2014

Green elevator project

Mahboobeh AbdalMahmoodAbadi & Mehran Nasseri
KTH
3/15/2014

1. Description of the application

The program implements an elevator controller using multithreaded in order to control the elevator in real time. The implemented controller accepts commands from the elevators and responds to them by forwarding the commands to the elevators' motor and doors.

The controller has one master thread, which is responsible for reading the commands from an input stream and putting them in the queue of the corresponding elevator thread that should execute that command. In addition, the master thread is responsible for finding the best and closest elevator to give service.
Furthermore, the controller has number of worker threads (i.e. one worker thread per each elevator) that are responsible for executing the commands forwarded by the master thread. Controller establishes TCP connection with ElevatorIO only for receiving messages (on port 4712). The class OutputThread is responsible to establish TCP connection with the ElevatorIO only for sending messages (on port 4711).

The algorithm implemented in a way that the best solution for choosing the closest elevator is the one, which is idle, and exactly in the same floor as the calling floor. Furthermore, In order to reduce waiting time and energy consumption these parameters are considered: moving direction and current position of the elevator as well as requested floor to be serviced and the intended direction of travelling. In order to prevent starvation and fair scheduling, the method toService() and rearrangeStopList() are defined in the program. These methods are responsible for adding the floor to be serviced in appropriate queuing list to schedule service.

2. programming environments

This program is implemented using JAVA socket API to handle communication between processes by sending and receiving commands. Each command will cause to invoke appropriate method to execute tasks. The program developed on the Windows 7, Intel Core i5-2.5GHz CPU using Netbeans 7.3.1 IDE.

3. Description of implementation.

3.1. Controller class

Creates new instance of the OutputThread, which is a thread responsible to establish a connection to ElevatorIO class (i.e. to send messages to ElevatorIO class). With the initialization of the Controller the following tasks are performed:

- Connection to the TCP socket is established.
- Ask OutputThread class to send info messages to ElevatorIO class.
- Controller receives the info, which includes the number of floors and elevators.
- Initiate the master thread and elevator threads.

3.2. MasterThread class

The MasterThread reads commands from input stream. In case of command "b" which identifies the request from a floor to go up/down, the MasterThread is responsible to find the best and closest elevator by invoking the method callElevator() to provide service. Furthermore, the MasterThread put the other full commands in the command queue of the corresponding elevator thread to execute that command.

How to find the closest elevator: callElevator() method:

- If there is only one elevator, it just returns 1.
- If there are more than one elevators the algorithm works as follow:
  - **Highest priority:** finds the one which is idle elevator and exactly in the same floor as the calling floor.
  - Otherwise, **second priority:** choose among the moving elevators. In order to reduce waiting time, energy consumption the following options are discarded:
    - Not moving elevators
    - Elevators that are moving in the opposite direction of the request
    - Elevator that is in higher floor and moves toward up
    - Elevator that is in lower floor and moves toward down

    For the remaining elevators, check which one is the closest one.
  - Otherwise, **third priority:** find the closest idle one.
  - **Last priority:** If the above algorithms failed, we have to choose one elevator randomly.

3.3. ElevWorker and ElevatorPlanner class

In a while loop, the ElevWorker takes commands from the command queue to execute a task.

The ElevWorker sends commands to ElevatorIO class through TCP socket created in the OutputThread class.

In case of command "b" (forwarded from MasterThread), the ElevWorker checks the situation of the elevator. If it is idle and in the same floor as the calling floor, just open and close the door. Otherwise, it invokes the method toService() in ElevatorPlanner class. The toService() method is responsible for adding the floor to the list of stops if the elevator is idle or going towards that floor. It means it does not pass the floor yet. If it has passed the floor, the floor number will be added to the waiting list instead to be serviced later.

In case of command "P" (forwarded from MasterThread), if p==32000 all states of the elevator will be reset and the elevator will be stop. Otherwise, it will go to the floor that the passenger inside the elevator wants to. The button number will be added to the toStopDown or toStopUp list depending on the current position of the elevator and the requested floor.

In case of command "d", the door of the elevator number x will be open if the command be 'd elevator number 1' and the door of the elevator number x will be close if the command be 'd elevator number -1'.

In case of command "f" which is sent continuously from ElevatorIO class, the exact position of the elevator is represented.

- If the current Position is not equal to the floor it aims to go, the elevator continues moving.
- If we are at the requested floor, the elevator stops by sending the 'command "m", elevator number, and 0' to the ElevatorIO class. Then, the door will be opened and closed and this floor number will be removed from toStopUp / toStopDown / waitingListUp / waitingListDown list by calling rearrangeStopList () in ElevatorPlanner class. At this point the method rearrangeStopList () acts as follow.

- rearrangeStopList() method
  - When stopping on a floor, that floor will be removed from all lists. For example current direction of the elevator is toward up, If there is a request to go down on that floor, it is added to the waitingListDown list by the toService() method. In order to save time and energy, it is assumed that the person who wants to go down can come to the elevator as the elevator stopped on that floor even though he/she wants to go down. In this way, the waiting time can be improved as extra time for one more stop and open/close door can be saved.

- Since we have two different stop list (toStopUp and toStopDown), after stop if there is no more request on the list of current direction (e.g. toStopUp) it will switch to service the other list by inserting element from the waiting list. For example, if no further toStopUp left, it will travel down by adding the element from the waitingListDown to the toStopDown list. If there is no request in the queue for going down, add from the waitingListUp to the toStopUp if any. This order of checking is for ensuring the fairness in serving both up and down if any.

- toService() method
  When up/down is pressed, the method will add the floor number to one of the service queues considering the following situation:
  - If the elevator is idle, the floor number will be added to toStopUp or toStopDown lists depends on which button down/up has been pressed.
  - If the elevator is not idle:
    - If elevator already passed the floor, it will be added to the waitingListUp or waitingListDown. These waiting lists will be served later as described above. Without these two waiting lists, assume that we have always some call toward up so the elevator will never service those who wants to go down as the toStopUp list always has some element and we can never switch to toStopDown list.
    - Otherwise, we can safely add it to the toStopDown or toStopUp lists.

  The way that these four lists managed by toService() method and rearrangStopList() , we can ensure the fairness of service and reduce the waiting time as much as possible.
- nextFloor() method
  This method simply returns the first element of toStopDown or toStopUp lists depends on current moving direction.

To sum up, these three methods together can prevent starvation and unnecessary back and forth and reduce the waiting time and improve fairness because:

- toStopDown and toStopUp lists are sorted so the floors will be serviced in order of the number on the way up or down. Therefore, it will never happen the elevator pass a floor that request for a service and back for it again.
- In addition, the way that toService() method and rearrangStopList() method are organizing element insertion, the fairness will be ensured.
- The nextFloor() method returns the first element of the list and decision of giving service to that floor will never be changed to back and service another floor that is already passed. This ensures the starvation free algorithm.

4. Test use cases

4.1. Run GE with 1 and 3 elevator, 5 floors:

- **Preventing starvation:** The starvation test passed successfully as it is expected. The algorithm for preventing starvation has been explained in the previous section**.** The way that we organized the list of stops in rearrangeStopList() and toService() method help to prevent starvation.
- **Quality of service:** when pressing 4 up, 3 up, 2up (or in reverse order), the elevator will stop at the second floor and then third and fourth floor. This order can reduce the waiting time. For example, since the indicated direction is toward up, person in second floor might want to get off at third or fourth floor. Therefore, it is better to tale him on the way up to prevent unnecessarily back and forth and reduce the waiting time.
  On the other hand, if you try 4 down, 3 down, 2 down (or in reverse order), the elevator stop at fourth floor first and then third and second floor. The same assumption as above is considered to prevent unnecessarily back and forth and reduce the waiting time. This behavior is result of the algorithm implemented by the toService() method. Also in case of three elevators only one elevator will service all three floors as a result of priority options considered in callElevator () method.

4.2. Run GE with 2 and 3 elevator, 5 floors:

- **Press 3 up, 3 down**: both elevators will service that floor because the intended directions are opposite. If one elevator to be sent to give service so, the waiting time will be increase.
- **Press 3 up. When the moving elevator is in between 1st and 2nd floor, press 1 up, 2 down:** one elevator will be sent to the third floor and the other goes to the first and one of them will be selected randomly to go to the second floor. This is due to the behavior of the algorithm implemented in callElevator() method. According to the algorithm, any of the two elevators have same priority to go to 3 up. When 1 up command issued, the elevator that is going to the third floor is worth option since it not only has passed floor one but also considering the passenger in third floor wants to go further up, the passenger in floor must wait long time to get the service. So, the other one will be selected. When it comes to select an elevator to go to the second floor, there is no good option. So, we choose randomly. We cannot say which elevator could provide less waiting time because the waiting time depends on which upper floor the passenger, from first and third floor, wants to go.
  However, if we initialize three elevators one will service the 1 up and the other one will service the 2 down. This is because of the assumption of waiting time raise when two opposite call have to be serviced by one elevator.

5. Reflection

At the first look, the assignment seemed too simple and easy but after going deep in to the problem we considered that the elevator problem can be complicated and many different use cases and algorithm must be considered to achieve a fair and fast solution. We became more interested in this problem as it can provide the basic idea for scheduling read and write from hard driver. The course helped us to organize our understanding about synchronization mechanisms and concurrent computing both practically and theoretically as it was ambiguous and difficult to understand in the past.