Implementation and Testing of the Leader Election Algorithm

Author: Yizhou Lu

1. Introduction

- The need for a consistent leader in distributed systems ensures coordinated action and resource management.
- The Chang and Roberts Leader Election algorithm was proposed as a solution for electing a leader in a distributed network with a ring topology.
- This report aims to implement this algorithm using Akka in Java and evaluate its performance.

2. Implementation Details:

(1) Akka Actor system setup - SimpleLeaderElectionAlgorithm

- In the main class the whole actor system is built. First is 'ActorSystem', then each process (actor) is created in a loop and stored in an array 'processes', they are created through another class 'ProcessActor'. Each process has its own UID. Here we create 18 processes as example.

```java
// Instantiate an actor system
final ActorSystem system = ActorSystem.create("System");

// Assuming there are 18 processes in the ring
ActorRef[] processes = new ActorRef[18];
for (int i = 0; i < processes.length; i++) {
    processes[i] = system.actorOf(ProcessActor.createActor(i),
"process" + i);
}
```

- Then we set up the ring topology, this is achieved in a loop. In each iteration we send a message to the current process, this message contains information about it's neighbor, which is next process. Since we have a class 'NextRef', this is considered as a message type, used for store neighbor information.

```java
// Set up the ring topology - reference to the next process
for (int i = 0; i < processes.length; i++) {
        ActorRef nextProcess = processes[(i + 1) % processes.length];
        processes[i].tell(new NextRef(nextProcess),
ActorRef.noSender());
}
```

- Finally we can choose any process to start the election. Here we choose to start with process 4, so we send a 'ElectionMessage' to process 5. This message type will be explained later.

```
// We can choose any process to start the election
// Here process 4 starts the election by sending election message to
its neighbor process 5
processes[5].tell(new ElectionMessage(4), ActorRef.noSender());
```

(2) Build Actor Class - ProcessActor

- First is basic set up.
- What we need to pay attention is an int type variable 'id' is created to be each process's UID; an boolean type variable 'participating' is to show weather this process is participated in the election activity.
- The variable next is for reference to the process's neighbor (next process), electedId is for storing the UID of elected leader.

```
// Logger attached to actor
    private final LoggingAdapter log =
Logging.getLogger(getContext().getSystem(), this);

    private final int id;
    private boolean participating = false; // Initially each process in
the ring is marked as non-participant.

    // Actor reference
    private ActorRef next;
    private int electedId;


    public ProcessActor(int id) {
        this.id = id;
    }

    static public Props createActor(int id) {
        return Props.create(ProcessActor.class, () -> new
ProcessActor(id));
    }
```

- Now is the key part of this project, this is how messages are passed. We define the 'onReceive' method, it handles the message taht the processs receives.
- In this system there are three types of message: 'NextRef', 'ElectionMessage' and 'ElectedMessage'.
- For 'NextRef' message, it stores the information about next process, so that the next process can be referenced to the current process.
- For 'ElectionMessage' and 'ElectedMessage' message, I encapsulated the message processing methods in corresponding methods.

```
@Override
public void onReceive(Object message) throws Throwable {
    if (message instanceof NextRef) {
        this.next = ((NextRef) message).getNext(); // References to the
next process
    }
    if (message instanceof ElectionMessage) {
        ElectionMessage electionMessage = (ElectionMessage) message;
        handleElectionMessage(electionMessage);
    }
    if (message instanceof ElectedMessage) {
        ElectedMessage electedMessage = (ElectedMessage) message;
        handleElectedMessage(electedMessage);
    }
        }
```

- This is the way to handle 'ElectionMessage', first is to compare 'incomingId' with its own 'id', then different actions are taken.
- If 'incomingId' is bigger, then current process forwards this election message to next process, 'incomingId' stay unchanged. In the same time the state of 'participating' will be true.
- If 'incomingId' is smaller and the current process is already participated, this message is dropped; if current process is not participated, then 'participating' becomes true and new election message with current id will be sent.
- If 'incomingId' is equal to current 'id', then current process is chosen as the leader, and it sends 'ElectedMessage' to next process, which contains the leader's id information.

```
private void handleElectionMessage(ElectionMessage message) {
    int incomingId = ((ElectionMessage) message).getId();
    if (incomingId > this.id) {
        participating = true;
        sendElectionMessage(incomingId); // Forward the election
message with the same incoming ID
    }
    if (incomingId < this.id) {
        if(!participating) {
            participating = true; // Mark self as participant
            sendElectionMessage(this.id); // Send an new election
message with own ID
        } else {
            log.info("Process {} discards the election message",
this.id);
        }
    }
    if (incomingId == this.id) {
        log.info("Process {} starts acting as the leader", this.id);
        participating = false; // The leader process marks itself as
non-participant
        this.electedId = this.id; // Records the elected UID
        sendElectedMessage(this.id);
    }
}
```

- For 'ElectedMessage', when a process receives this message, if this process is not elected leader, then 'participating' state turns to false, and the 'electedId' is recorded in current process. After that this message is forwarded with nothing changed. If current process is the leader, then this election is completed.

```
private void handleElectedMessage(ElectedMessage message) {
        int electedId = ((ElectedMessage) message).getId();
        if (electedId != this.id) {
            participating = false;
            this.electedId = electedId; // Records the elected UID
            log.info("Process {} records {} as the leader", this.id,
electedId);
            next.tell(message, getSelf()); // Forwards the elected
message unchanged
        } else {
            log.info("Process {} received its own elected message. The
election is complete.", this.id);
        }
    }
```

- This is how 'ElectionMessage' and 'ElectedMessage' are sent, with corresponding id information as parameter.

```
private void sendElectionMessage(int id) {
    log.info("Process {} is sending an election message with ID {}",
this.id, id);
    next.tell(new ElectionMessage(id), getSelf());
}

private void sendElectedMessage(int id) {
    log.info("Process {} is sending an elected message with ID {}",
this.id, id);
    next.tell(new ElectedMessage(id), getSelf());
}
```

- Define message type 'ElectionMessage'

```
public class ElectionMessage {
    private final int id;

    public ElectionMessage(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
```

- Define message type 'ElectedMessage'

```
public class ElectedMessage {
    private final int id;
    public ElectedMessage(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
```

- Define message type 'NextRef', storing information about next process

```
public class NextRef {
    private final ActorRef next;

    public NextRef(ActorRef next) {
        this.next = next;
    }

    public ActorRef getNext() {
        return next;
    }
}
```

3. Testing Methodology

- A process that notices a lack of leader starts an election. It creates an election message containing its UID. It then sends this message clockwise to its neighbour.
- To test this algorithm, I simply execute the following line in main class. We can choose any process to start this election, but eventually always the one that owns the biggest id wins the election.
- Here I choose process 4 to start as example.

```
processes[5].tell(new ElectionMessage(4), ActorRef.noSender());
```

5. Results

- Now present the results of the algorithm execution from part 3. We can see how this election process runs through the output words and data in comsol.

```
[INFO] [02/01/2024 12:37:51.223] [System-akka.actor.default-dispatcher-5] [akka://System/user/process5] Process 5 is sending an election message with ID 5
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-dispatcher-21] [akka://System/user/process6] Process 6 is sending an election message with ID 6
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-dispatcher-5] [akka://System/user/process7] Process 7 is sending an election message with ID 7
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-dispatcher-21] [akka://System/user/process8] Process 8 is sending an election message with ID 8
```

```
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process9] Process 9 is sending
an election message with ID 9
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process10] Process 10 is
sending an election message with ID 10
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process11] Process 11 is
sending an election message with ID 11
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process12] Process 12 is
sending an election message with ID 12
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process13] Process 13 is
sending an election message with ID 13
[INFO] [02/01/2024 12:37:51.224] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process14] Process 14 is
sending an election message with ID 14
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process15] Process 15 is
sending an election message with ID 15
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process16] Process 16 is
sending an election message with ID 16
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process17] Process 17 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process0] Process 0 is sending
an election message with ID 17
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process1] Process 1 is sending
an election message with ID 17
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process2] Process 2 is sending
an election message with ID 17
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process3] Process 3 is sending
an election message with ID 17
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process4] Process 4 is sending
an election message with ID 17
[INFO] [02/01/2024 12:37:51.225] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process5] Process 5 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process6] Process 6 is sending
an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process7] Process 7 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process8] Process 8 is sending
an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process9] Process 9 is
```

```
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process10] Process 10 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process11] Process 11 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process12] Process 12 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process13] Process 13 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process14] Process 14 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.226] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process15] Process 15 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process16] Process 16 is
sending an election message with ID 17
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process17] Process 17 starts
acting as the leader
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process17] Process 17 is
sending an elected message with ID 17
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process0] Process 0 records
17 as the leader
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process1] Process 1 records 17
as the leader
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process2] Process 2 records
17 as the leader
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process3] Process 3 records 17
as the leader
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process4] Process 4 records
17 as the leader
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process5] Process 5 records 17
as the leader
[INFO] [02/01/2024 12:37:51.227] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process6] Process 6 records
17 as the leader
[INFO] [02/01/2024 12:37:51.228] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process7] Process 7 records 17
as the leader
[INFO] [02/01/2024 12:37:51.228] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process8] Process 8 records
17 as the leader
[INFO] [02/01/2024 12:37:51.228] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process9] Process 9 records 17
as the leader
[INFO] [02/01/2024 12:37:51.228] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process10] Process 10 records
17 as the leader
[INFO] [02/01/2024 12:37:51.228] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process11] Process 11 records
17 as the leader
[INFO] [02/01/2024 12:37:51.229] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process12] Process 12 records
17 as the leader
[INFO] [02/01/2024 12:37:51.229] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process13] Process 13 records
17 as the leader
[INFO] [02/01/2024 12:37:51.229] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process14] Process 14 records
17 as the leader
[INFO] [02/01/2024 12:37:51.229] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process15] Process 15 records
17 as the leader
[INFO] [02/01/2024 12:37:51.229] [System-akka.actor.default-
dispatcher-21] [akka://System/user/process16] Process 16 records
17 as the leader
[INFO] [02/01/2024 12:37:51.229] [System-akka.actor.default-
dispatcher-5] [akka://System/user/process17] Process 17 received
its own elected message. The election is complete.
```

6. Conclusion

- In distributed systems, there is often a need for a coordinating node (leader) to manage resources, control decision-making processes, or coordinate activities. This algorithm provides an efficient method to elect such a leader.
- When there's a single process starting the election, the algorithm requires 3N-1 sequential messages, in the worst case. Worst case is when the process starting the election is the immediate following to the one with greatest UID: it takes N-1 messages for the election message to reach it, then N messages for it to get back its own UID, then other N messages to send everyone in the ring the elected message.
- The leader election algorithm allows for the re-election of a new leader in case of failure of the original leader or dynamic changes in the system, thereby enhancing the system's fault tolerance and robustness. However, the fault tolerance can still be increased if every process knows the whole topology, by introducing ACK messages and skipping faulty nodes on sending messages.
- The algorithm allows for leader election without central control, which is crucial for decentralized distributed systems as it reduces the risk of a single point of failure.

- The design of the Chang and Roberts algorithm is both simple and effective, applicable to any number of processes, and does not require each process to be aware of how many other processes are in the ring.

7. References

- Chang, E., & Roberts, R. (1979). An improved algorithm for decentralized extrema-finding in circular configurations of processes. Communications of the ACM, 22(5), 281–283. https://doi.org/10.1145/359104.359108
- "Chang and Roberts algorithm." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, date last visited. Web. Date last accessed. https://en.wikipedia.org/wiki/Chang_and_Roberts_algorithm.

Appendix

- Source code

```
package demo;

import akka.actor.ActorRef;
import akka.actor.ActorSystem;

public class SimpleLeaderElectionAlgorithm {
    public static void main(String[] args) {

        // Instantiate an actor system
        final ActorSystem system = ActorSystem.create("System");

        // Assuming there are 18 processes in the ring
        ActorRef[] processes = new ActorRef[18];
        for (int i = 0; i < processes.length; i++) {
            processes[i] = system.actorOf(ProcessActor.createActor(i),
"process" + i);
        }

        // Set up the ring topology - reference to the next process
        for (int i = 0; i < processes.length; i++) {
            ActorRef nextProcess = processes[(i + 1) %
processes.length];
            processes[i].tell(new NextRef(nextProcess),
ActorRef.noSender());
        }

        // We can choose any process to start the election
        // Here process 4 starts the election by sending election
message to its neighbor process 5
        processes[5].tell(new ElectionMessage(4), ActorRef.noSender());

        try {
                        waitBeforeTerminate();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                } finally {
                        system.terminate();
                }
        }

        public static void waitBeforeTerminate() throws
InterruptedException {
                Thread.sleep(5000);
        }
}
package demo;

import akka.actor.ActorRef;
import akka.actor.Props;
import akka.actor.UntypedAbstractActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;

public class ProcessActor extends UntypedAbstractActor {

    // Logger attached to actor
    private final LoggingAdapter log =
Logging.getLogger(getContext().getSystem(), this);

    private final int id;
    private boolean participating = false; // Initially each process in
the ring is marked as non-participant.

    // Actor reference
    private ActorRef next;
    private int electedId;


    public ProcessActor(int id) {
        this.id = id;
    }

    static public Props createActor(int id) {
        return Props.create(ProcessActor.class, () -> new
ProcessActor(id));
    }

    @Override
        public void onReceive(Object message) throws Throwable {
        if (message instanceof NextRef) {
            this.next = ((NextRef) message).getNext(); // References to
the next process
        }
        if (message instanceof ElectionMessage) {
            ElectionMessage electionMessage = (ElectionMessage)
message;
            handleElectionMessage(electionMessage);
        }
        if (message instanceof ElectedMessage) {
            ElectedMessage electedMessage = (ElectedMessage) message;
            handleElectedMessage(electedMessage);
        }
        }

    private void handleElectionMessage(ElectionMessage message) {
        int incomingId = ((ElectionMessage) message).getId();
        if (incomingId > this.id) {
            participating = true;
            sendElectionMessage(incomingId); // Forward the election
message with the same incoming ID
        }
        if (incomingId < this.id) {
            if(!participating) {
                participating = true; // Mark self as participant
                sendElectionMessage(this.id); // Send an new election
message with own ID
            } else {
                log.info("Process {} discards the election message",
this.id);
            }
        }
        if (incomingId == this.id) {
            log.info("Process {} starts acting as the leader",
this.id);
            participating = false; // The leader process marks itself
as non-participant
            this.electedId = this.id; // Records the elected UID
            sendElectedMessage(this.id);
        }
    }

    private void handleElectedMessage(ElectedMessage message) {
        int electedId = ((ElectedMessage) message).getId();
        if (electedId != this.id) {
            participating = false;
            this.electedId = electedId; // Records the elected UID
            log.info("Process {} records {} as the leader", this.id,
electedId);
            next.tell(message, getSelf()); // Forwards the elected
message unchanged
        } else {
            log.info("Process {} received its own elected message. The
election is complete.", this.id);
        }
    }

    private void sendElectionMessage(int id) {
        log.info("Process {} is sending an election message with ID
{}", this.id, id);
        next.tell(new ElectionMessage(id), getSelf());
    }

    private void sendElectedMessage(int id) {
        log.info("Process {} is sending an elected message with ID {}",
this.id, id);
        next.tell(new ElectedMessage(id), getSelf());
    }
}
package demo;

public class ElectionMessage {
    private final int id;

    public ElectionMessage(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
package demo;

public class ElectedMessage {
    private final int id;

    public ElectedMessage(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
package demo;
import akka.actor.ActorRef;

public class NextRef {
    private final ActorRef next;

    public NextRef(ActorRef next) {
        this.next = next;
    }

    public ActorRef getNext() {
        return next;
    }
}
```