University of St Andrews

Distributed Systems

CS4103

Ring-Based Distributed System

Author:
150008022

April 22, 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.

1.1 Configuration

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.

1.2 Sockets

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 1.

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

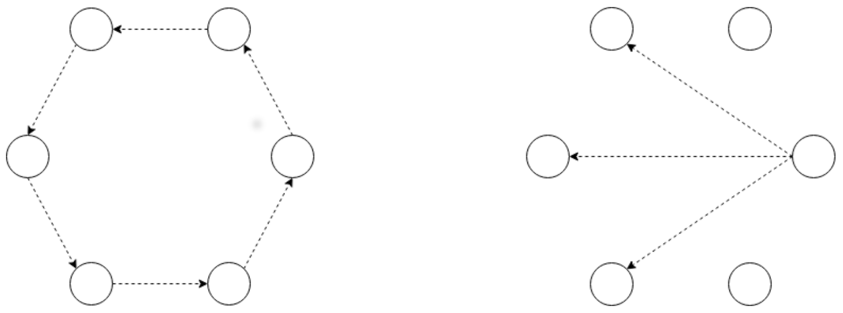
1. Reliable communication avoids token being lost.
2. Connection is reused frequently between predecessors and successors, justifying handshake overhead.

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 1: The different types of communication used

2 Ring Formation

2.1 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 2. Figure 3 shows how the state of each connection changes over the course of the joining procedure.

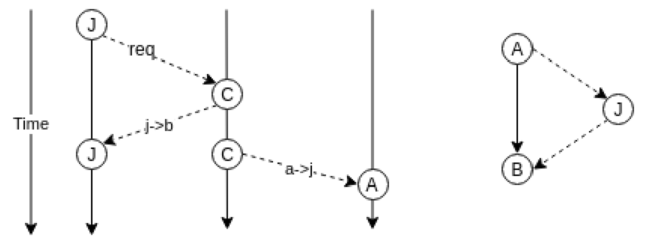


Figure 2: Joining protocol over time, with topology shown on right.

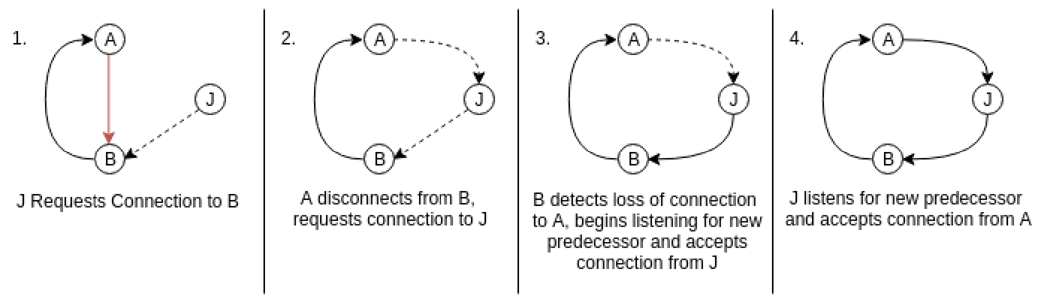

1.
A
J
B
J Requests Connection to B
2.
A
J
B
A disconnects from B, requests connection to J
3.
A
J
B
B detects loss of connection to A, begins listening for new predecessor and accepts connection from J
4.
A
J
B
J listens for new predecessor and accepts connection from A

Figure 3: 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 3.

2.2 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 4 shows the topology change when a node fails.

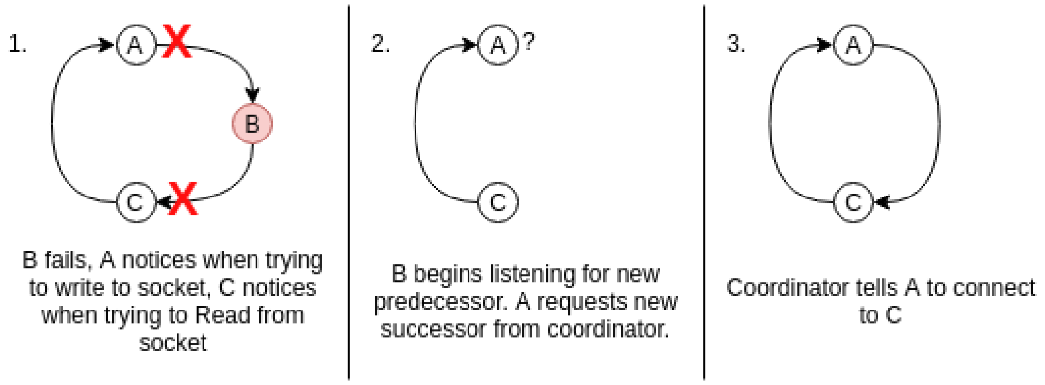

1.
A
B
C
B fails, A notices when trying to write to socket, C notices when trying to Read from socket
2.
A
?
C
B begins listening for new predecessor. A requests new successor from coordinator.
3.
A
C
Coordinator tells A to connect to C

Figure 4: Network topology during the recovery from node B failing. Either A or C are the coordinator in this scenario.

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 message does not reach the predecessor in time, it could be possible for two tokens to then be in circulation.

3 Leader/Coordinator Election

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.