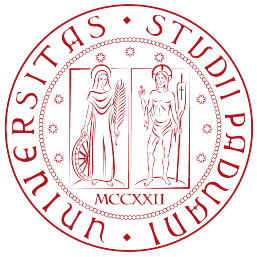


**CONCURRENT AND DISTRIBUTED PROGRAMMING**
**2011-2012**

# Report: *Clone Detector*

Zambon Jacopo
574308

# Contents

# 1 Program Design

## 1.1 Problem Description

The program to develop for the Concurrent and Distributed Programming course is a Java application to simulate a clone attack in a node-communicating network and the relative detection by the use of one of the following two detection protocols for Ad-Hoc Networks: *LSM* and *RED*. The system has to involve a configuration file on a http server, a client simulator and an RMI server which communicates with the client and gets the results of the simulations.

## 1.2 System Overview

The system implements an RMI Server, which task is to provide the client an instance of the Remote Object, useful to get the results of the client simulations, and a client that runs the simulation itself, with the possibility for the user to insert the URL to a configuration file link on a http server and an RMI Server URL.
The way you can start the simulation and the RMI Server is explained in the `readme.txt` file, provided in the submissed folder.

## 1.3 Design Description

In order to describe the project Design, it's useful to take a closer look at the main classes of the system:

- *ProjGUI*: is the GUI of the system, it allows the user to submit a configuration file link and an RMI Server URL, it contains a text area where you can see the result of every simulation and it provides two button, the first one to start the simulation (creating and starting a *Simulation* object) and the second one to stop it.
- *Simulation*: it extends Thread and its tasks are to get the configuration file from the http server and to start the real simulation creating an *Hypervisor* object. The simulation will continue until one of two things happen: the user presses the STOP button or the simulation completes all of the nsim cycles, as described in the configuration file.
- *Hypervisor*: it fills the unit-square area with the specified number of nodes, assigning each of them the relative neighborhood, then it creates from one of these nodes a clone and starts them all. Then it stays in a wait status until the clone node is found or every *Node* thread is terminated or in a waiting status, as checked by the *NoDeamon* Thread. Finally, it starts the connection with the *RMIServer* and send him the elaborated result of every simulation.
- *Node*: it extends Thread, when it starts it sends to all of its neighbors a location claim (*LocationClaim*) message, pushing it in the message buffer of the addressee neighbor. It then stays in a wait status until it receives a message, so it can pop the message from the buffer in order to process it. Some critic aspects of the processing choices will be discussed in the subsection 1.4 and in the section 2.

- *NoDeamon*: this thread controls if every Node is still alive: if a Node is in a waiting or in a terminated status it is considered dead. Once this thread finds out all the nodes thread are dead, it notifies to the hypervisor and stops itself.
- *LocationClaim*: it's the message, containing the ID and the coordinates of the sender node, a boolean flag indicating if it's already forwarded or it is a location claim still to be processed and the destination coordinates, setted when a node decide with the probability indicated in the configuration file to process the message.
- *RMIServer*: it creates an instance of the remote object *txtPrintImpl* and takes care of registering the object with the RMI registry, in order to make it available for the client.
- *txtPrintImpl*: it's the Remote Object, it indeed extends UnicastRemoteObject and implement the interface available to the client (*txtPrint*). When it's called, it creates an `output.txt` file, where it prints the result of every simulation.
- *txtPrint*: the interface of the remote object, used by the client in order to communicate with the RMI Server.

Other classes present in the project:

- *IllegalValueException*: it extends Exception, thrown by Simulation whenever it occurs a problem with a variable or a field of the configuration file on the http server.
- *Coordinate*: it represents the coordinates of a node with the use of two fields of type double (x,y).

## 1.4 Testing and Evaluation

### 1.4.1 Quality Issues

The tests and evaluations for this program quality were especially based upon (1) the need to avoid deadlocks, (2) the efficiency of the program, also when the configuration file sets a huge number of iterations or nodes in the network and (3) the need to avoid useless expenditure of energy by the nodes.
(1) was risolved by the use of the *synchronization*, used to lock the node when it has to send (file `Node.java`, line 105), receive (file `Node.java`, lines 287-299) or save a message (file `Node.java`, line 136, 225, 252: locks obtained by the synchronization of the whole methods).
(2) was risolved by the interrupt() method, called for every node thread: it is implemented in the file `Hypervisor.java`, lines 139-142.
(3) was limited by the use of the static boolean flag foundClone. It allows to check if the clone node was founded before doing anything and spend energy (`Node.java`, lines 103, 113, 120, 148, 191). This is indispensable because it's possible for the program to find the clone while other nodes/threads are processing some messages, so it's important they stop the processing as soon as possible.

### 1.4.2 Test Issues

Some tests were done upon the configuration files provided by the teacher, and the average results are shown in Table 1, where the radius of the nodes and the protocol are changed inside the code for testing purposes, and in Table 2.
What we can observe from this tables is that, assuming the same probability of forwarding a message and increasing the communication radius, *RED* protocol finds the clone node more time than the *LSM* one; however, in the last one we can see a more constant and progressive growth than the *RED* protocol.
Concering to the velocity of the elaboration of a single step simulation, with the configuration files provided by the teacher and regardless of the communication radius of a node or the protocol, all the simulations runned showed an elaborating time for every simulation step much lower than 1 second, as shown in Table 3.

# 2 Main Design and Implementation choices

## 2.1 Configuration File Consistency

To avoid undesiderable values provided by the configuration file, in `Simulation.java` (method tokenize(String line), line 115) is set a control flow, with the following behaviour: if the processed line from the file starts with "%", it's a comment and we have to ignore it; if it starts with "PROTO" we have to control if the value equals to *LSM* or *RED*: every other value is not acceptable and will cause the throws of an IllegalValueException Error. If the line starts with "nsim", "g", "n" or "E" we expects an integer value greater than or equal to 0: if the value it's not an integer, it throws an InputMismatchException, handled in line 231, else if the value is less than 0 an IllegalValueException is thrown. If the line starts with "p" or "r" we expects a float value greater than or equal to 0 (and, in the "p" case, less than 1 too): if the value it's not a float, it throws a NumberFormatException handled in line 227, else if the value is not in the desiderable range an IllegalValueException is thrown. If the line starts with a string or a character different from all of the cases described, the program will throw an IllegalValueException, indicating a wrong configuration file link.

## 2.2 GUI behaviour

The main design choices for the GUI behaviour are:

- to avoid undesiderable behaviours, when the user presses *"Start"*, this button is disabled
- the window stays on foreground even when the processing of the "nsim" simulations end, for further analyisis and studies of the data
- when the "nsim" simulations end or the user presses *"Stop"*, this button is disabled
- when the simulations end or the user presses the Stop button, the Start button is not re-enabled, so if the user wants to run another elaboration, it has to shut the window and start it over again

## 2.3 STOP Button behaviour

When the user presses the Stop button from the GUI, the boolean flag "stop" in `Simulation.java` is setted to false so, even if the number of performed simulations is less than the expected ("nsim" as setted by the configuration file) the simulation ends in a very reasonable time the simulation step it was performing and, thanks to the control (`while(!stop & ...)`) in `Simulation.java`, line 239, stops the client processing.

## 2.4 Broadcast Location Claim message

The broadcast location claim message dispatch from a node to all of its neighbors is guaranteed by the fact it's the first thing a node does when it starts (`Node.java`, lines 267-282) if, of course, it has enough energy to do this.

## 2.5 Energy of a node

### 2.5.1 Sending a message

The energy spent for sending a message is scaled every time the node send a location claim message (`Node.java`, lines 273-280) and when it has to forward a message (`Node.java`, lines 231-236).

### 2.5.2 Receiving a message

The energy spent for receiving a message is scaled when a notify is sent to a waiting (with an empty message buffer) node (`Node.java`, lines 292-298)

### 2.5.3 Signing or Verifying a message

The energy spent for signing a message is scaled once at the creation of the message to broadcast (`Node.java`, lines 269-271); the energy spent for verifying a signature is scaled when the node have to check if the received message is already stored in its hash of saved message (`Node.java`, lines 123-126, 212-215, 239-242)

### 2.5.4 Consistency upon the node energy

Some controls are set upon the remaining energy of the node: before sending, receiving or signing/verifying a message, the system controls if the remaining energy of the node is greater than or equal to the energy it needs to perform the operation. If the node has not enough available energy, the operation is not allowed, but the node thread stays alive 'cause it could be able to perform other activities.

## 2.6 Life cycle of a message

Assuming the conditions of available energy just described, it follows a quickly description of the life cycle of a node, from the moment of its reception: when a node receives a message, it controls if it received a (1) location claim message or an (2) already forwarded one.
(1) In the first case (`Node.java`, method: receiveLC(LocationClaim mess)), with

probability "p" (as described in the configuration file) the node processes the message (`Node.java`, method: forward(LocationClaim mess)): if the protocol is *LSM*, the node clones the message "g" (number of locations) times, changing only the destination coordinates, different for every clone, and actually forwards the created messages; if the protocol is *RED*, the behaviour is very similar to the one just described, but the "g" destination coordinates are chosen using a hash function (in this case, the message digest algorithm is *MD5*) that takes as input the node ID, the randomic value as setted by the Hypervisor (`Hypervisor.java`, lines 111-116) and a counter value (that starts from 0 and it is incremented for each of the "g" points): finally, the "g" messages are forwarded.
When the node has to forward the message (`Node.java`, method: forw(LocationClaim mess)), it's calculated the neigbhor node closer to the destination: if he is the closer to the destination, he checks if the node is already stored and if it is, but the coordinates are different, he found the clone else he just saves the message; if he's not the closer node to the destination he (only with the protocol *LSM*) checks if the message is already present and saves the message or finds the clone: finally, regardless of the protocol, the node sends the message to the neighbor closest to the destination.
(2) When the node receives a forwarded message (`Node.java`, method: receiveLCForw(LocationClaim mess)), if the protocol is *LSM* it has to check if the message is already stored: if it is not, the node has to save it and forward it, else the node has to check if it's the clone (if it's not, forward the message). If the protocol is *RED*, just forward the message. The forwarding follows the same policy as previously described in the second part of point (1).

### 2.6.1 Stored messages/Clone Detection

The messages are stored using a HashMap with key=ID of the message (=node ID) and value=coordinates of the message (=where the node is in the unit-square area), which provides constant-time performance for the basic operations (get and put), without the use of a cycle to scan all the list of saved messages. Infact, for the detection of the clone, the system simply calls the get method on the HashMap of saved messages and checks for an already present key equals to the examined clone ID value.

## 2.7 End of a simulation

A simulation step ("nsim") of the whole model ends when the clone is found or when all the nodes thread are in a waiting or in a terminated status (as checked by the NoDeamon thread), which means they all have no more messages to process.

## 2.8 Server behaviour

The RMI Server prints on a file called `output.txt` the results of every step of the simulation. If more simulations are runned without closing the server (`Ctrl+C` from terminal), the results of the new simulation do not overwrite the results of the previous one, but they're sequentially added at the end of the file. To reset the file, just turn the RMI Server off and on again.

# A Tables

| *Radius* | **LSM** | **RED** |
|---|---|---|
| 0.1 | 1 | 1 |
| 0.2 | 31 | 40 |
| 0.4 | 68 | 90 |
| 0.6 | 86 | 99 |
| 0.8 | 94 | 99 |
| 1 | 95 | 100 |

Table 1: project_config.txt, p=0.1

| | **LSM** | **RED** |
|---|---|---|
| min | 6 | 8 |
| max | 15 | 18 |

Table 2: project_config_2.txt

| **Configuration File** | **Total Time (s)** | **Time, single nsim (s)** |
|---|---|---|
| `project_config` | 5 | 0,05 |
| `project_config_2` | 45 | 0,0225 |

Table 3: Average Timing