------+------+-------------+
|      1 | localhost | 5001 |           2 |
|      2 | localhost | 5002 |           5 |
|      3 | localhost | 5003 |           4 |
|      4 | localhost | 5004 |           6 |
|      5 | localhost | 5005 |           3 |
|      6 | localhost | 5006 |           7 |
|      7 | localhost | 5007 |           1 |
|      8 | localhost | 5008 |        NULL |
+--------+-----------+------+-------------+
```

(a) Node database table.

```
MariaDB [jm354_distsys]> select * from coordinators;
+---------------+---------------+
| coordinatorId | singleRowLock |
+---------------+---------------+
|             7 |             1 |
+---------------+---------------+
```

(b) Coordinator database table.

Figure 14: The state of the database after recovering from node 8 failing.

Figure 13 shows the state of the database after 8 nodes join in a random order. Figure 14 shows the state of the database after node 8 fails and a new coordinator is elected.

3 Leader/Coordinator Election

3.1 Election API

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 had the COORDINATOR_ELECTION type would be routed to the handler for the election algorithm specified in the election header. For example, an election message that required the bully algorithm would be routed to the bully election handler.

In the configuration for each node, a default election algorithm is specified. Therefore the type of election that would be started could vary depending on which node initiated it. If a node received an election message for a different algorithm to the one they are currently using, then they would replace their election handler with one that could process the arriving message.

The methods required by the election API are summarised in table 3. The constructors could vary since different resources may be necessary. For example, the ring-based and Chang Roberts algorithms both only require communication around the ring, whilst the Bully algorithm only requires communication across the ring.

Method	Usage
getMethodName	Returns the name of the underlying algorithm used by the implementation of this interface.
startElection	Starts an election.
handleMessage	Handles any election messages.
getResult	Returns the result of the election (if concluded).
electionConcluded	Returns true if a result has been obtained.

Table 3: Methods of the election handler interface.

3.2 Triggering Elections

Initially, elections could only be triggered by the predecessor of a failed coordinator. This meant that if a new node joined with a higher ID, it would not be elected until the current coordinator failed. This was resolved by

adding logic to the joining procedure that once part of the ring, the new node would trigger an election if it found that its ID was greater than the current coordinators.

This meant it would be possible for multiple elections to be ongoing simultaneously if many nodes joined the ring in quick succession with higher IDs than the current coordinator.

3.3 Ring-Based

The ring based algorithm was the simplest to implement and test. The node starting the election would initialise the ELECTION message with its own ID and forward this to its successor and set the *ongoing* flag to true. The message would be forwarded around the ring by each node after appending their ID to the list of candidates until it returned to the node that started the election.

This node then chose the new coordinator from the list of candidates based on which had the highest ID, and forwarded the result to its successors as a COORDINATOR message. The result would be forwarded by each node until it reached a node that was already aware of the new coordinator.

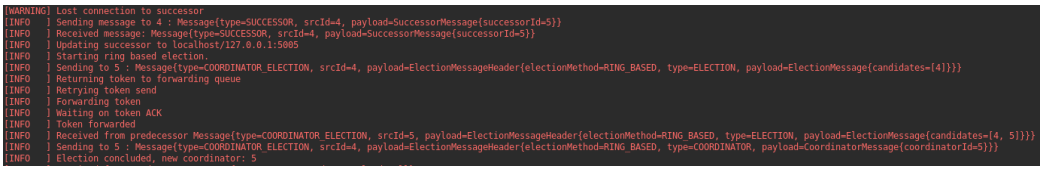


Figure 15: Logs from node 2 after detecting coordinator failure. Note how token passing resumes as soon as a new successor is found despite the loss of the coordinator in order to make node failure transparent.

3.4 Chang Roberts

The Chang Roberts algorithm reused most of the components from the ring-based algorithm, other than the forwarding logic for ELECTION messages.

If the ELECTION message contained the receiving nodes ID, then it would conclude the election and forward the COORDINATOR message as in the ring-based algorithm. However, in order to avoid redundant forwarding

of election messages if two elections are started at the same time, ELECTION messages would only be forwarded if the receiving node was a better candidate (i.e. had a greater ID), or if it had not already participated by forwarding a message previously.

Figure 16 shows logs from a new node that has joined the ring and triggered an election. Initially the election message is sent, and later it returns with the same ID and so node 8 elects itself. It then forwards the coordinator message until it arrives back to itself at which point it is dropped.

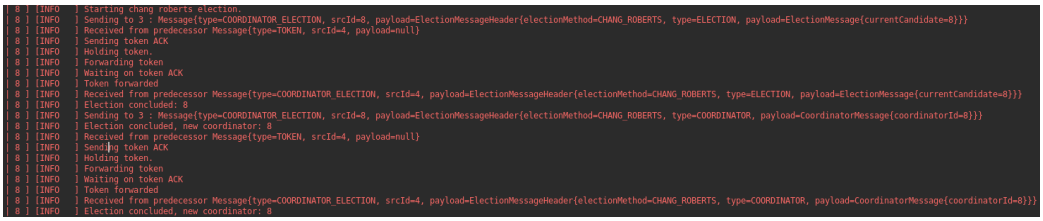


Figure 16: Logs from node 8 when it joins the ring and notices its ID is greater than the current coordinator.

3.5 Bully

The bully algorithm was the most difficult to implement due to its dependence on timing. An election would be started by a node by sending an ELECTION message to all nodes with IDs greater than it's own. It would then create a thread which would sleep for a preset time before assuming it is to be the coordinator. If one of the other nodes that received the ELECTION message replied with an OK message, the sleeping thread would simply terminate after the preset time had expired. Another thread would be spawned to again sleep for a duration of time after receiving the OK message. If no coordinator message was received before this second duration elapsed, then the node would restart the election.

Each node that received an ELECTION message would reply with an OK message if it had not already started an election, and then carry out the same procedure as the first node had to trigger the election. Eventually the node with the highest ID would not have any other nodes to send ELECTION messages to, and so it would consider itself the coordinator and inform all the other nodes in the ring using the COORDINATOR message.

Figure 17 shows how effective the bully algorithm is at dealing with failure and demonstrates a node being bullied into letting another node be coordinator. The order of events in the logs shown are:

1. Node 6 receives election message from node 4.
2. 6 replies with OK message, and begins its own election by sending an election message to node 7.
3. 6 also receives election message from node 5, but ignores it since it is already carrying out an election.
4. Timeout occurs without 7 sending OK message, 6 tries to elect self as coordinator and sends coordinator message to all other nodes.
5. Node 7 has in the meantime elected itself as coordinator, and 'bullies' node 6 into letting it be coordinator.

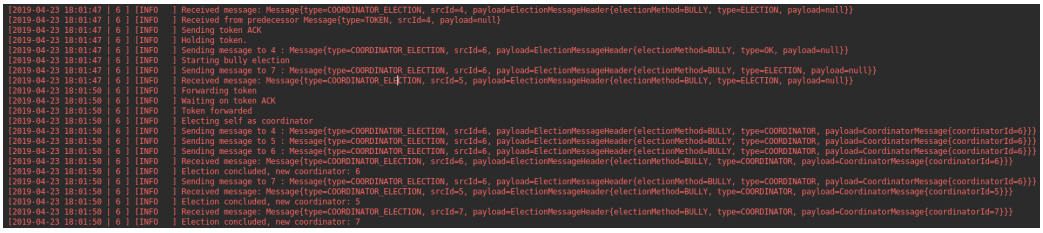


Figure 17: Logs from node 6 during an election where coordinator 8 has failed, and 7 should be the next coordinator.

4 Receive/Send Posts

4.1 Mutual Exclusion

At any one time, only one server/node should be able to read or write from the message queue. In order to provide this mutual exclusion, the token-ring method was used. A token is created by the first node in the ring when initializing the network, and is forwarded around the ring after a) the node holding the token carries out a operation on the resource or b) a timeout expires.

Since only one node can have the token at any one time, mutual exclusion is ensured. No starvation can occur since the token will always eventually return to each node in the ring. The downsides of the token-ring algorithm is that the wait time between each operation grows linearly with the size of the ring. It also requires some sort of recovery mechanism if the node holding the token fails.

In order to ensure that the token is at least delivered successfully, the token acknowledgement message described earlier for detecting lost nodes was utilised. The following steps are taken between waiting on a token and forwarding it:

1. Ring message handling thread polls predecessor socket for token.
2. Client handling thread polls the single element *usableTokenQueue* waiting for the token to become available.
3. When the token arrives, the ring message thread replies with an ACK, and puts the token in the *usableTokenQueue* for the client to consume. It then begins polling the single element *forwardableTokenQueue* where the client will return the token.
4. The client dequeues the token from the *usableTokenQueue* and uses it to serve it's clients then returns it to ring thread via the *forwardableTokenQueue*.
5. Ring thread then removes the token from the *forwardableTokenQueue* and forwards it onto it's successor.

4.2 Web Server

In order for clients to make and receive posts, a simple web server was configured. In order to isolate coordination traffic from client traffic, the two are served on different ports. In a real-life scenario, the web server would run on port 80 as standard and coordination traffic would also run on a consistent port across all hosts. However, to allow for local testing, it was necessary to use port numbers to differentiate between host sockets.

This was actually the first time I had implemented a HTTP web server at such a low level, and so it was constructed step by step. First a simple hello world message was sent to clients, then a HTML document, then the ability to receive and send messages was added.

In a production environment, a ring topology would likely be abstracted by a load balancer. This load balancer is what clients would connect to when wanting to use the service by connecting at some FQDN. It would then proxy messages between clients and the server in the ring that is responsible for them, which it could choose based on how many clients each server is already serving.

5 Testing

6 Evaluation

7 Summary

8 How to Run

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.