**Hirschberg-Sinclair Algorithm**

UAI/503 Distributed Algorithm

Group Members

Hadeera Munsif   
Ammar Zafar

University of South Bohemia

Contents

[Introduction 2](#_Toc123763789)

[Working 2](#_Toc123763790)

[Method of Implementation 3](#_Toc123763791)

[Implementation 3](#_Toc123763792)

[Server 3](#_Toc123763793)

[ServerINT.java 3](#_Toc123763794)

[Leader.java 4](#_Toc123763795)

[Indicator.java 7](#_Toc123763796)

[Controller.java 8](#_Toc123763797)

[Client 10](#_Toc123763798)

[ClientINT.java 10](#_Toc123763799)

[Neighbours.java 11](#_Toc123763800)

[NodeImpl.java 12](#_Toc123763801)

[Receive.java 13](#_Toc123763802)

[UML Diagram 14](#_Toc123763803)

[Server UML 14](#_Toc123763804)

[Client UML 15](#_Toc123763805)

[Conclusion 16](#_Toc123763806)

[Impact 16](#_Toc123763807)

# 

# Introduction

The Hirschberg–Sinclair algorithm for leader election in a synchronous ring network is significant because it is a distributed algorithm that allows a group of nodes in a network to elect a leader without the need for a central coordinator. This can be useful in situations where a central coordinator is not available or is unreliable, as it allows the nodes to self-organize and reach a consensus on the leader.

1. In addition, the Hirschberg–Sinclair algorithm has a number of other important properties that make it a useful tool for leader election in distributed systems:
2. It is a linear-time algorithm, meaning that its time complexity is O(n), where n is the number of nodes in the network. This makes it much faster than other algorithms with higher time complexity, such as the dynamic programming approach, which has a time complexity of O(n^2).
3. It is a simple and easy-to-implement algorithm, making it a good choice for applications where simplicity is a priority.
4. It is a fault-tolerant algorithm, meaning that it can handle failures and errors in the network without affecting the leader election process.

Overall, the Hirschberg–Sinclair algorithm is a useful tool for leader election in distributed systems and has been widely used in a variety of applications.

## Working

The Hirschberg–Sinclair algorithm can be used for leader election in a synchronous ring network.

The steps for the Hirschberg–Sinclair algorithm for a leader election in a synchronous ring network are:

1. Each node in the network is assigned a unique identifier (ID).
2. The nodes are organized into a ring, with each node connected to two neighbors.
3. The nodes begin the leader election process by sending their IDs to their right neighbor.
4. When a node receives an ID from its left neighbor, it compares the ID to its own. If the received ID is greater, the node passes the received ID onto its right neighbor and continues to listen for IDs. If the received ID is less than or equal to the node's own ID, the node assumes the role of leader and sends a "stop" message to its neighbors.
5. The process continues until a "stop" message is received by a node, at which point the leader election process is complete and the node with the highest ID is elected as the leader.

# Method of Implementation

In order to execute this program, first install the RMI plugin for Eclipse at <http://www.genady.net/rmi/v20/>.

Java Remote Method Invocation (RMI) is a Java API that allows a Java program to invoke methods on an object that is running on a different Java Virtual Machine (JVM). In order to use RMI, a programmer needs to set up a special environment and configure the compiler.

We downloaded a special tool called a "RMI plug-in for eclipse." This tool automates the process of setting up the environment and configuring the compiler for RMI, and it can generate a registry (a special server that keeps track of RMI objects) with just a few clicks.

We then created a new Java project and divided it into two packages called "client" and "server." The files in these two packages are able to communicate with each other using RMI.

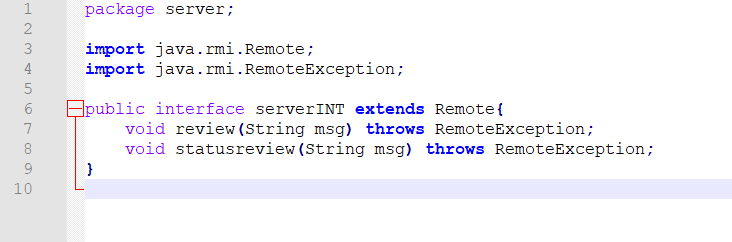
# Implementation

Several classes and java files have been defined for the server and client to run this model.

## Server

### ServerINT.java

Starting with the server. A serverINT file has been defined. The following code is used:



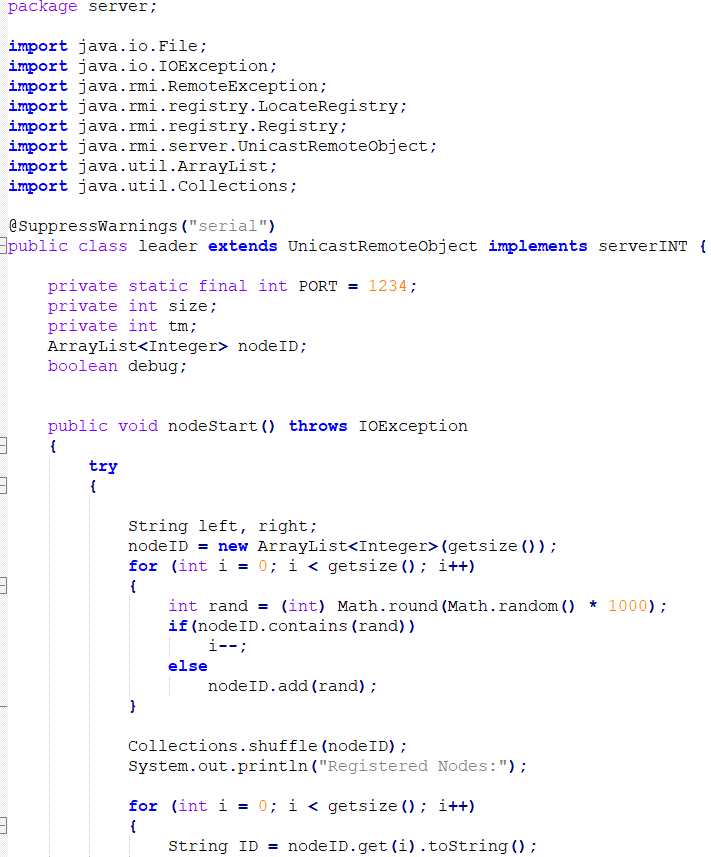
This code defines a Java interface called "serverINT" that extends the "Remote" interface from the Java RMI library. An interface in Java is a set of method declarations that can be implemented by any class.

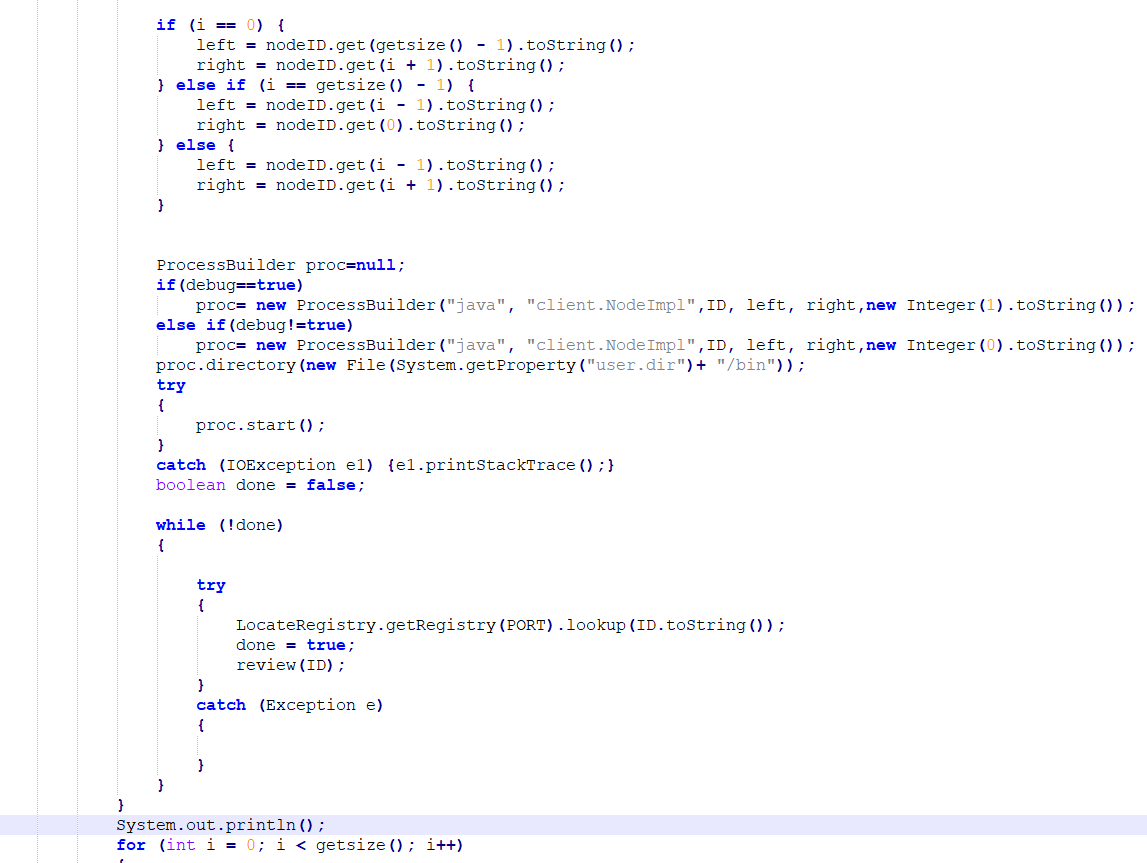
The "serverINT" interface contains two methods: "review" and "statusreview." Both methods are marked with the "throws RemoteException" clause, which indicates that they can throw an exception of type "RemoteException" if there is an error when the method is called remotely over RMI.

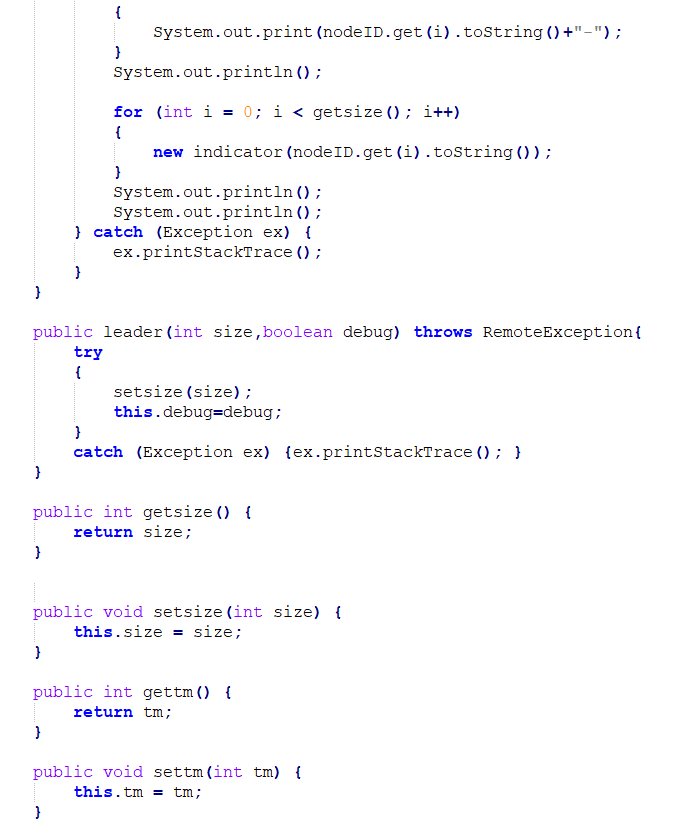
The "review" method takes a single parameter of type "String," which is a message to be reviewed. The "statusreview" method also takes a single parameter of type "String," which is a message containing the status of a review.

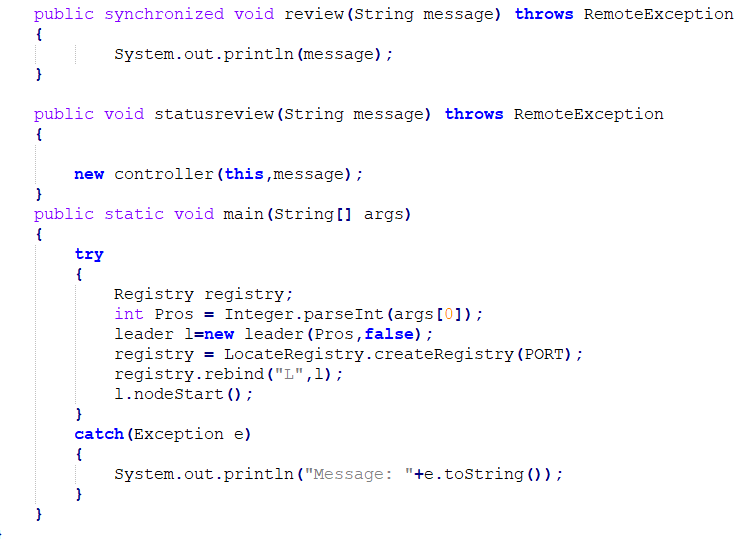
This interface will likely be implemented by a server class, which will define the behavior of these methods. The server class will be able to receive messages and review them, as well as send back messages containing the status of the review.

### Leader.java

****

****

****

****

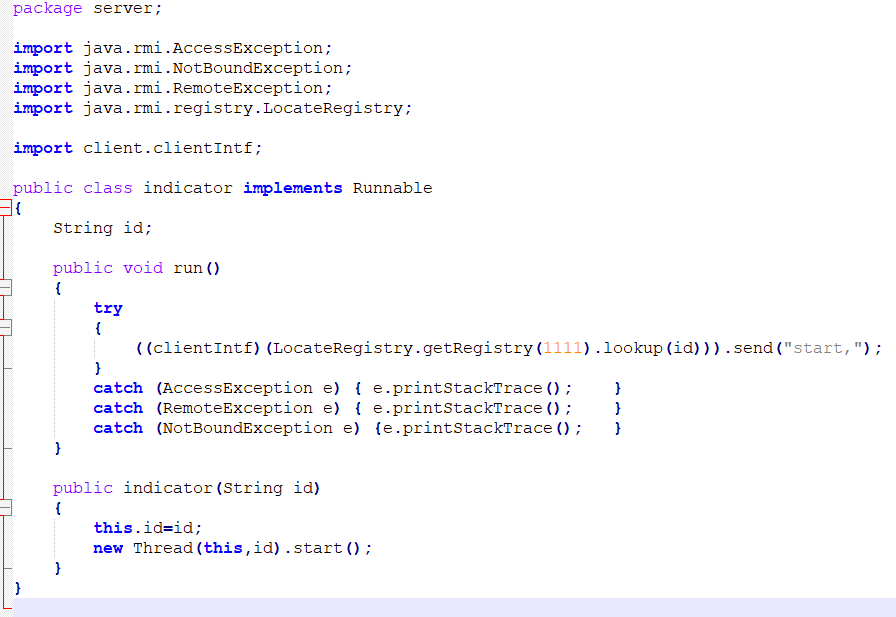
This code is the implementation of the Java class called "leader." The class extends the "UnicastRemoteObject" class, which allows it to be used in Java RMI as a remote object. The class also implements the "serverINT" interface.

The "leader" class contains a number of instance variables, including "size," "tm," "nodeID," and "debug." It also contains a method called "nodeStart," which is the main method of the class.

The "nodeStart" method performs several tasks:

1. It creates an array list called "nodeID" and fills it with a certain number of random integers.
2. It shuffles the elements in the "nodeID" array list.
3. It creates a new process for each element in the "nodeID" array list, using the "ProcessBuilder" class. Each process is a separate Java program that runs in its own JVM.
4. It waits for each process to start and be registered with a registry server.
5. It prints the IDs of the registered processes to the console.
6. It starts a new thread for each process, using the "indicator" class.
7. It waits for the threads to complete.
8. It prints the final status of the leader election process to the console.

### Indicator.java

**

The class implements the "Runnable" interface, which means that it contains a "run" method that can be executed in a separate thread.

The "run" method of the "indicator" class performs the following tasks:

1. It looks up a remote object in the RMI registry with a given ID.
2. It invokes a method on the remote object called "send," passing it a string parameter.

The "indicator" class also contains a constructor method called "indicator," which takes a single parameter of type "String" representing the ID of the remote object. The constructor creates a new thread and starts it, passing the "run" method as a target. The "indicator" class is used to start a new thread for each node in the system, which sends a message to the node using RMI.

### Controller.java



This code is the implementation of the Java class called "controller."

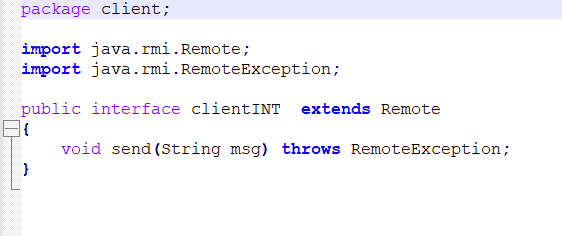
The "run" method of the "controller" class performs the following tasks:

1. It splits a string message into an array of strings using the "split" method.
2. It checks the first element of the array to see if it is the string "status."
3. If the first element is "status," the method performs the following tasks:
   * It parses the third element of the array as an integer and adds it to the "tm" (time measure) field of the "lead" object.
   * It calls the "review" method of the "lead" object, passing it a concatenated string consisting of the second element of the array and the string “chooses " and the third element of the array.
4. If there is an exception thrown at any point during the execution of the method, the method catches the exception and calls the "review" method of the "lead" object, passing it the string representation of the exception.

The "controller" class also contains a constructor method called "controller," which takes two parameters: a "leader" object and a string message. The constructor creates a new thread and starts it, passing the "run" method as a target.

## Client

### ClientINT.java

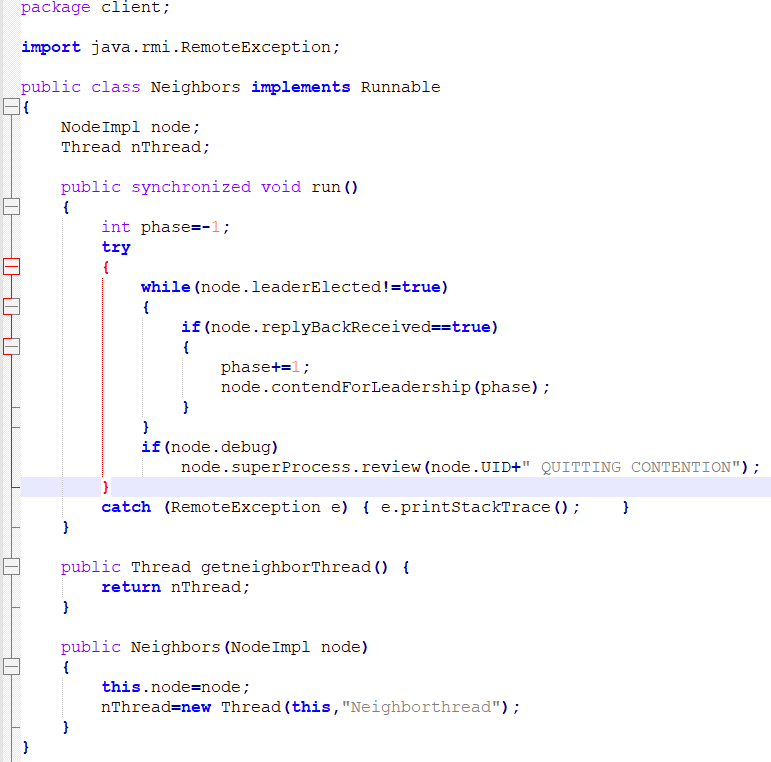


This code defines the Java interface "clientINT" that extends the "Remote" interface from the Java RMI library. An interface in Java is a set of method declarations that can be implemented by any class.

The "clientINT" interface contains a single method called "send," which takes a single parameter of type "String" and returns nothing. The method is marked with the "throws RemoteException" clause, so that it can throw an exception of type "RemoteException" if there is an error when the method is called remotely over RMI.

This interface is implemented by the client class, which defines the behavior of the "send" method.

### Neighbours.java



The "run" method of the "Neighbors" class performs the following tasks:

1. It enters an infinite loop that continues until the "leaderElected" field of the "node" object is "true."
2. Within the loop, it checks the "replyBackReceived" field of the "node" object. If it is "true," the method increments the "phase" variable by 1 and calls the "contendForLeadership" method of the "node" object, passing it the "phase" variable.
3. If the "debug" field of the "node" object is "true," the method calls the "review" method of the "superProcess" field of the "node" object, passing it a concatenated string consisting of the "UID" field of the "node" object and the string "QUITTING CONTENTION."

The "Neighbors" class also contains a method called "getneighborThread" that returns the "nThread" field of the class, which is a "Thread" object. It also contains a constructor method called "Neighbors," which takes a single parameter of type "NodeImpl" representing the "node" object. The constructor creates a new thread and starts it, passing the "run" method as a target.

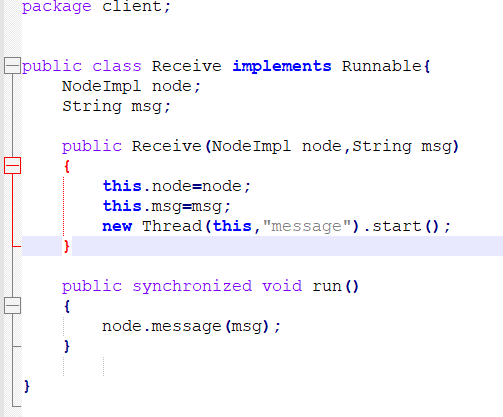
### NodeImpl.java

This class implements the clientINT interface, which allows it to act as a remote object that can receive messages from other nodes.

The NodeImpl class has a number of variables that store information about the node, such as its unique identifier (UID), its left and right neighbors in the ring (left and right), the total number of messages it has sent (totalMsgs), and flags to keep track of various states of the node (contention, leftreplyrecv, rightreplyrecv, leftreaderrcv, rightreaderrcv, Electedleader, debug, and replybackrecv).

The NodeImpl class also has a reference to an instance of the serverINT interface, which represents the remote object that serves as the "supervisor" or "controller" for the leader election algorithm. The NodeImpl class uses this reference to call methods on the supervisor object, such as review to report its status or send messages.

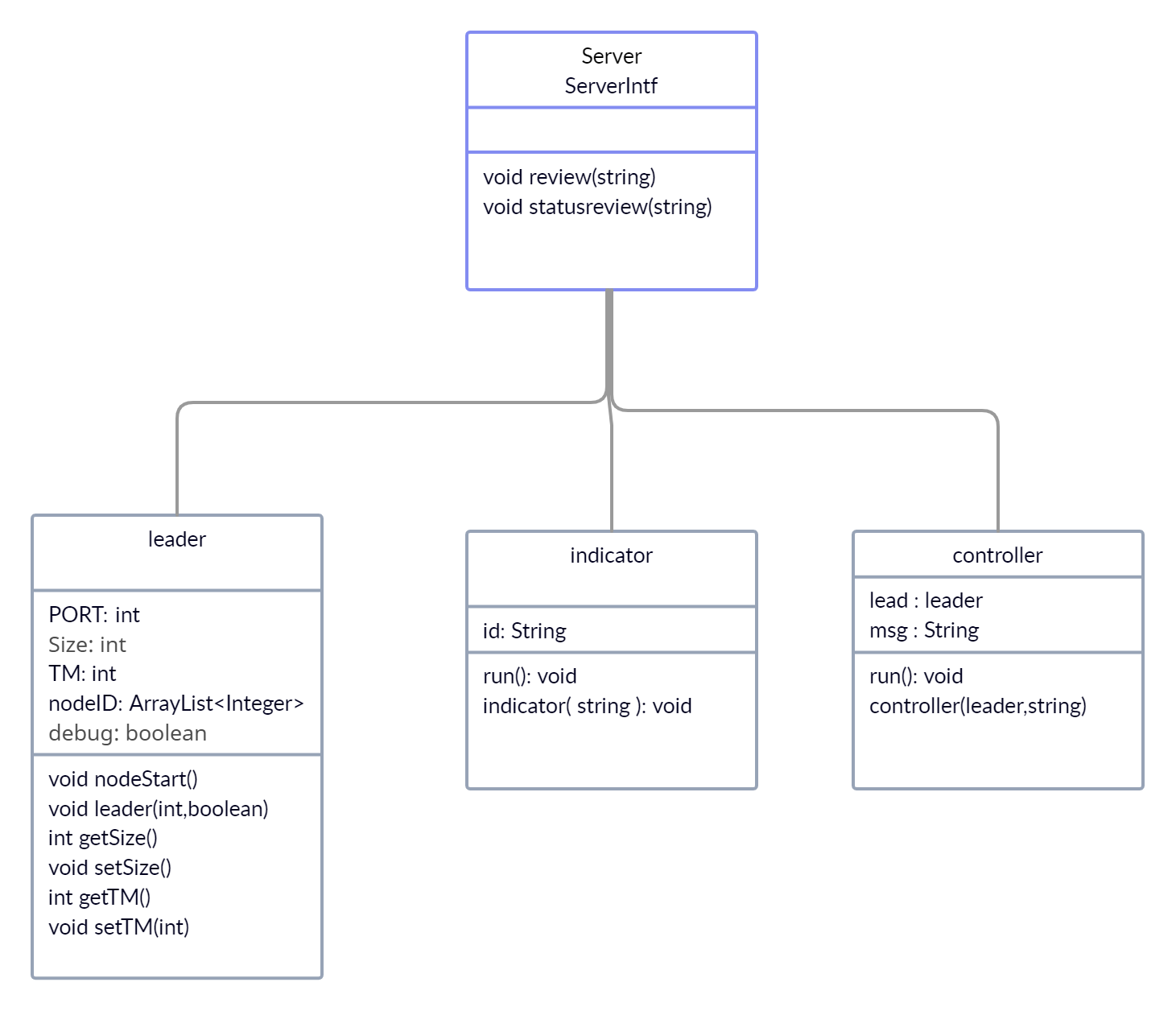
The NodeImpl class has a number of methods that perform different tasks related to the leader election algorithm. The send method is used to receive messages from other nodes and pass them on to the Receive class for further processing. The message method is used to process messages that are received from other nodes. The contendForLeadership method is used to initiate a contention phase for the node, during which it sends messages to its neighbors to determine which node has the highest unique identifier and should be elected leader. The electLeader method is used to elect a leader once the contention phase has been completed. The sendElectedLeader method is used to send a message to the supervisor object to report the elected leader.

Receive.java  


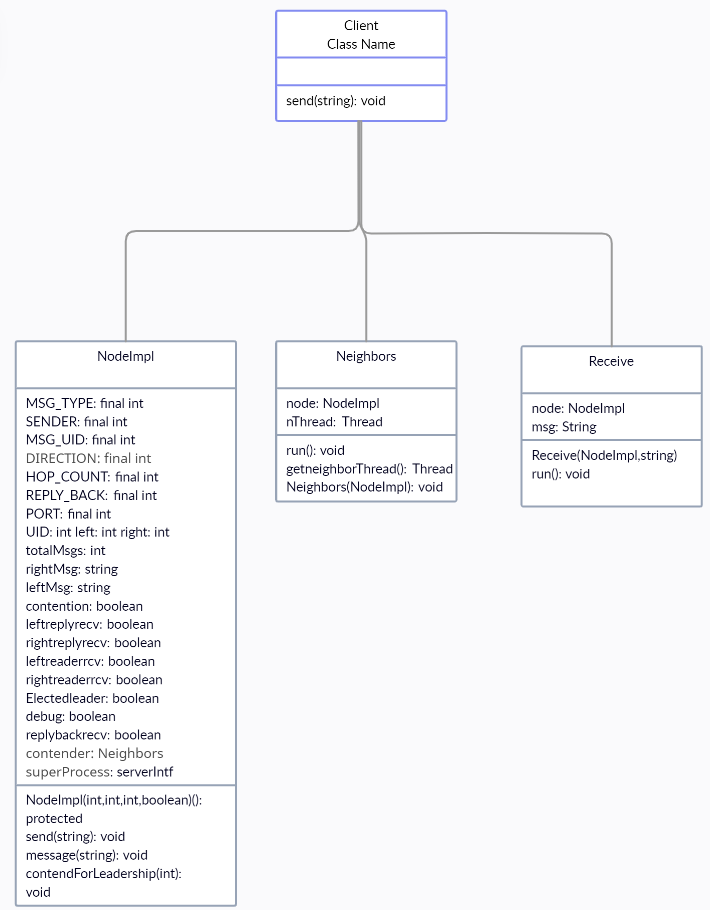
The Receive class allows the NodeImpl class to process incoming messages in a separate thread. When an instance of Receive is created, it is passed a reference to a NodeImpl object and an incoming message. The run method of the Receive class then calls the message method of the NodeImpl object and passes it to the incoming message.

## UML Diagram

### Server UML



Client UML



## Conclusion

In conclusion, the Hirschberg-Sinclair algorithm is a distributed algorithm designed for the leader election problem in a synchronous ring network. We implemented it using Java RMI which allows for the automation of the environment configuration and compilation process through the use of a RMI plugin for Eclipse. The algorithm operates by having nodes send messages to their neighbors and eventually choosing a leader based on the responses received. One of the main benefits of using this algorithm is its simplicity and efficiency, as it requires only a small number of messages to be exchanged in order to elect a leader. Additionally, it has the potential to significantly improve the performance of distributed systems that rely on leader election for their operation.

## Impact

There are several benefits to using the Hirschberg-Sinclair algorithm for leader election in a distributed system:

* Simplicity: The Hirschberg-Sinclair algorithm is relatively simple to understand and implement, which makes it easier to develop and maintain.
* Low overhead: The Hirschberg-Sinclair algorithm has low overhead, which means it requires relatively few resources to run. This can be especially important in resource-constrained systems, such as those found in IoT (Internet of Things) environments.
* Fast convergence: The Hirschberg-Sinclair algorithm has a fast convergence time, which means it can quickly determine the leader of the distributed system. This can be important in systems where it is critical to have a leader in place as soon as possible.
* Fault tolerance: The Hirschberg-Sinclair algorithm is fault tolerant, which means it can continue to operate even if one or more nodes in the distributed system fail. This can improve the reliability and availability of the system as a whole.

Overall, the use of the Hirschberg-Sinclair algorithm for leader election can lead to improved performance, simplicity, and reliability in distributed systems.