Green Elevator

Andreas Gustafsson || andreg@kth.se
Mattias Knutsson || matknu@kth.se

9 mars 2012

Innehåll

1 Assumptions & Simplifications

- The panel button 'Stop' is emergency halt. The elevator is not supposed to restart after a stop event without maintenance. Therefore the controller thread of an elevator that emergency stops will terminate.
- It is assumed that it takes twice the time to open and close the doors than it takes to travel one floor[1].
- We assume the system should run for eternity (the elevators should never be taken offline. Therefore no communication teardown or synchronized thread termination is implemented.
- When a button is pressed (not a panel button), the controllers are told only which direction is desired, not the exact destination.
- The controllers does not know how many floors there is. It is assumed that a floor request from the elevators will always be within the bounds of the building i.e it is not possible to press the button for the seventh floor in a six-story building.
- There is no way to determine how many people are in the elevator at any one time. Example:
  Five people embark on an elevator on the second floor, and request traveling to the fourth floor. Whilst they travel from the second to third floor, a lone man presses the up button on the third floor. Assume it takes thirty seconds to traverse one floor, and thirty seconds to open the doors, admit passengers and close them again. If the elevator pause on the third floor to admit the lone man, it wastes two and a half minutes of the passengers time (thirty seconds each). If it instead travel to the fourth floor and let them disembark before going down to the third floor to pick up the lone man it would only waste a minute and a half (the lone man waits thirty seconds for the elevator to travel from the third floor to the fourth floor, thirty seconds while they disembark and thirty seconds for the elevator to return to his floor.
  The controllers will completely disregard this and pick up the lone man (it could be a hundred men waiting, the controllers know only that the button was pressed once).

[1] As observed in the elevators in the E-building, KTH Campus Valhallavägen.

2 Application

The application that has been added to the Elevators project consists of three Java classes; MasterController.java, ElevatorController.java and Message.java.

2.1 MasterController.java

MasterController receives messages as strings over TCP sockets from the elevators, converts them into messages as described in section 2.3 and forward them to the correct ElevatorController, and receives messages, as described in section 2.3 from the ElevatorControllers and forward them to the elevators. The only ”actual” work the MasterController does is to allocate floor button presses to an elevator, for more information see section 3.1.

2.2 ElevatorController.java

This class manages a single elevator and trusts the MasterController makes the best decisions. The class receives tasks from the Master and incorporates them into its current schedule, see section 3.2 for how this is done. The ElevatorController will send four different kinds of messages to its allotted elevator. It will send ”move in X direction”, ”stop moving”, ”open doors” and ”close doors”.

2.3 Message.java

This class represents a multi-purpose message sent between an ElevatorController and the MasterController. When a message is received from the elevators by MasterController, the string is parsed into a Message and will remain a Message until the command is successfully accepted by the Controller. A Message may be created by the ElevatorController and sent to the Master to be converted into a string and sent back to the elevator through the socket.

3 Algorithm

3.1 Elevator selection

This algorithm is used by the master controller when a floor button is pressed and decides which elevator to server the request. The algorithm will first search for an elevator to ”join”. To ”join” an elevator is to enter an elevator that is already heading in your direction, and has an destination past yours. For instance, an elevator is traveling to floor four from floor one, and you press the ”up” button on floor three. The elevator will then pause at floor three to admit you, before proceeding to floor four.
If there is no elevator to join, the algorithm searches for a free elevator to send to your floor. If it finds one it will be assigned to you and others may ”join” your ride.
If there is neither an elevator to join nor and empty, the algorithm will wait until one of the cases is true. If both are true, a ”join” will be prioritized over assigning an empty elevator.

3.2 Floor scheduling

This algorithm is used by each elevator controller, oblivious to what the other elevator controllers are doing, to schedule the order of floors to visit.
When a new task (a task is ”travel to floor X from the current position before open and close the doors”) is added, it goes through a series of checks to determine when it should be performed.

1. If the elevator is not moving, execute the task immediately.

2. If the elevator is moving upwards, and the new target floor is below (if traveling upwards) or above (if traveling downwards) the elevator's current floor, the button press is discarded. This means that someone has requested to travel one direction, and then pressed a button that would cause the elevator to reverse direction. Douchebaggery is unacceptable.

3. If the new task is acceptable, it is placed into a queue of tasks. This queue is a priority queue that is ordered by the target floors as integers descending if the elevator is currently moving downwards and ascending if it is moving upwards.

4 Implementation

4.1 MasterController

This class forwards messages from the elevators to the proper ElevatorController and assigns a proper elevator and controller to a floor request (when someone on floor X request an elevator that is moving in direction Y). MasterController extends Thread, and is started from within the initialization of Elevators. See appendix B on page page 13 for code and fig. 1 on page 9 for a flow chart.

4.1.1 run()

Overrides the Thread.run() method. What it does is to connect to the elevators through TCP socket, opens the I/O streams and invoke the controlElevators function (see **??**).

4.1.2 controlElevators()

Main workhorse of the MasterController. It starts a thread that will repeatedly poll each controller for a message they wish to be sent to the elevators, and forward it. This thread is anonymously created by wrapping it around a runnable and started inline. Proceed into an infinite loop that reads messages from the socket, and if it is an ”p” or ”f” message, it is forwarded to the proper ElevatorController. If it is a ”b” message, an Assigner (see **??**) is inline instantiated and trusted to manage the message.

4.2 Assigner

This transforms a ”b” message to a ”p” message and forwards it to the proper Elevator-Controller, see section 3.1 for more info about the algorithm.
Assigner extends Thread. See appendix B on page 13 for code and fig. 2 on page 10 for a flow chart.

4.2.1 run()

The assigner knows which floor is requested, and which direction is requested.
Its task is to determine which controller gets the requested task. There are three different outcomes. First is to ”join” a ride (see section 3.1 for clarification of ”join”), the second is to assign an empty elevator to the task. The third solution, if both previous are false, then simply have the Thread yield and repeat the first two steps during the next context switch. Eventually one of the two will be true and the assigner posts the task to the proper thread and terminates.

4.3 ElevatorController

This controls a single elevator. Will receive messages from the MasterController and handle the tasks contained within those messages. ElevatorController is both a sort of monitor and extends Thread (it interacts with the MasterController through synchronized method calls). See appendix C on page 18 for code and fig. 3 on page 11 for a flow chart.

4.3.1 run()

This is the main workhorse of ElevatorController. While it has nothing to do, it yields. It detects if it has something to do by first checking if its inbox (a queue where MasterController puts messages destined to this controller) is empty, and that its taskQueue is empty.
If either of these two conditions fail, it will look at the first element of its task queue (without removing it). If it is not null, and its current floor is not within a ± 0.05 interval of the target floor[2], it means it should move towards that floor. It will decide which type of movement is required with decideMove() (see **??**).
It will the proceed to check its inbox, to see if any messages have arrived from the elevators (via MasterController). If the message is not null, there is something. If the message is a "p" message with floor value of 32000, it is an emergency stop and a stop message is immediately sent to the elevator.
If it is a "p" message and not an emergency stop, it means that it is a new task, so the message is added to the taskQueue (see section 4.3.2).
If it is a "f" message, it means that the elevator has moved and the current floor field must be updated. Now, if the current floor is within the ± 0.05 interval of the target floor, a "stop moving" message is sent to the elevator and the doorAction() function is invoked (see section 4.3.4).

4.3.2 addQueue()

This function will add a task to the task queue, and it will do it in one of two ways. If the elevator is intended to move downwards, it will add the task such that the queue is ordered in a descending manner with the highest floor at the head of the queue.
If the elevator is intended to move upwards, it will att the task such that the queue is ordered in an ascending manner with the lowest floor at the head of the queue.
If the new task would cause the elevator to change directions without the task queue being empty at least once before the direction change, the task is ignored.

4.3.3 decideMove()

This function decides if the elevator should move, and in what direction.
If the elevator is already moving, the function call does nothing.
If the current floor is lower than target floor minus 0.05 (see section 4.3.1 for explanation), the direction is upwards and a "move up" message will be sent to the elevators.
If the current floor is greater than the target floor plus 0.05 a "move down" message will be sent to the elevators.
If the current floor is within the interval and the elevator is not moving, this function call does nothing.

4.3.4 doorAction()

This function opens and closes the doors of the elevator. It will send an "open door" message, sleep for a second, send a "close doors" message and sleep a second before returning.

[2]The elevators will send floor update messages with 0.04 intervals.

4.3.5 Remaining methods in ElevatorController

They are merely synchronized getters/setters adders/pollers.

4.4 Message

Message is a multi-purposed message that is sent between the MasterController and ElevatorControllers. See appendix D on page 25 for code.
A message contains the following fields:

- type: 'p', 'f', 'm', 'd'.
- elevator: Which elevator/controller to the message should go to.
- targetFloor: if it is a 'p' message, it will be the floor the elevator should go to. If it is a 'd' message, it will be 1 for open doors and -1 for close doors. If it is a 'm' message it will be 1 for move up and -1 for move down. If it is an 'f' message, targetFloor is unused and may be whatever.
- curPos: If it is an 'f' message, it will be the current floor. Otherwise it is unused and it will be whatever.

4.4.1 Methods of Message

They are simply getters/setters and overrides of equals and toString. Just standard stuff.

5 Environments

5.1 OS & Java

The program has been developed on, and tested on, a computer running Genuine Windows 7 and Java (TM) 6 Update 22 (64-bit).

5.2 System Specifications

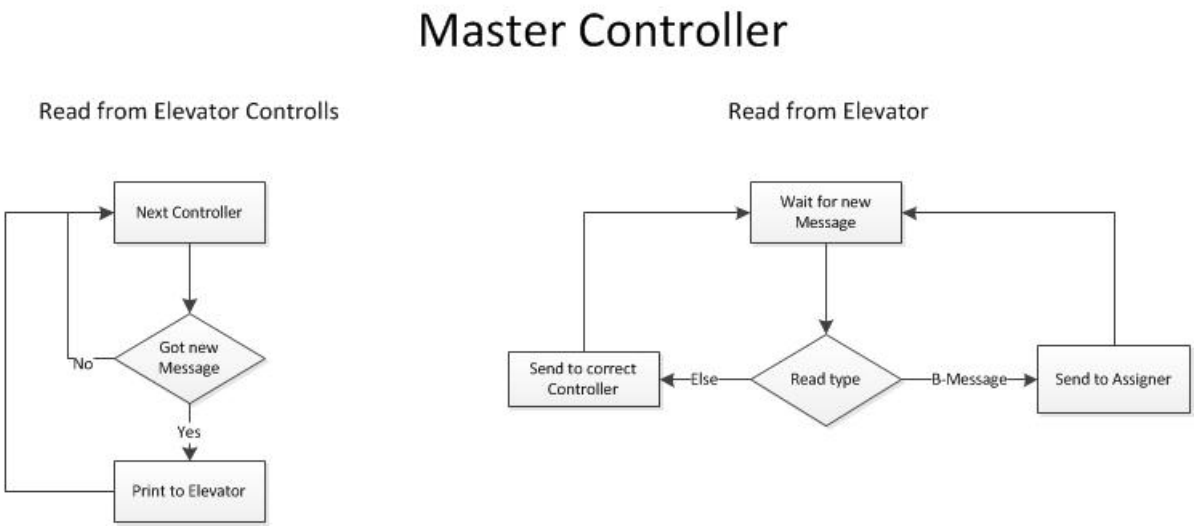
Computer model: Asus UL30VT
OS: Windows 7 Professional 64-bit (6.1, Build 7601)
System Manufacturer: ASUSTeK ComputerINC
Processor: Genuine Intel(R) CPU Û7300 @ 1.30GHz (2 CPUs), ĩ1.3GHz[3]
Memory: 4096MB RAM

[3]Clocked to 80% efficiency for power conserving purposes.

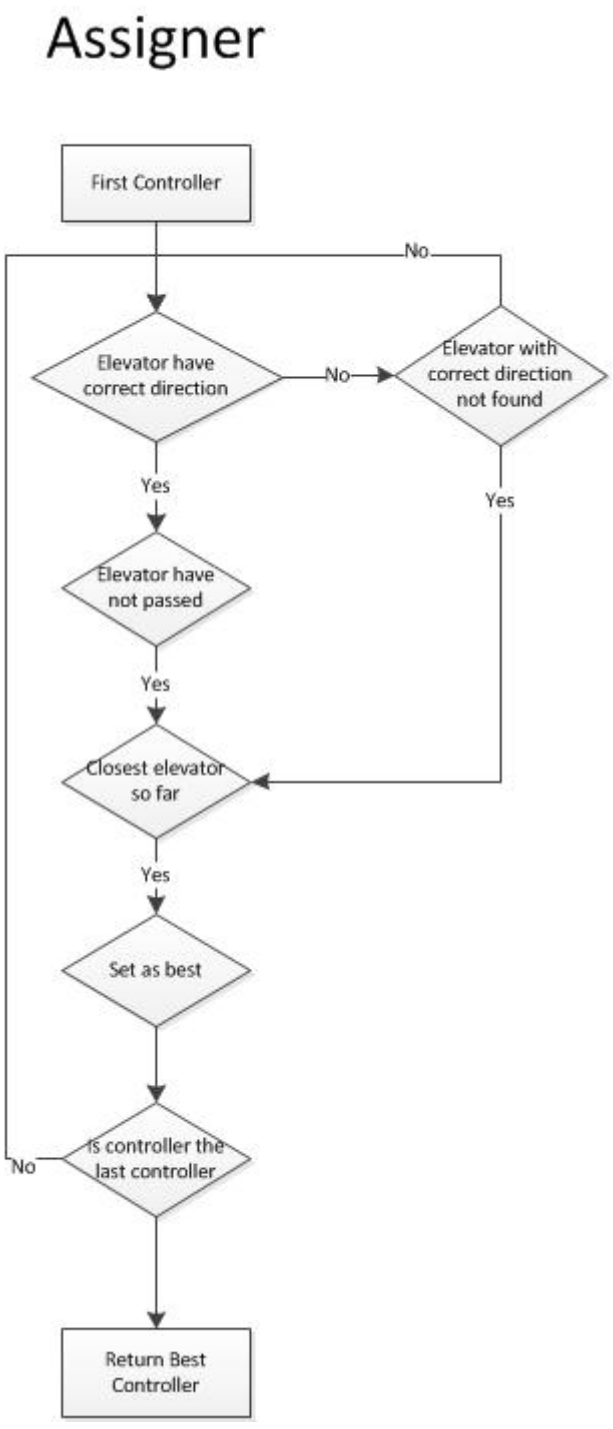
6 Achievements

The task was to minimize service time and elevator movement. It is not possible to do both, one must be sacrificed to benefit the others. The controllers cannot make accurate computations since it may not observe the entire theater. The elevator does not know how many people there are in the elevator, nor how many wish to embark. Nor is it possible for the controller to know how far a person (or group of people) want to travel when they summon they summon the elevator. The controller only knows the desired direction.
Therefore we chose to, if possible, combine trips with the elevator. For instance, one person travels from the second to the fourth floor, and someone summons the elevator to travel upwards from the third floor. Then the elevator already traveling upwards will gather this request and pause at the third floor. This way the goal of minimizing elevator movement (by having "required moves" overlap, and minimize service time by not locking an additional elevator.

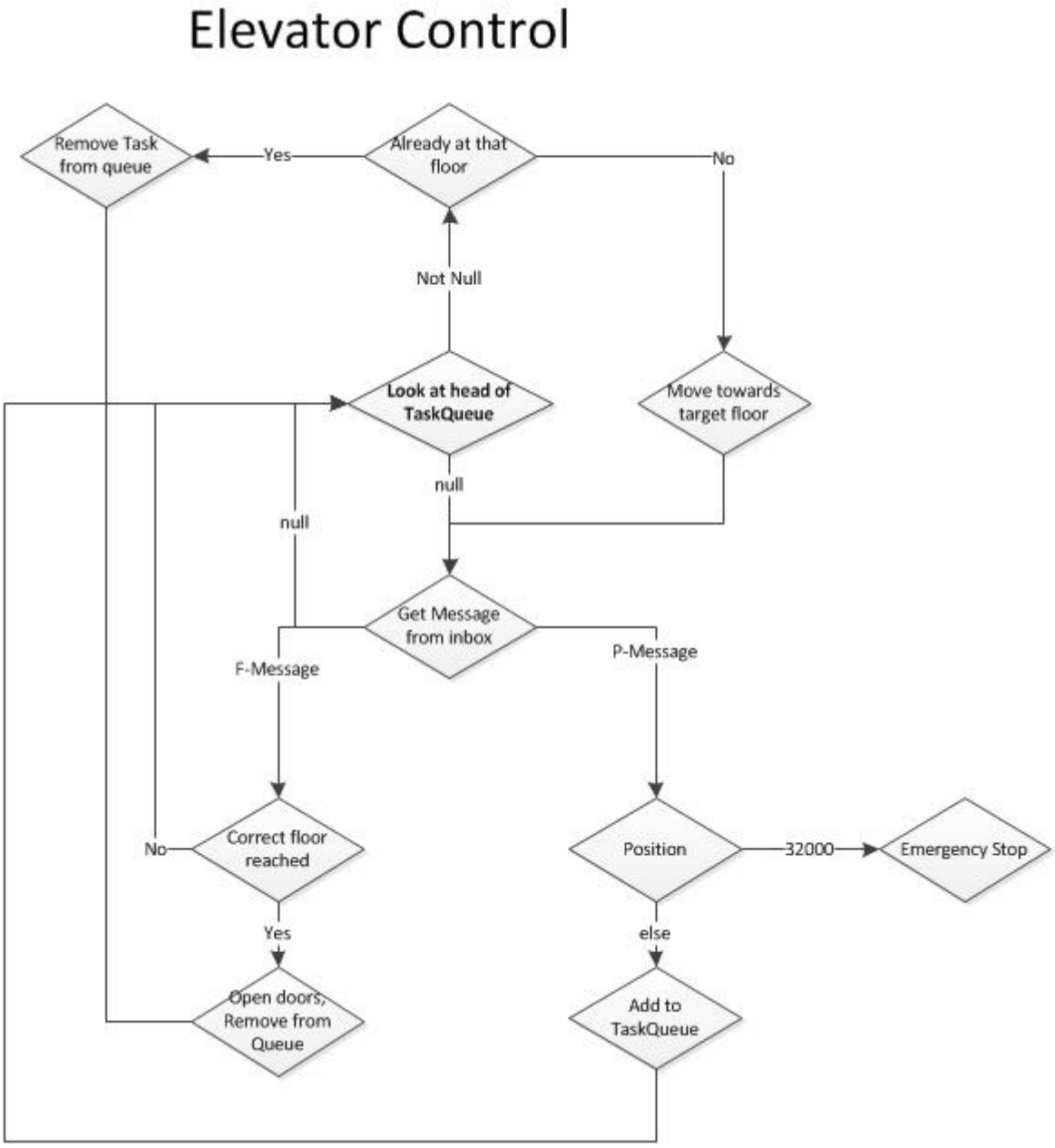
A Flow charts



Figur 1: Flowchart for MasterController.



Figur 2: Flowchart for Assigner.



Figur 3: Flowchart for ElevatorController.

B MasterController.java

```
1  package controller;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.io.PrintWriter;
7  import java.net.Socket;
8  import java.net.SocketException;
9  import java.net.UnknownHostException;
10 import java.util.ArrayList;
11
12 import elevator.Elevators;
13
14 /**
15  *
16  * @author Mattias Knutsson and Andreas Gustafsson
17  *
18  */
19 public class MasterController extends Thread {
20     private Socket s;
21     private BufferedReader in;
         // The reader from the socket
22     private PrintWriter out;
         // The writer to the socket
23     private final ArrayList<ElevatorController> controllers;
         // The list of controllers
24
25     public MasterController() {
26         controllers = new ArrayList<ElevatorController>();
27     }
28
29
30
31     /**
32      * Initialize the socket and read/write, then run
           controllElevators
33      */
34     @Override
35     public void run() {
36         while (true) {
37             try {
38                 s = new Socket("localhost", 4711);
39                 in = new BufferedReader(new InputStreamReader(
40                         s.getInputStream()));
```

```
41                  out = new PrintWriter(s.getOutputStream(),
                        true);
42                  controllElevators();
43              } catch (UnknownHostException e) {
44                  continue;
45              } catch (IOException e) {
46                  continue;
47              }
48              break;
49          }
50      }
51
52      /**
53       *
54       * @throws SocketException
55       */
56      private void controllElevators() throws SocketException {
57          System.err.println("Controller starts.");
58          for (int i = 0; i < Elevators.numberOfElevators; i++) {
                        //Start all controllers
59              controllers.add(new ElevatorController(i + 1));
60              controllers.get(i).start();
61          }
62
63          //A thread that read the controllers outbox and send to
                Elevators
64          new Thread(new Runnable() {
65              public void run() {
66                  while (true) {
67                      for (ElevatorController c : controllers) {
68                          Message msg = null;
69                          if ((msg = c.retrieveMessage()) !=
                                null) {
70                              out.println(msg.toString());
71                          }
72                      }
73                      Thread.yield();
74                  }
75              }
76          }).start();
77
78
79          //Read input from Elevators and convert to a message
80          //If the message is a b-message send it to assigner
81          //Else send it to the correct controller
82          while (true) {
```

```
83              while (!s.isInputShutdown()) {
84                  String[] message = null;
85                  try {
86                      message = in.readLine().split(" "); //Parse
                            the message
87                      char type = message[0].charAt(0);
88                      int elevator = Integer.valueOf(message[1]);
89                      double modifier =
                            Double.valueOf(message[2]);
90                      if (type == 'b') {
91                          new Assigner((int) elevator, (int)
                                modifier).start(); //If b-message
                                send to assigner
92                          continue;
93                      }
94
95                      //Create and send message
96                      Message msg = new Message(type, elevator,
                            (int) modifier,
97                              modifier);
98                      controllers.get(elevator -
                            1).postMessage(msg);
99                  } catch (Exception e) {
100                 }
101
102             }
103         }
104     }
105     /**
106      * Assigner is threads trying to assign a task to a elevator
107      * @author Mattias Knutsson and Andreas Gustafsson
108      */
109     private class Assigner extends Thread {
110
111         private final int floor;
112         private final int direction;
113
114         /**
115          * Constructs a Assigner.
116          * @param floor - The floor a person want to leave.
117          * @param direction - The direction the person want to
                go.
118          */
119         public Assigner(int floor, int direction) {
120             this.floor = floor;
121             this.direction = direction;
```

```
122         }
123
124         /**
125          * Try to assign the task to a elevatorController
126          * The assigner prefers to assign the task to a
               elevator passing the floor
127          * If that isn't possible take the closest unassigned
               elevator
128          * Or what until one of this gets possible
129          */
130         @Override
131         public void run() {
132             ElevatorController closestEmpty = null, closestJoin
                  = null;
133             double costEmpty, costJoin;
134             costEmpty = costJoin = Double.POSITIVE_INFINITY;
135             for (boolean first = true; closestEmpty == null
136                     && closestJoin == null; first = false) {
                        //While the task haven't a possible
                        assignable elevatorController
137                 if (!first)
138                     Thread.yield();
                        //Yield at the start everytime except
                        the first
139
140                 for (ElevatorController c : controllers) {
141
142                     // Join to a current tour
143                     if (c.getIntendedDirection() == direction
                                //If the elevators direction
                            is same as the persons direction and the
                            elevator havn't yet passed the floor
144                             && c.getDirection() * c.getFloor()
                                    < c
145                                             .getDirection() * floor) {
146                         double thisCost = Math.abs(floor -
                            c.getFloor()) + 2     //Calculate the
                            cost
147                                 * c.getTaskQueueSize();
148                         if (thisCost < costJoin) {
149                             costJoin = thisCost;
150                             closestJoin = c;
                                                //Set the
                                controller as closest if it has
                                the least cost
151                         }
```

```
152                     }else if
                          (c.getIntendedDirection()!=c.getDirection()
                          && c.getIntendedDirection()==direction){
                          //If the elevator have the same intented
                          direction but not started to move in
                          that direction yet
153                         double thisCost = Math.abs(floor -
                              c.getFloor()) + 2
154                                 * c.getTaskQueueSize();
155                         if (thisCost < costJoin) {
156                             costJoin = thisCost;
157                             closestJoin = c;
                                                  //Set the
                                controller as closest if it has
                                the least cost
158                         }
159                     }
160
161                     // Assign Empty ELevator
162                     if (c.getTaskQueueSize() == 0 &&
                          c.getInboxSize() == 0) {    //If the
                          elevator is unassigned
163                         double thisCost = Math.abs(floor -
                              c.getFloor()) + 2
164                                 * c.getTaskQueueSize();
165                         if (thisCost < costEmpty) {
166                             costEmpty = thisCost;
167                             closestEmpty = c;
                                                  //Set the
                                controller as closest if it has
                                the least cost
168                         }
169                     }
170                 }
171             }
172             ElevatorController cf = closestEmpty;
173             if (closestJoin != null)                  //If the is
                  a possible join
174                 cf = closestJoin;
175             else
176                 cf.setIntendedDirection(direction);
177
178             Message m = new Message('p', cf.getElevator(),
                  floor, floor);    //Create the message translated
                  to a p-message
179             System.out.println("ASSIGNER MESSAGE: " + m);
```

```
180                 cf.postMessage(m);
                                    //Send the message
181             }
182         }
183 }
```

C ElevatorController.java

```
1  package controller;
2
3  import java.util.ArrayDeque;
4  import java.util.ArrayList;
5  import java.util.Queue;
6
7  /**
8   * Controll
9   * @author Mattias Knutsson and Andreas Gustafsson
10  *
11  */
12 public class ElevatorController extends Thread {
13      private final Queue<Message> inbox;          //The inbox
          with messages from MasterController
14      private final Queue<Message> outbox;         //The outbox
          with messages to MasterController
15      private final ArrayList<Message> taskQueue; //The queue
          with tasks the controller will execute
16      private double floor;                        //The current
          position
17      private double targetFloor;                  //The
          destination floor
18      private final int elevator;                  //The ID
19      private int direction;                       //The direction
          (1 = UP, -1 = DOWN)
20      private int intendedDirection;               //The intended
          direction (the direction after first pickup)
21
22      /**
23       *  Constructs a new elevator with standard settings
24       * @param elevator - The number ID of the elevator
25       */
26      public ElevatorController(int elevator) {
27          setDirection(0);
28          setIntendedDirection(0);
29          floor = setTargetFloor(0);
30          this.elevator = elevator;
31          taskQueue = new ArrayList<Message>();
```

```
32          outbox = new ArrayDeque<Message>();
33          inbox = new ArrayDeque<Message>();
34      }
35
36      /**
37       * The running class in a elevator controller.
38       * Parsed message from MasterController and run tasks from
             a queue
39       */
40      @Override
41      public void run() {
42          try {
43              while (true) {
44                  while (inbox.isEmpty() && taskQueue.isEmpty()){
                        //While elevator can stay idle
45                      Thread.yield();
46                  }
47                  Message m = peekQueue();
                        //Looks at a the top task in the
                       elevators quere
48                  if (m != null) {
                        //If was a task in queue
49                      setTargetFloor(m.getTargetFloor());
                            //Set target to top task floor
50                      if (getFloor() >= getTargetFloor() - 0.05
                          && getFloor() <= getTargetFloor() +
                          0.05) { //If you already are on the
                          correct floor
51                          doorAction();
                                //Open the doors
52                          pollQueue();
                                //Remove the task from the queue
53                          setDirection(0);
                                //Set the elevator to not move
54                          continue;
55                      }
56                      decideMove();
                            //Get the elevator moving in the
                           correct direction
57                  }
58                  Message msg = pollMessage();
                        //Get message from mastercontroller
59                  if (msg != null) {
60                      switch (msg.getType()) {
                                //Get the type of the message
```

```
61                  case 'p':
                        //P-Message is a message with a move
                       to this floor order
62                      if (msg.getTargetFloor() == 32000) {
                            //Emergency stop
63                          addMessage(new Message('m',
                            getElevator(), 0, 0)); //Stop
                            elevator
64                          synchronized (this) {
65                              inbox.clear();
                                    //Clear both queues
66                              taskQueue.clear();
67                          }
68                          setDirection(0);
                                //Set direction to not move
69                          continue;
70                      }
71                      addQueue(msg);
                            //Add the message to the queue
72                      break;
73                  case 'f':
                        //F-Message information to the
                       controller about the elevators current
                       position
74                      if (getDirection() == 0)
                            //If the elevator have stopped
                           moving
75                          continue;
76                      setFloor(msg.getcurPos());
                            //Set the elevators position
77                      if (Math.abs(getFloor() -
                           getTargetFloor()) < 0.05) {    //If
                           the correct floor is reached
78                          addMessage(new Message('m',
                            getElevator(), 0, 0));  //Stop
                            the elevator
79                          doorAction();
                                                //Open and Close
                            the door
80                          pollQueue();
                                                //Remove the
                            task from the taskqueue
81                          setDirection(0);
                                                //Set the
                            direction to not moving
82                      }
```

```
83                          break;
84                      default:
85                          System.err.println("Unhandled
                               message.");
86                      }
87                  }
88                  decideMove();
                        //Get the elevator moving in the correct
                       direction
89                  Thread.yield();
                        //Yield the remaining timeslice
90              }
91          } catch (InterruptedException e) {
92              e.printStackTrace();
93          }
94      }
95
96      /**
97       * Adding a task to taskqueue sorted by prior order
            (earlies approach first)
98       * @param msg
99       */
100     public synchronized void addQueue(Message msg) {
101         System.out.println("Adding task: " + msg);
102
103         if (!(getDirection() * msg.getcurPos() < getDirection()
                        //Assume that the task is in the elevators
                   direction
104                 * getFloor())) {
105             for (int i = 0; i < taskQueue.size(); i++) {
                        //Sort in the task in correct position
106                 if (getDirection() * msg.getcurPos() <
                       getDirection()
107                         * taskQueue.get(i).getcurPos()) {
108                     taskQueue.add(i, msg);
                                //Add task to position i in the
                           queue
109                     return;
110                 }
111             }
112             taskQueue.add(msg);
                        //Add last in queue
113         }else{
114             System.err.println("The Elevator is not heading
                   that way, douche.");     //If a button push is in
                   the wrong direction
```

```
115             }
116         }
117 
118         /**
119          * Let the controller check the top prior task
120          * @return The Message in top of the taskqueue
121          */
122         public synchronized Message peekQueue() {
123             return taskQueue.isEmpty() ? null : taskQueue.get(0);
124         }
125 
126         /**
127          * Remove and returns the first element in the taskqueue
128          * @return the message
129          */
130         public synchronized Message pollQueue() {
131             if(taskQueue.size() == 1) setIntendedDirection(0);
132             return taskQueue.isEmpty() ? null : taskQueue.remove(0);
133         }
134 
135         /**
136          * Sets direction and sends a message to start move the
                elevator in the correct direction
137          */
138         private void decideMove() {
139             if (getDirection() == 0
140                     && Math.abs(getFloor() - getTargetFloor()) >
                            0.05) {
141                 System.err.println("DECIDE MOVE");
142                 int modifier;
143                 if (getFloor() < getTargetFloor() - 0.05) {
                      //If the elevator should move up
144                     System.err.println("Going up.");
145                     setDirection(1);
146                     modifier = 1;
147                 } else if (getFloor() > getTargetFloor() + 0.05) {
                      //If the elevator should move down
148                     System.err.println("Going down.");
149                     setDirection(-1);
150                     modifier = -1;
151                 } else {
                      //If the elevator is in the correct floor
152                     return;
153                 }
154                 Message msg1 = new Message('m', getElevator(),
                        modifier, 0);
```

```
155                 addMessage(msg1);
                        //Add the message to outbox
156             }
157         }
158
159         /**
160          * Send message to open the door, wait a sec, then close
                the doors again
161          * @throws InterruptedException If the sleep interrupts
162          */
163         private void doorAction() throws InterruptedException {
164             addMessage(new Message('d', getElevator(), 1, 0));
165             Thread.sleep(1000);
166             addMessage(new Message('d', getElevator(), -1, 0));
167             Thread.sleep(1000);
168         }
169
170
171
172         /**
173          * Below here is only getters and setters.
174          */
175
176         public synchronized Message retrieveMessage() {
177             return outbox.poll();
178         }
179
180         public synchronized void addMessage(Message msg) {
181             System.err.println("Adding message: " + msg);
182             outbox.add(msg);
183         }
184
185         public synchronized int getInboxSize(){
186             return inbox.size();
187         }
188
189         public synchronized int getTaskQueueSize(){
190             return taskQueue.size();
191         }
192
193         public synchronized void postMessage(Message msg) {
194             inbox.add(msg);
195         }
196
197         public synchronized Message pollMessage() {
198             return inbox.poll();
```

```
199     }
200
201     public synchronized double getFloor() {
202         return floor;
203     }
204
205     public synchronized void setFloor(double floor) {
206         this.floor = floor;
207     }
208
209     public synchronized int getDirection() {
210         return direction;
211     }
212
213     public synchronized void setDirection(int direction) {
214         // System.err.println("Setting direction to " +
              direction);
215         this.direction = direction;
216     }
217
218     public synchronized double getTargetFloor() {
219         return targetFloor;
220     }
221
222     public synchronized double setTargetFloor(double
          targetFloor) {
223         this.targetFloor = targetFloor;
224         return targetFloor;
225     }
226
227     public int getElevator() {
228         return elevator;
229     }
230
231     public synchronized int getIntendedDirection() {
232         return intendedDirection;
233     }
234
235     public synchronized void setIntendedDirection(int
          intendedDirection) {
236         this.intendedDirection = intendedDirection;
237     }
238
239 }
```

D Message.java

```
1  package controller;
2
3  /**
4   * Message used to communicate between threads.
5   * The class consists of getters and setters and overrides
       toString and equals for our convenience.
6   * @author Mattias Knutsson and Andreas Gustafsson
7   */
8  public class Message {
9
10     private final char type;                    //The type of the
         message (F,P,M etc.)
11     private final int elevator;                 //The ID of the
         elevator
12     private final int targetFloor;              //The target floor
13     private final double curPos;                //The current
         position
14
15     public Message(char type, int elevator, int targetFloor,
         double curPos) {
16         this.type = type;
17         this.elevator = elevator;
18         this.targetFloor = targetFloor;
19         this.curPos = curPos;
20     }
21
22     public char getType() {
23         return type;
24     }
25
26     public int getElevator() {
27         return elevator;
28     }
29
30     public double getTargetFloor() {
31         return targetFloor;
32     }
33
34     @Override
35     public boolean equals(Object obj) {
36         if (obj instanceof Message) {
37             Message m = (Message) obj;
38             return type == m.type && elevator == m.elevator
39                     && targetFloor == m.targetFloor;
```

```
40          }
41          return false;
42      }
43
44      @Override
45      public String toString() {
46          if(type == 'f')
47              return type + " " + elevator + " " + curPos;
48          else
49              return type + " " + elevator + " " + targetFloor;
50      }
51
52      public double getcurPos() {
53          return curPos;
54      }
55  }
```