# University of South Bohemia České Budějovice

Course

UAI/503 – Distributed Algorithm

Semester Project

Implementation of Hirschberg- Sinclair Algorithm

Group Names

Tushar Rewatkar Personal No.: B21462

Aditya Yadav Personal No.: B21464

# Contents

# Introduction

The Hirschberg–Sinclair algorithm is a distributed algorithm designed for leader election problem in a synchronous ring network. It is named after its inventors, Dan Hirschberg and J. B. Sinclair. It is a randomized algorithm that uses a two-phase process to select a leader. In the first phase, each processor sends a message to a nearest neighbour, asking if it is the leader. If the neighbour is not the leader, it sends the message to another nearest neighbour, and the process repeats until the leader is found. In the second phase, the processor that first receives the message becomes the leader, and broadcasts this information to all other processors in the network.

## Algorithm

The working of Hirschberg-Sinclair algorithm could be understood by the following steps:

1. Each process sends a request message to its neighbour processes, asking to be considered as a candidate for the leader.
2. If a process receives a request message, it checks whether it is the leader. If it is, it sends a response message back to the sender. If not, it repeats the process by forwarding the request message to its own neighbour processes.
3. When a process receives a response message, it sets itself as the leader and broadcasts the leader information to all its neighbour processes.
4. Each process updates its leader information when it receives a broadcast message.

This process continues indefinitely, with the processes sending request messages to their neighbours and the leader being elected through a series of request and response messages. If a new leader is elected, the leader broadcasts the leader information to all its neighbours, and the processes update their leader information accordingly.

## Working

To implement Hirschberg and Sinclair's algorithm in a distributed programming environment, you would need to design a system for communicating messages between processors and for handling the leader election process.

Here are the general steps you could follow:

Define a message type for the leader election process, with fields for the sender and receiver IDs, a message type (e.g., request, response, broadcast), and any other necessary data.

Implement a function for sending a message to a nearest neighbour. This function should take a message and a list of neighbours as input, and send the message to one of the neighbours chosen at random.

Implement a function for handling incoming messages. This function should determine the type of message and take the appropriate action, such as sending the message to another neighbour (if it is a request), becoming the leader and broadcasting the result (if it is a response), or updating the leader information (if it is a broadcast).

In the main program loop, have each processor send a request message to a nearest neighbour, and handle incoming messages as they arrive.

When a processor becomes the leader, it should broadcast the result to all other processors in the network.

# Implementation

To implement the Hirschberg–Sinclair (HS) algorithm using Java RMI (Remote Method Invocation), we need to have the following.

1. The *interface* for the RMI service that will be used to implement the HS algorithm. This interface should define the methods that will be used to perform the computation and return the results.
2. The *implementation* using the RMI service as a Java class. This class should implement the interface defined in the interface and provide an implementation for the methods defined in the interface.
3. Define a main class that will be used to start the RMI service and invoke the methods defined in the RMI service class.
4. Use the *java.rmi.registry.LocateRegistry* class to create a registry on the host machine and bind the RMI service to the registry.
5. Create an instance of the main class and use the *java.rmi.Naming* class to bind the instance to a name in the registry.
6. Use the *java.rmi.registry.LocateRegistry* class to lookup the RMI service in the registry from a client machine, and use the *java.rmi.server.UnicastRemoteObject.exportObject* method to export the RMI service to the client.
7. On the client side, use the *java.rmi.Naming* class to look up the RMI service and invoke its methods as needed.

This is a general outline of how the HS algorithm could be implemented using Java RMI.

Below is the detailed description of the Implementation.

**HS.java**

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HS extends Remote {

    int getId() throws RemoteException;
    void startElection() throws RemoteException;
    void receiveElection(HSNodeImpl requestingNode) throws RemoteException;
    void becomeCoordinator(HSNodeImpl coordinator) throws RemoteException;
    boolean isLeader() throws RemoteException;
    void showMessage() throws RemoteException;
    void receiveCoordinator(HSNodeImpl coordinator) throws RemoteException;
}
```

*Figure 1: HS.java*

This is the interface that defines a set of methods that will be used to perform Hirschberg-Sinclair Algorithm for leader election. The interface extends the Remote interface, which is a marker interface that indicates that an object can be used over a network.

The interface defines the following methods:

*getId():* returns the ID of the node.
*startElection():* starts the election process.
*receiveElection(HSNodeImpl requestingNode):* receives an election request from another node.
*becomeCoordinator(HSNodeImpl coordinator):* becomes the coordinator of the distributed system.
*isLeader():* returns a Boolean value indicating whether the node is the leader or not.
*showMessage():* displays a message.
*receiveCoordinator(HSNodeImpl coordinator):* receives the coordinator from another node.

**HSNodeImpl.java**

Following is the implementation of for the function define in the interface.

```
// Implementation of HS algorithm for leader election
public void startElection() throws RemoteException {
    // Check if the right neighbor has a higher ID than the current node
    if (right.getId() > id) {
        // Pass on the request to start the election to the next node in the ring
        right.startElection();
    } else {
        // The current node has the highest ID, so start the election by sending a request to the left neighbor
        left.receiveElection(this);
    }
}
```

*Figure 2: startElection() method from HSNodeImpl.java*

The *startElection()* method is a part of the implementation of the HS algorithm for leader election in a distributed system. The method is called to initiate the election process to determine the leader of the distributed system.

The method first checks if the right neighbour has a higher ID than the current node. If it does, it passes on the request to start the election to the right neighbour by calling the *startElection()* method on it. If the right neighbour does not have a higher ID, it means that the current node has the highest ID. In this case, the method starts the election by sending a request to the left neighbour by calling the *receiveElection(HSNodeImpl requestingNode)* method on it, passing itself as an argument.

```
public void receiveElection(HSNodeImpl requestingNode) throws RemoteException {
    int requestingNodeId = requestingNode.getId();
    if (requestingNodeId > id) {
        if (right != null) {
            right.receiveElection(requestingNode);
        } else {
            becomeCoordinator(requestingNode);
        }
    } else if (requestingNodeId < id) {
        if (left != null) {
            left.receiveElection(requestingNode);
        } else {
            becomeCoordinator(requestingNode);
        }
    } else {
        becomeCoordinator(requestingNode);
    }
}
```

*Figure 3: receiveElection() method from HSNodeImpl.java*

The method takes an HSNodeImpl object as an argument, representing the node that is sending the request. It first retrieves the ID of the requesting node and compares it to the ID of the current node. If the requesting node's ID is higher, it checks if the current node has a right neighbour. If it does, it passes on the election request to the right neighbour by calling the *receiveElection(HSNodeImpl requestingNode)* method on it. If the current node does not have a right neighbour, it means that the requesting node has the highest ID and the current node becomes the coordinator by calling the *becomeCoordinator(HSNodeImpl requestingNode)* method.

If the requesting node's ID is lower, the method checks if the current node has a left neighbour and passes on the election request in a similar manner. If the requesting node's ID is equal to the current node's ID, the current node becomes the coordinator.

```
public void receiveCoordinator(HSNodeImpl requestingNode) throws RemoteException {
    if (requestingNode.getId() != id) {
        right.receiveCoordinator(requestingNode);
    }
}
public void becomeCoordinator(HSNodeImpl requestingNode) throws RemoteException {
    isLeader = true;
    if(left!=null){
    left.receiveCoordinator(requestingNode);
}
if(right!=null){
    right.receiveCoordinator(requestingNode);
}
}
public boolean isLeader() throws RemoteException {
    return isLeader;
}
```

*Figure 4: other methods from HSNodeImpl.java*

The *receiveCoordinator(HSNodeImpl requestingNode)* method is a part of the implementation of the HS algorithm for leader election in a distributed system. The method is called when a node receives the coordinator from another node as part of the election process to determine the leader of the distributed system.

The *becomeCoordinator(HSNodeImpl requestingNode)* method is called when a node becomes the coordinator of the distributed system. It sets the *isLeader* variable to true to indicate that the node is the leader. It then sends the coordinator to the left and right neighbours, if they exist, by calling the *receiveCoordinator(HSNodeImpl requestingNode*) method on them.

The *isLeader()* method returns a Boolean value indicating whether the node is the leader or not. It returns the value of the *isLeader* variable.

The *HSNodeImpl* class implements the HS interface and provides an implementation for these methods. The election process is initiated by calling the *startElection()* method on a node, which sends an election request to its left neighbor. The request is passed along the distributed system until it reaches the node with the highest ID, which becomes the leader and sends the coordinator to all the other nodes in the system. The *isLeader()* method can be called on a node to determine whether it is the leader or not.

**HSServer.java**

```java
public class HSServer {
    Run | Debug
    public static void main(String[] args) throws RemoteException, AlreadyBoundException {
        // Read the number of nodes from the command line
        int numNodes = Integer.parseInt(args[0]);
        // Create the first node in the ring
        HSNodeImpl firstNode = new HSNodeImpl(id: 1, left: null, right: null);
        HSNodeImpl currentNode = firstNode;
        // Create the remaining nodes in the ring and add them to the ring structure
        for (int i = 2; i <= numNodes; i++) {
            HSNodeImpl nextNode = new HSNodeImpl(i, currentNode, right: null);
            currentNode.right = nextNode;
            currentNode = nextNode;
        }
        // Set the right field of the last node to point back to the first node, completing the ring
        currentNode.right = firstNode;
        // Start the RMI registry
        Registry registry = LocateRegistry.createRegistry(port: 1099);
        // Bind the nodes to the registry
        for (int i = 1; i <= numNodes; i++) {
            registry.bind("Node" + i, currentNode);
            currentNode = (HSNodeImpl) currentNode.right;
            System.out.println("Node " + i + " with id: " + i );
        }
        // Start the election
        firstNode.startElection();
        System.out.println(x: "Server is ready");
    }

}
```

*Figure 5: HSServer class*

This *HSServer* class is a Java program that sets up a distributed system using Java Remote Method Invocation (RMI) and initiates the leader election process using the HS algorithm.

The *main()* method reads the number of nodes in the distributed system from the command line and creates the nodes using the *HSNodeImpl* class. It creates a ring structure by setting the right field of each node to point to the next node, and the right field of the last node to point back to the first node. It then starts the RMI registry and binds the nodes to it using the Registry interface, allowing them to be referred to by a name and allowing remote objects to invoke methods on them. Finally, it initiates the election process by calling the *startElection()* method on the first node in the ring. This method sends an election request to the left neighbour and the request is passed along the ring until it reaches the node with the highest ID, which becomes the leader and sends the coordinator to all the other nodes in the system.

**HSClient.java**

```java
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class HSClient {
    Run | Debug
    public static void main(String[] args) throws RemoteException, NotBoundException {
        // Read the number of nodes from the command line
        int numNodes = Integer.parseInt(args[0]);

        // Get the RMI registry
        Registry registry = LocateRegistry.getRegistry(port: 1099);

        // Check if each node is the leader
        for (int i = 1; i <= numNodes; i++) {
            HS node = (HS) registry.lookup("Node" + i);
            if (node.isLeader()) {
                node.showMessage();
                System.out.println("Node with id: " + node.getId() + " is the leader");
            }
        }
    }
}
```

*Figure 6: HSClient class*

The *main()* method reads the number of nodes in the distributed system from the command line and gets the RMI registry using the Registry interface. It then looks up each node in the registry by its name and checks whether it is the leader by calling the *isLeader()* method. If a node is the leader, it calls the *showMessage()* method to display a message indicating that it is the leader. The *isLeader()* method returns a Boolean value indicating whether the node is the leader or not. The *showMessage()* method displays a message on the server indicating that the node is the leader.

This implementation allows nodes in a distributed system to communicate with each other and participate in an election process to determine the leader node. The *HS* interface defines the methods for a distributed system, the *HSNodeImpl* class provides an implementation of the HS algorithm, the *HSServer* class sets up the distributed system and initiates the election process, and the *HSClient* class connects to the distributed system and checks which node is the leader.

## UML Class Diagram

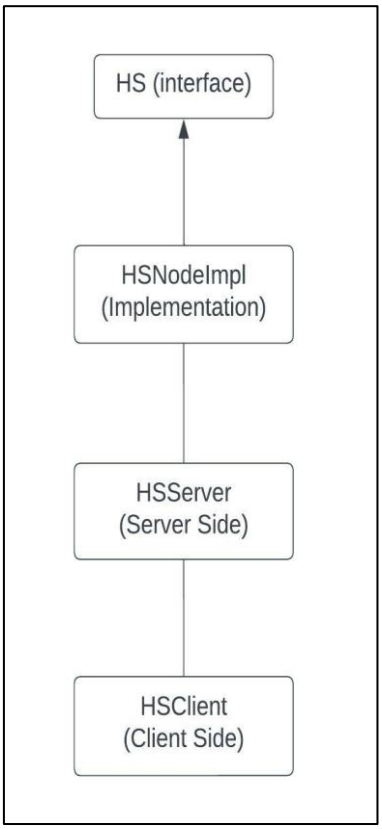


*Figure 7: Class diagram*

This is a class diagram for a distributed system using Hirschberg-Sinclair ring-based leader election algorithm. The *HS* class represents a node in the system, and has several methods for participating in the leader election process, as well as a method to check if the node is the leader and display a message. The *HSNodeImpl* class is a concrete implementation of *HS*, with additional instance variables to represent the node's identity and its neighbours in the ring. The *HSServer* and *HSClient* classes are the main entry points for the server and client programs, respectively.

# Observations

**How the project works**

To run the project, user must run the server file i.e., *HSServer.java* which will create a ring topology with the given number of nodes. After this, running the client file i.e., *HSClient.java* which invokes the methods to start the election for choosing the leader will present the output with a message on both Client and Server to announce the leader.

**Output**

```
Node 1 with id: 1
Node 2 with id: 2
Node 3 with id: 3
Node 4 with id: 4
Node 5 with id: 5
Server is ready
```

*Output 1: Server side*

This is the output that will be generated by the *HSServer.java* file, number of nodes is determined by the input.

After running the *HSClient.java* file (for the number of nodes: 5), both server and client terminal will be presented with the following message.

```
Node with id: 5 is the leader
```

This is the announcement that the node with the respective id is chosen as the leader.

**Conclusion**

To conclude, the Hirschberg-Sinclair (HS) algorithm is used to implement leader election in a distributed system. The algorithm works by having each node send an election request to its right neighbour if its own ID is greater than that of its

right neighbour. If the right neighbour's ID is greater, it passes the request on to its own right neighbour. This process continues until the request reaches a node whose ID is the highest in the ring. This node becomes the coordinator and sends a message to all other nodes in the ring to inform them of its status.

Overall, the HS algorithm is an effective and efficient way to implement leader election in a distributed system and can be easily extended and customized for different applications.

# References


[1] https://en.wikipedia.org/wiki/Hirschberg%E2%80%93Sinclair_algorithm
Wikipedia page of HS algorithm, for the introduction

[2] http://comet.lehman.cuny.edu/griffeth/classes/Spring05/Lectures/0217.pdf
Paper on the Leader Election and its complicity

[3] https://cse.iitkgp.ac.in/~pallab/dist_sys/Lec-08-LeaderElection.pdf
Different Algorithm that can be used for Leader Election in a ring topology and their efficiency

[4] http://pages.cpsc.ucalgary.ca/~woelfel/paper/leaderelection/le.pdf
Ring leader election in Distributed systems