Green Elevator

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

9 mars 2012

Innehåll

1 Assumptions & Simplifications **1**

2 Application **2**
2.1 MasterController.java 2
2.2 ElevatorController.java 2
2.3 Message.java 2

3 Algorithm **3**
3.1 Elevator selection 3
3.2 Floor scheduling 3

4 Implementation **4**
4.1 MasterController 4
4.1.1 run() 4
4.1.2 controlElevators() 4
4.2 Assigner 4
4.2.1 run() 4
4.3 ElevatorController 4
4.3.1 run() 5
4.3.2 addQueue() 5
4.3.3 decideMove() 5
4.3.4 doorAction() 5
4.3.5 Remaining methods in ElevatorController 6
4.4 Message 6
4.4.1 Methods of Message 6

5 Environments **7**
5.1 OS & Java 7
5.2 System Specifications 7

6 Achievments **8**

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 **??** for code.

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 anonomously 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 **??** for code.

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 recieve 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).

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 isempty.
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 emergencystop, 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 targetfloor, 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 synchronised getters/setters adders/pollers.

4.4 Message

Message is a multi-purposed message that is sent between the MasterController and ElevatorControllers.
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) CPUÛ7300 @ 1.30GHz (2 CPUs), ĩ.3GHz[3]
Memory: 4096MB RAM

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

6 Achievments

1. hej