Umeå University
Department of Computing Science

Object-Oriented Programming Methodology 7.5 p
5DV133

OU4 Sensor Network

Submitted
2016-05-23

Revision date
2016-06-10

Authors:
Johan Eklund (kv03jed@cs.umu.se)
Tommie Lindberg (c15tlg@cs.umu.se)
Jakob Lundin (c14jln@cs.umu.se)
Lorenz Gerber (dv15lgr@cs.umu.se, lozger03@student.umu.se)

Instructors:
Anders Broberg
Niklas Fries
Adam Dahlgren
Jonathan Westin
Erik Moström
Alexander Sutherland

Contents

1 Introduction

In predefined groups of students we were asked to create a simulation of a network consisting of a number of *nodes* with a short communication range [2]. The simulation consits of time steps, for every time step the network goes through the network and node and makes sure it does its job.

Using the *rumour routing* algorithm [1] you start with a network of *nodes*, in the network there are *messages*, signals, that travel the network to collect and to share information. These messages have a set amount of steps they can make before they vanish. The nodes detect unique *events* and *queries* can be made in any node for that event, the network will then send the *query* to find the *node* that detected the *event*.

Communication is done by using the rumour-routing algorithm [1]. An event in the simulation is an ID, a time stamp and a position of where it happened. Nodes knows its neighbors and they are static in the grid. Nodes can have a maximum of 8 neighbors depending on location. Nodes have have a table of events and know how far away every other node is. Nodes generate messages in the form of agents and queries. Queries should come back to the node within 45 steps if not the node should send another one. Queries are messages that carries the information of an event it's trying to find and a stack of nodes it has traveled through. The node the query passes through checks it's table to see if it know about the event, if it finds it it will send the query on its way to the event node. If it doesn't find it, it will send it to a random neighbor. If a query On the group chat, I felt pretty alone until the weekend before submission deadline while I tried to document and comment what I am currently working on.finds its target node it will get sent back to its original node the same way it got to the event so it can report about it.

In our simulation a query will vanish if it can't find the event in 45 steps. Agents are messages that carry a table of events and the path to these events. The agents travel through the network and synchronizes with the nodes it passes. The events with the shortest distance gets saved in both the nodes and the agents tables. In our simulation the agents will vanish in 50 steps. In our simulation agents have a 50% chance to be generated when an event is detected. In our simulation the network is 50 by 50 and runs for 10000 time steps. Every 400 time steps four randomly chosen nodes are supposed to generate queries.

2 Compiling and Running of the Program

The program is written according to Java 1.7 and compiles on the commandline with standard command `javac RumourRoutingApp.java`. Invoking the compiled program is done by calling `java ./RumourRoutingApp.class` from the command line. A typical run will result in screen output similar to the following:

```
Event at x: 30 y 120, at time 1135, id 317
Event at x: 210 y 20, at time 1040, id 286
Event at x: 0 y 50, at time 1422, id 389
Event at x: 150 y 250, at time 1410, id 385
Event at x: 130 y 0, at time 1826, id 490
Event at x: 130 y 220, at time 3002, id 748
Event at x: 50 y 270, at time 4546, id 1164
Event at x: 130 y 210, at time 3848, id 977
Experiment finished
```

A number of parameters can be modified in the `RumourRoutingApp.java` file. The meaning of these will be described in the section 'Description of Program Structure'.

```
int NODES_X                 = 50;
int NODES_Y                 = 50;
int NO_NODES                = NODES_X * NODES_Y;
double NEW_EVENTS           = 0.0001;
int NODE_RANGE              = 15;
double PROB_AGENT           = 0.5;
int TTL_AGENT               = 10;
int TTL_QUERY               = 50;
int QUERY_NODES             = 4;
int QUERY_PERIODICITY       = 400;
int TIMESTEPS               = 10000;
int NUMBER_OF_RECENT_NODES  = 5;
int QUERY_RESEND_WAIT       = TTL_QUERY *8;
```

2.1 Javadoc

JavaDoc pages were created with the built-int functions in IntelliJ and can be found in the javadoc subdirectory. The pages are in HTML format and can be viewed by opening index.html in the javadoc directory with a web browser.

3 General Program Structure

The design of our rumour routing implementation from 'OU3' assignment proofed to be mostly feasible for implementation. Certainly, some details that were left open in the design slightly changed other structures, but mostly the presented code can be seen as an implementation of 'OU3'. Hence below follows just a rather short account for the general design and we will focus more on practical implementation details.

A general aim of object oriented design is to develop a model where real world entities have their counterparts modelled as classes/objects with similar properties, behaviour and relations. For the current implementation, the real world entities of interest are sensor *nodes* with the capability to detect events and send/receive information. These three activities define two other abstract real world entities: *events* and *messages*. The former are *detected* and the latter can be *sent* and *received*. Finally, as we are interested to investigate specific situations involving a large number of the above mentioned entities, we also consider the *environment* a required entity in our implementation. Hence, *environment*, *node*, *event* and *message* were chosen as base classes. To modell the behaviour described as *rumour routing* algorithm, we designed specialications of the class *message*: *agent* and *query*.

3.1 Initialization of System

Using the variables to fit the specifications we construct an environment class. Iterate through every node, giving them each a position, such that they align as a grid. Four random nodes are made query origin nodes. Initialize neighbours for every node using the node range. Run experiment advancing time.

3.2 Query Movement Algorithm

Set time to live to 0.

Entering a node

If mode is set to *search*. Then if the event is found in the current nodes list of events change mode to *track*. If mode is set to *track*. Then iterate through current nodes list of events. If a match for the target event is found. Then if the distance to the event is 0. Put a clone of the found event into the query. Save current nodes coordinates. Change mode to *homing*. If mode is set to *homing*. Then if current nodes matches the original node. Remove the query from the resendable queries. Print the events destination nodes X and Y, time and ID. Reset time to live.

Choosing where to go next

If mode is set to *search*. Iterate through current nodes list of neighbours. When it finds a node that is not in the list of visited nodes. Then if that node is not busy. Put that node on the route stack. Return the destination node. If mode is set to *track*. Iterate through current nodes list of events. If it finds its target event in the list. Iterate though current nodes list of neighbours. When it finds the matching node in the list. If that node is not busy. Put that node on the route stack. Return the destination node. If mode is set to *homing*. If the top of the stack is not a busy node. Return the top of the stack.

3.3 Agent Movement Algorithm

Clone the event and put into routing map

Entering a node

Iterate through the routing map. If current nodes eventlist contains an eventkey matching the agents eventkey. Iterate through the nodes eventslist. If the nodes event key matches the agents event key. Then if the the nodes event keys distance is greater than the agents. Update the nodes eventlist. If the current nodes eventlist does not contain a eventkey matching the agents eventkey. Update the nodes eventlist. Iterate through the nodes eventlist. If current agents routing map contains an eventkey matching an eventkey in the nodes eventlist. Iterate through the agents routingmap. If the agents event key matches the nodes event key. Then if the agents event keys distance is greater than the nodes. Update the agents routingmap. If current agents routing map does not contain an eventkey matching an eventkey in the nodes eventlist. Updte the agents routingmap. When sent update the routingmap event distance and node ID.

Choosing where to go next

Iterate through current nodes list of neighbours. When it finds a node not in the list of recent nodes. If that node is not busy, Return node.

4 Specific Design Decisions

4.1 A generic Environment

To obtain a generic environment that allows positioning nodes in which ever way desired, a coordinate based system was implemeted. To simplify the main loop, a data structure containing the neighbour nodes in send/receive range is generated prior to the actual experiment. During the experiment, no more distances need to be calculated, only the initialized node neighbour data structure can be used.

4.2 Time managment

In our simulation, time is semi continuous, turn/step based. In each time step, each node has at least once focus to do one activity. According to the assignment instruction, activities that consume such a time slot at the node are *sending* and *receiving* a message while *detecting* an event and *creating* messages does not consume a time step.

Busy States

The steps that consume a time slot at the node (sending/receiving) always involve two nodes. Hence, two scenarios related to concurrency have to be handled: (1) A node wants to send a message and the receiver node has already used its turn for the current timestep, or, (2) when the activity sequence comes to a node, that has already used his turn by receiving a message earlier in the same time step.

We defined a boolean *busy state* property on the node class which is switched to *true* after having used the current time step. In the end of a time step, the busy state of all nodes is reset back to *false*.

4.3 Events - copy or clone?

Event construction is initiated by the environment at individual nodes. The event object is first stored in a routing table at the node where it occured. In our implementation, event objects are carried around as piece of information within messages. They have a *distance* property that indicates the distance in node jumps back to the node where the event occured.

As event objects are carried around by messages, they have to update the distance property individually, hence, they need to be clones rather than just references pointing to the same event object. To recognize the equality of two events, the hashcode and equalTo methods were overridden to make use of the *eventId* property instead of the object reference.

4.4 Managing the Routing Tables

Routing tables where implemented as HashMaps with the event as Key and the *NodeId* as value. Routing tables are present both in *nodes* and in *agents*. Updating and managing the routing tables is the main action of agents. First both HashMaps are updated with events missing in one or the other and second, if an event is available at the node and at the agent, the one with the shorter distance to the event node is chosen.

Currently, these operations are done in rather clumsy nested *for loops*. The problem with the current code is that it involves accessing properties (distance to event node) of a specific key. This is not a normal supported operation of a HashMap, hence, it requires another *for loop*. A improvement to this structure will be proposed later.

4.5 Where to go - nextNode mode

Finding the next node to visit is implemented individually in *agent* and *query*. The method in the *query* class needs to handle three different cases, depending on the current status of the *query*. The method of the *search* mode is the same as the *nextNode* method of the *agent*. Both *query* and *agent* inherit from their superclass *message* a *queue* that holds the most recent visited *nodes*. In the practical implementation, instead of a *queue*, the datatype *dequeue* was chosen which allows also to check the last element added.

The *nextNode* methods of *query* for *track* and *homing* method were straightforward. The former matches his query event with the event in the routing table and read out the nodeId. While the latter pops nodes from the route stack which holds references to the nodes travelled.

Figure 1 shows the UML diagram of the chosen design, automatically produced using IntelliJ.

5 Limitations and Future Development

5.1 General Datastrcuture

Current limitations are mostly two-fold: First the current implementation has a rather bad time complexity due to several nested for loops. This is a result of the chosen data structures and grouping of information. For example is currently the routing table a *HashMap* with a relation between an *Event* object and a *nodeId* (Integer). The nodeId represents the direction to follow for finding the event node in question. However, as event objects that, according to the *equalTo* method are idenical, can contain different information (distance to event node). Hence sometimes, also HashMap keys need to be looked up which is not a normal operation in the interface of a Hash Map. This is the main cause for the nested *for loops*. In a improved version, event obejcts should also include the *nodeId* of the path to the event node as this information anyway is associated to the distance already found as property in the *Event*.

5.2 Validation of Result

Currently, the only validation that the implementation works correct has been done by elaborate debugging sessions. A help here was the very generic implementation which allowed to quickly modifying the experimental parameters. The currently available test cases asses the validity of individual functions and some smaller structures. In a future versions more general test cases should be added that unambiguously track and validate the whole lifecycles of individual objects, from instantiation to desctruction. Also a number of more higher-lever situations need to be tested: What happens to a message when all neighbour nodes of the current node have already been visited? Could such a message block the message queue of a node?

6 Testing Framework

Our aim was to have JUnit tests ready before programming but we ended up using the pseudocode as a base to produce our actual code in most cases, testing functionality while making the code. We have JUnit tests to make sure our constructors and methods work properly. We also have other ways of testing our program through exceptions and try and

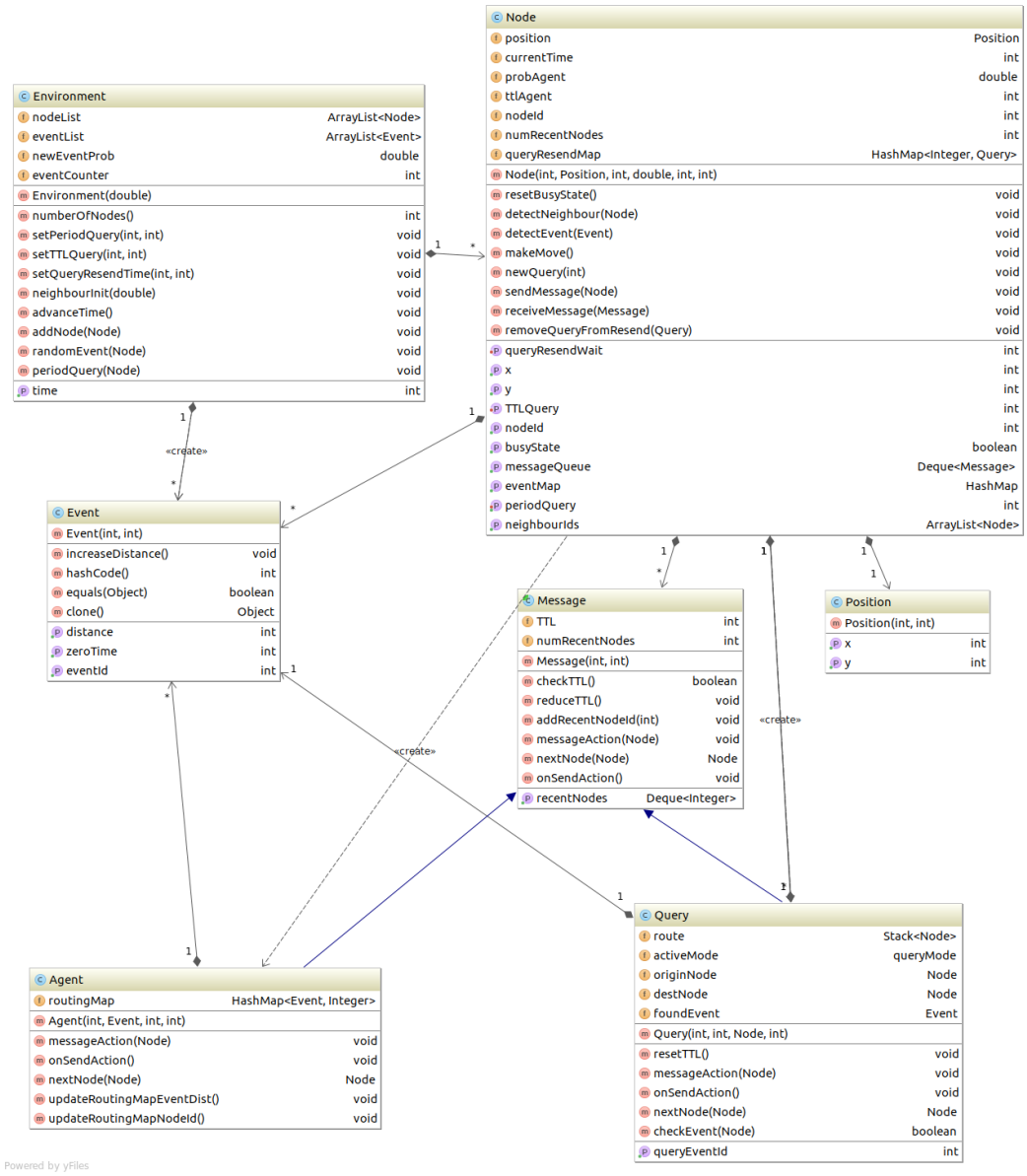

Figure 1: *UML diagram for implementing a sensory network application that allows testing of the rumour routing algoritm.*

catch methods. Printouts of certain scenarios, smaller networks etc, were handy when recoignizing if the program ran as expected.

```
Event is created id: 87 time: 370
An agent is created
Event is created id: 88 time: 372
Event is created id: 89 time: 375
An agent is created
Event is created id: 90 time: 376
An agent is created
Event is created id: 91 time: 376
Event is created id: 92 time: 376
Event is created id: 93 time: 381
An agent is created
Event is created id: 94 time: 382
An agent is created
Event is created id: 95 time: 387
Event is created id: 96 time: 396
Query created searching for event id 33
Query on search mode
Query created searching for event id 21
Query created searching for event id 77
Event is created id: 97 time: 400
An agent is created
Query on search mode
Query on track mode
Query on search mode
Query on track mode
Query on track mode
Event is created id: 98 time: 410
An agent is created
Event at x: 60 y 70, at time 333, id 77
Query on way home
An agent has reached zero in time to live
Event at x: 60 y 130, at time 129, id 33
Query on way home
An agent has reached zero in time to live
An agent has reached zero in time to live
Event is created id: 99 time: 428
An agent is created
Event is created id: 100 time: 428
Event is created id: 101 time: 429
An agent is created
Event is created id: 102 time: 434
Event is created id: 103 time: 440
```

In this is an example of the printout from timestep 370 to 440, we see agents and events getting created, an agent dieing, queries working as intended searching, tracking, finding event, reporting and finally going home.

7 Reflections and Account for Individual Contributions

Below follows a short account of each group member. All work was done on a public github repository which can be found here: `https://github.com/lorenzgerber/rumour_routing`.

7.1 Johan Eklund

We had a pretty good idea what we wanted to do right from the first meeting. We made a solid UML diagram early on and we used it to make pseudo-code that helped us a lot through the implementation of the code. We got a group chat and a git setup early in the project which has helped a lot. We had an early idea of making unit tests before implementing code, in the end we did mostly the other way around, which made the tests a bit redundant. Differences in knowledge of the different tools has made some of the group members spend a lot of time learning the tools and not contribute as much as the more knowledgeable members. More meetings and joint programming may have helped. The problem specification can be read in different ways and a lot of different solutions can be made for whatever you believe the assignment to be. I assume this is part of the idea of the assignment. I made the initial CRC suggestion that later became our first UML-diagram. Worked on the position class and the main class that in the final version became RumourRoutingApp. Made other small contributions to other classes among them exceptions for a couple of classes. Wrote JUnit tests working with Tommie. Worked on the final report. Had some fierce battles with git.

7.2 Tommie Lindberg

We had a pretty good idea how we wanted to build the program from the start and a good pseudo code to follow. During the implementations did we discover some problems we may interrupt and had some differences how we wanted to avoid that. We discussed the possible solutions and decided the one that may work best. The co-operation of this assignment could have worked better, we work in different ways but we tried to stay in contact with each other on a mobile group message application called Slack. We did also have a common github folder that we worked with. Some people of the group didn't work so much with the assignment and one did more of it. I guess this is what we are about to learn about working together with other people on an assignment. That people work in different ways and the key is to find a way that works for the co-operation of a project. I implemented the message class in the program, we then encountered some things we wanted to change about it, so we did. My main implementation in this program are all the test classes which are built with junit tests and also added some exceptions.

7.3 Jakob Lundin

no text received

7.4 Lorenz Gerber

I enjoyed this assignment as I think it helped to get a first impression of Java coding and object oriented software design. My contributions can be looked up on the public github address above. Further I also set up the latex documents, the git repo and the slack channel.

References

[1] D. Braginsky and D. Estrin. Rumor routing algorthim for sensor networks. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 22–31. ACM, 2002.

[2] Umeå University, 5dv133 obligatorisk uppgift 3. `http://www8.cs.umu.se/kurser/5DV133/VT16/uppgifter/ou3/`, 2016. accessed: 2016-04-28.