A Multithreaded Controller for the "Green Elevator"

Mattias Cederlund, mcede@kth.se

Introduction

The goal of this programming project was to design, implement and evaluate a controller to the Elevators application. The controller should register changes from the elevators such as buttons being pressed and position changes. It should react to events by sending control commands to the elevators motors, doors and scales. Because you should be able to controll all elevators in “real-time” the controller should be implemented as a multithreaded application.

Algorithm description

A controller algorithm needed to be developed. It should provide fairness and good quality of service, which can be measured as a sum of waiting time and traveling time. The goal is to minimize that sum. The controller should also take power consumption in consideration.

In the implemented algorithm the elevator will only serve one direction a time, going from the lowest floor with a job scheduled to the highest before turning around. In this way we minimize the traveling distance and therefore also the power consumption. This will also minimize the average travel time for any stops scheduled from the inside buttons.

When scheduling a floor job (from button presses outside of the elevators) the main controller will ask each individual elevator to calculate the cost for the job and it will assign the job to the elevator with the lowest cost.

The cost is calculated as a sum of waiting time until the elevator reaches requested floor, the extra time the stop will take for the elevator and a sum of delays. If the elevator already is stopping at a floor the extra stopping time will be zero. The waiting time is calculated based on how many floors the elevator needs to move and how many stops there will be before reaching the requested floor. These values are then modified using the elevators velocity to get a value in time. To minimize travel time the sum of delays for jobs being postponed will be added to the cost as well.

I think that minimizing the travel time further is hard in this elevator model because it does not have any information on people using the elevators and which floor they are going to and from. Also it would be helpful to know to which floor someone who presses a floor button wishes to go to.

In another try to minimize power consumption, the controller multiplies the service time with the number of extra floors it has to pass. This will take service time / power consumption (generalized to extra travel distance) tradeoff into consideration, but might decrease service time performance a lot.

Implementation

The controller was implemented in Java (compliance level 1.7) using Java RMI for controlling the elevators. The development and tests was carried out in Eclipse running on Windows 7 with an 8-core processor.

The controller consists of three classes: Controller, ElevatorHandler and Job.

The Controller is the main controller that receives button events and assigns the jobs to the elevators. When starting the controller, it creates individual ElevatorHandlers for all elevators. After initiation it only listens to events from the buttons. If an inside button is pressed the controller will assign the job to the corresponding elevator. If a floor button is pressed it will ask all elevatorhandlers to calculate a cost for the job. The elevatorhandler with the lowest cost will be assigned the job.

The Job class is a container for information about a job. It contains destination floor and direction (if any). Only floor jobs gets assigned a direction because it is not important to stop in any specific direction when the job is scheduled from pressing a button inside of the elevator. The Job class also contains comparators to sort jobs by destination floor in both ascending and descending order.

The ElevatorHandler controls an individual elevator. It is running in an infinite loop that checks for and performs jobs and keeps track of the position of the elevator. Each elevatorhandler has its own list of jobs to perform and if there are no jobs in the joblist, the thread will wait for a condition variable to signal. The signal to wake up will be sent by the main controller when a new job is added to the list. Upon waking up the elevatorhandler will perform the first job in the joblist and continue onwards until the list is empty. If the elevator is at its destination, it will stop and open the doors before removing the job and marking it as finished, thus making sure the next job will be able to be performed.

Performing a job consists of the following tasks: closing the doors, start the elevator in the correct direction and wait for the elevator to reach the floor specified by the job.

Adding a new job is more tricky. First off the elevatorhandler checks for duplicates to the new job in its joblist. If the new job is not a duplicate, it will be added to the list. Then the list will be reordered (with the function prioritizeJobs) in an appropriate order to ensure that the elevator will only serve one direction at a time before turning around and serving the other. After reordering the joblist the condition variable is notified, waking the elevatorhandler if it is waiting for the condition. The current job will also be updated to point out the first job of the joblist.

PrioritizeJobs is the function that orders the jobs appropriately. This is done by grouping the jobs in three different groups. The first group is for jobs of current direction with destination floors the elevator has not already passed. The second group is for jobs of the next (opposite to the current) direction. The third group is for jobs of current direction with destination floors the elevator already passed. These three groups will be the only ones needed for any jobs of any direction at any time.

The ElevatorHandler also contains a cost function (with helper functions) that is used to calculate the cost for floor jobs. The cost function adds a new job to a copy of the joblist and orders it using prioritizeJobs. From this list the number of stops and travel distance to the new job is measured and modified with elevator speed to get the total waiting time.

If the job before or after the new job has the same destination floor as the new job, it is considered a duplicate job and the extra stopping time will be deducted from the result in an attempt to minimize the traveling time for the elevators. Also the sum of delays for jobs being postponed by adding a new job to the list will be added to the cost for the same reason.

Lastly the total time (waiting and traveling delay) will be multiplied with the number of extra floors the elevator needs to travel to make sure power consumption is minimized.

All functions and the whole codebase is properly commented for extra depth in implementation details.

Synchronization mechanisms

The only synchronization used is between the controller thread and the individual elevatorhandlers when a new job is added to the list. This is done with java.util.concurrent Condition, with the behavior of a condition variable. When wait is called, it will cause the elevatorhandlers to be blocked until they are signaled when needed. In this way no CPU time is wasted when the elevatorhandler has no jobs.

An improvement to the controller would be re-scheduling of floor jobs to optimize waiting time further. This would need extra synchronization between the elevatorhandlers.

Test cases and results

1. **1 elevator, 6 floors**
    1. **Guaranty of service: Press 3 up, panel 5, and then 3 down.**
       The elevator will serve floor 3 upwards, floor 5 and lastly floor 3 downwards. This result is expected by the algorithm.
    2. **Test of starvation: Press 3 up. When the elevator is in between the 2nd and the 3rd floor, press 2 up, 3 down, 2 down.**
       The elevator will stop at floor 3 and then at floor 2. It might seem weird with just two stops but if you look closer at the scheduling you find that the elevator stops at floor 3 up, 3 down, 2 down, 2 up. If there would be any button presses inside of the elevator for appropriate floors in between, the elevator actually world serve those before performing the next floor job.
    3. **Quality of service: Press 4 up, 3 up, 2 up**
       The elevator will stop at floors in order: 2, 3, 4. This result is expected by the algorithm.
    4. **Quality of service: Press 2 up, 3 up, 4 up**
       Same result as test case 1.3, as expected by the algorithm.

2. **2 elevators, 6 floors**
    1. **Press 3 up, 3 down**
       The first elevator serves floor 3 up and then floor 3 down. This is expected with the same explanation as test case 1.2.
    2. **Press 3 up. When the moving elevator is in between 1st and 2nd floor, press 1 up, 2 down.**
       The first elevator serves floor 3 up and the second serves floor 1 up and then 2 down. Here we have the same situation as for test case 1.2. If there would be an appropriate button pressed inside the elevator two in between floor 1 and 2, then it would be performing that job before the one on floor 2 down. Therefore this is also the expected result.

3. **3 elevators, 6 floors**
    1. **Guaranty of service: Press 3 up, panel 5 (elevator 1), and then 3 down.**
       The first elevator serves floor 3 up and floor 5. The second serves 3 down.
    2. **Test of starvation: Press 3 up. When the elevator is in between the 2nd and the 3rd floor, press 2 up, 3 down, 2 down.**
       Elevator 1 serves all the requests, just like 1.2.

3. **Quality of service: Press 4 up, 3 up, 2 up**
   The elevators serve one floor each. Because of the order the buttons are pressed all elevators will have the same distance to the requested floor, but the ones with a job assigned will have greater delay time. This will cause the controller to choose a different elevator for each job.
4. **Quality of service: Press 2 up, 3 up, 4 up**
   First elevators serves all floors. Because of the order the buttons are pressed the first elevator will always have a shorter extra distance and therefore be chosen.
5. **Press 3 up, 3 down**
   First elevator serves all floors. Same explanation as 2.1 and 1.2.
6. **Press 3 up. When the moving elevator is in between 1st and 2nd floor, press 1 up, 2 down.**
   Same result and explanation as 2.2.

One may think that the results for test cases 3.x do not follow the algorithm but in fact they all do. Although the extra distance (power consumption) might have too big of an impact on minimizing the waiting and traveling time. The algorithm returns the best time per power consumption correctly, but it is not so intuitive.

Summary and conclusions

The developed controller does meet the quality of service requirements, but it is not always easy to realize just by examining the elevator movements. To get the complete picture you also need to read the controller printouts.

The main problem I had with the project was in the setup phase because of incomplete instructions of starting the rmiregistry. I actually had to start it in a specific directory (elevator/classes) to get it working.

I think this project has been good for showing how multithreaded applications can be used to split the operation of a system into individual but yet centralized pieces. Although I think that for the sake of learning multithreaded development a project based on optimization for multicore processors would be better suited.