Umeå University
Department of Computing Science

Object-Oriented Programming Methodology 7.5 p
5DV133

OU3 Sensor Network

Submitted
2016-05-04

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

The assignment was described on the course homepage [4]. The main aim idea was to develop software that allows to perform experiments on sensor networks as described in Braginsky and Estrin [1]. The main topic of [1] is the use of *rumour routing* as an energy saving message transportation algorithm that for example be used in environment surveillance networks.

In object oriented software design, it is common to build a model representation of the real world system [2] by defining classes the correspond to the real world systems' entities. Here the real world system is a sensor network as described in Braginsky and Estrin [1]. The realworld entities modelled in this assignment can be classified in two main groups: Physical components such as the sensor nodes and non-physical ones, information packages travelling the network, such as the queries and agents. Further, a third type, the environment entity simulates the real surrounding.

Unified Modelling Language (*UML*) diagrams were composed according to Börstler [2]. The theory of rumour routing is described by Braginsky [1]. Horstman was used as Java language reference [3] for chosing datatypes.

2 Unified Modelling Language Class Diagram

Figure 1 shows the UML diagram of the chosen design.

3 Class Descriptions and Pseudo Code

3.1 Initialization and State Stepping

We define our variables to fit the specifications, then constructs an environment class. The program then iterates through every node, giving them each a position, such that they align as a grid. Four random nodes are given a value for periodQuery, making them the query origin nodes. To create a neighbour list, we iterate through every node, checking distance to each node and storing the nodes in reachable distance in the Node.neighbors list. After this, the network is initialised and we loop the myEnv.advanceTime() function to run the program.

```
Main
{

  //Globals
  NODES_X = 50
  NODES_Y = 50
  NO_NODES = NODOES_X * NODES_Y
  NEW_EVENTS = 0.0001
  NODE_RANGE =
  TTL_AGENT = 50
  TTL_QUERY = 50
  QUERY_NODES = 4
  QUERY_PERIODICITY = 400

  // Create Environment
```

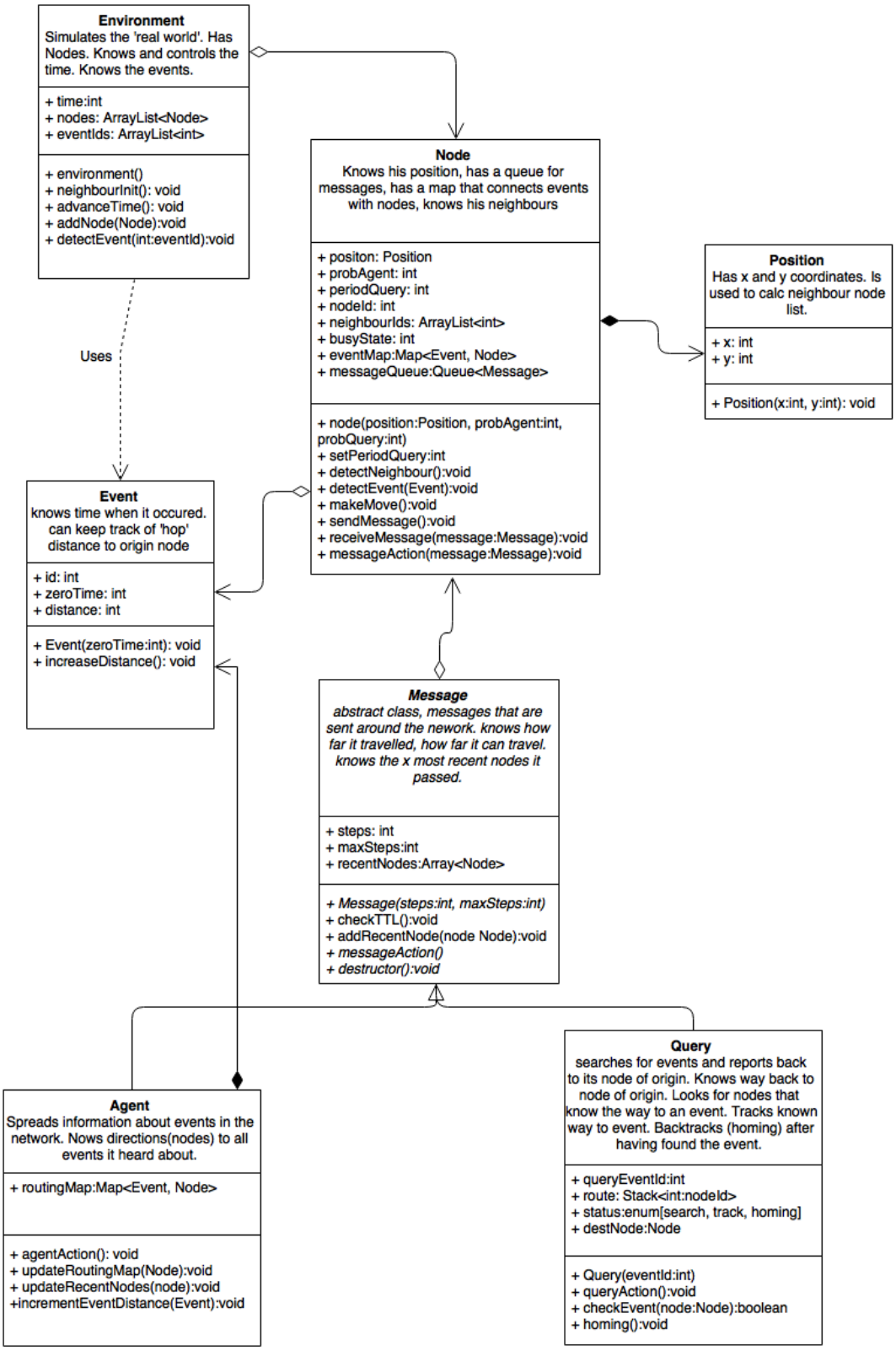

Figure 1: *UML diagram for implementing a sensory network application that allows testing of the rumour routing algoritm.*

```
  myEnv = Environment()


  // Create and position Nodes
  For (x = 1:NODES_X)
  {
    For (y = 1 -> NODES_Y)
    {
      myEnv.addNode(Node(Position(x, y),0.5, NULL))
    }
  }

  // Set Query nodes
  queryNodeIds[] = sample(1, NO_NODES, QUERY_NODES)
  For (i = 1:QUERY_NODES)
  {
    myEnv.nodes.getNode(queryNodeIds[i]).setPeriodQuery(QUERY_PERIODICITY)
  }


  // Create Neighbour list
  // For every node
  For(nodes_create_list = 1:NO_NODES)
  {
    // create neighbour list

    For(nodes_check_distance = 1:NO_NODES)
    {
      // neighbourInit() checks distance and adds
      // node id to neighbour if in range
    }
  }


  //
  For(timesteps 1:10000)
  {
    myEnv.advanceTime()
  }
}
```

3.2 Environment Class

The Environment class stores information in the network, stores nodes, keeps track of events, and controls time. It is the simulation environment, which simulates the ‘real world’. The advanceTimer() lets each node make a “move”, sending a message if there is a message in its queue.

```
Class Environment
```

```
{

  int time
  ArrayList<Node> nodes

  Environment(){

    new Environment
  }

  advanceTimer()
  {
    //id 1:NO_NODES){
    makeMove(node_id)
  }
}
```

3.3 Node Class

The Node class stores information about specific nodes, such as position, neighbouring nodes, and paths to events. The nodes also send and receive messages. Each time the makeMove() method is called, the detectEvent() method "rolls a die" to see if an event will be created on the node. If there is an event, there is a chance of an agent being created, according to the specified probability. If it is one of the query origin nodes, it checks the queryPeriod. If 0, it creates a new query and puts it on queue. If its BusyState is false, it will send a message from its queue. When a message is sent, the method checks to see what type of message it's trying to send. If the message is a query, it then checks the query state to see where it should be sent. It then checks if the intended receiver is free, if not, the message is sent. The method recieveMessage adds the incoming message to its queue called messageQueue. It will then set the node in busyState so that the node can't receive any more messages this timestep. If the message is received from an agent it will increment the events distance with one.

```
Class Node
{
  detectEvent()
  {

  randomFunction(PROB_NEW_EVENTS){
  eventMap.add(Event(), self.nodeId)
  }

  makeMove()
  {

  // new event? (detectEvent())
    // Random function to create Agent
  // New Query?
    //put query on message queue
```

```
  // Do we have a message on Message Queue
    // Do messageAction()
    // BusyState, have we already done our move
    // if we can send, send message

  // reduce queryPeriodicityCounter by 1
  }

  sendMessage()
  {
  // find out receiver
    // Check type of message
      // if message "Query"
        //check Query status (search/track/homing)
    // is receiver free
      // send message
      // call receiveMessage of receiver
      // set BusyState
  }


  receiveMessage()
  {
  // Add message to messageQueue
  // set BusyState
  // is Message Agent?
    // call Event Increment distance
  }
}
```

3.4 Classes Message, Query and Agent

The Message class is an abstract class and will use the abstract method messageAction. This will be used so that the node don't need to determine if the message is an agent or query.

The Query class extends message and will call to the public method queryAction. The queryAction will first check which mode is initialised for the query. The query has three different mode setups, the search, track and homing mode. When the query is in search mode it will step through the nodes in the routingMap and all the steps is inserted in a stack called route. It will look in each nodes eventMap until it finds the unique eventId it is set to find. If the eventId is found the query switches to track mode. Track mode is used to find the unique events origin. It will track through the neighbours to the node where the event first was found until it finds where the event distance is zero. When the origin of the event is found it will store the node in destNode and it switches mode to homing. Homing mode is used to track back the same way it went to the start node. The query will use the stack route to find the exact way it stepped until the stack is empty. It will then exclaim the events position, the time the event happened and which node caught the event.

The Agent class extends message and will call to the public method agentAction. The

queryAction will compare it's eventMap with the nodes eventMap. If they are not alike the node will add the differences to it's own eventMap. The agent will then check it's time to live. If the time to live is zero the agent will not step to another node. If it's not zero it will decrement it's time to live before it steps to another node.

```
Class Message
{

  messageAction()

}


Class Query
{

  queryAction()
  {
  // checkMode

    //// Search Mode
    // check eventId in Node eventMap
    // if path found
      // switch mode

    //// Track Mode
    // check destination reached
      // switch mode

    //// Homing Mode
    // check start reached / is stack empty
      // exclaim Event(Node Position, zeroTime, NodeId)
  }
}


Class Agent
{
  agentAction()
  {

  // compare and adjust eventMaps between agent and node
  // check TTL
   // decrement TTL
  }

}
```

4 Testing Framework

The program will be tested with both JUnit tests for methods and smaller programs to test the functionality of the different classes. During the building of the classes we will use JUnit tests, to make sure the constructors and methods work properly. We will also use different exceptions and try and catch methods because that will make it easier for us to determine and find where something goes wrong in the code. When the building of a class is completed we are going to build a small program that we know only will get one result if it works as we expect. This program will be used on the finished class so we know that it works and we get the results we want out of the class. When all the classes are built and tested we will put them together to a complete program and some easier tests will be made for the program so we know it's stable and works as we want.

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] Jürgen Börstler. Objet-oriented analysis and design through scenario role-play. `http://www8.cs.umu.se/kurser/5DV081/CRC_UMINF04.04.pdf`, Umeå University, Department of Computing Science, Umeå, Sweden, 2004. accessed: 2016-04-24.

[3] C. S. Horstman. *BIG JAVA Early Objects*. John Wiley & Sons, New Dehli, 5 edition, 2014.

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