Lab 1 Elevator Scheduler

2014CST 2014050693 贺嘉允
International School, Jinan University

Abstract

Suppose we need five elevators in a 20-floor building, and a control system for these elevators are required. For each elevator, there should be some buttons and a screen is needed to show the current status of the elevator. For each floor, an up and down button is required. All the elevators are initially at the first floor. Most importantly, a proper scheduling algorithm must be implemented to make sure the elevators answer requests from inside and outside the elevator in a reasonable manner.

1. Introduction

To ensure the requests can be made, the button inside each elevator and on each floor must be specified. For each elevator, Each elevator: there should be some buttons: the numeric keys, opened the door, closed key, up key, down key, alarm key, ascending signal, descending signal etc. There should be a digital display to indicate the current state of the elevator; and for each floor, there should be up and down buttons and digital display the current state of the elevator. Additionally, five of elevator door button is interconnected, five elevator door button interconnection junction, that is, when an elevator button is pressed the corresponding button of the other elevator also lit at the same time pressed. Scheduling algorithm is based on the state of the five elevators, looking for the nearest and request the same direction of the current floor elevator and give a response.

2. Scheduling Algorithms

To handle different kind of requests, I first divide the request into two parts: internal request and external request. The former one means the request from inside the elevator, and the latter one means the request from the floors.

In the *elevator* class, I used two lists to store the two kind of requests. Since the request inside the elevator are likely to be handled first in reality, in the *run()* method implemented in the *elevator* class, the elevator will first check the internal request list. If the list is not empty, take the first element in the list as the temporary request and set it as a target floor. If the internal request list is empty, the system will then check whether the external request list is empty.

```
//      System.out.println("elevator" + (elevatorNum+1) + " is running");
        while(true){
            direction.setText("-");
            while(!internalRequest.isEmpty() || !externalRequest.isEmpty()){
//              System.out.println("EL: " + elevatorNum + "current floor:" + currentFloor);
                //电梯内请求
                if(!internalRequest.isEmpty())
                    tempRequest = internalRequest.pop();
                else if(!externalRequest.isEmpty())
                    tempRequest = externalRequest.pop().requestFloor;

                directionFeedback(tempRequest);
                setDirectionLabel();
```

Figure 1. Algorithm for handling inside and outside request in a single elevator

The function using for going upward are shown as below. The going downward function is similar to this. The function will not have a target floor as an input but I set it to move upward one floor each time. This is helpful when the elevator have external request in the same direction (upward for instance) and can take the request during the going upward process.

```
public void goingUpward(){
    System.out.println("Floor Request up:" + floorRequest);
    //填充当前楼层为白色
    floorLabel[numberOfFloor - currentFloor].setBackground(Color.whi
    //填充下一楼层为褐色
    floorLabel[numberOfFloor - currentFloor - 1].setBackground(Color
    floor.setText(String.valueOf(++currentFloor));
    try{
        Thread.sleep(800);
    }
    catch (InterruptedException e){
        e.printStackTrace();
    }
```

Figure 2. The goingUpward function

To answer external request while the elevator is moving, the elevator will judge whether there is an external request exists (distributed to this elevator) before going up or down one floor.

```
if(currentStatus == "Up"){
    System.out.println("tempRequest:" + tempRequest + " tempDistance" +
    for(int i = 0; i < tempDistance ; i++){
        if(!externalRequest.isEmpty()){
            System.out.println("exRequest:" + externalRequest.peek().re
            if(externalRequest.peek().requestFloor < tempRequest
                    && currentFloor == externalRequest.peek().requestFl
                    && externalRequest.peek().direction == currentStatu
                externalRequest.pop();
```

Figure 3. Algorithm for handling same-direction request (Going upward for instance)

We now have algorithms for a single elevator to deal with the request, and an algorithm for distributing external request to these elevators is needed. To realize this, I separate all the elevator to two states, one for those are moving and one for those have already done with requests and are currently stop. After the button on a specific floor is pushed, the corresponding floor and direction will send to the *locateElevator* function through the action Listener method. And then the function will check all the distance of the elevators and find the minimal one for the moving elevator and the motionless elevator. And then, the corresponding index of the elevator located will be returned by this function.

```
public int locateElevator(FloorButton flBtn){
    int[] motionlessDistance = new int[numberOfElevator];
    int[] movingDistance = new int[numberOfElevator];
    int motionlessOptimal = 20, movingOptimal = 20;
    int motionlessIndex = 4, movingIndex = 4;
    for(int i = 0; i < numberOfElevator; i++){
        if(elevatorArr[i].currentStatus == "Stop"){
            motionlessDistance[i] = Math.abs(flBtn.index - elevatorArr[i].currentFloor);
            System.out.printf("Motionless Distance[%d] = %d\n",i,motionlessDistance[i]);
            if(motionlessDistance[i] < motionlessOptimal){
                motionlessOptimal = motionlessDistance[i];
                motionlessIndex = i;
                System.out.printf("motionlessOptimal: %d, motionlessIndex: %d\n", motionlessOptimal,motionlessIndex);
            }
        }
        else if(elevatorArr[i].currentStatus == flBtn.direction){
            movingDistance[i] = Math.abs(flBtn.index - elevatorArr[i].currentFloor);
            if(movingDistance[i] < movingOptimal){
                movingOptimal = movingDistance[i];
                movingIndex = i;
                System.out.printf("movingOptimal: %d, movingIndex: %d\n",movingOptimal,movingIndex);
            }
        }
    }
    //如果相等，优先选择动的的电梯
    if(motionlessOptimal < movingOptimal)
        return motionlessIndex;
```

Figure 4. Algorithm for distributing external request (request on floors) to the nearest elevator in the same direction

3. GUI Specifications

The GUI is divided into two *JFrames*, one is for the elevator control and one for live demonstration. The control panel is consists of two parts, the buttons inside each elevator, which includes the floor buttons and an emergency button, and the main panel for demonstration, including the moving of each elevator and the up and down buttons on each floor.

In the control panel, the current status of the corresponding elevator will be displayed, including the current floor and the current status. The buttons will lights in white color to represent the inner request from the elevator.

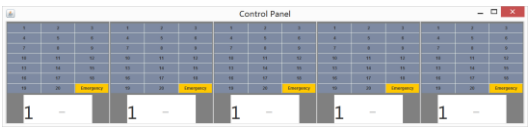

Figure 5. The elevator control panel

While in the main demonstration panel, the five columns represents the status of the corresponding elevator. And the ascending as well as the descending will be demonstrate as the orange part (represents the current floor) changing its location. And, in this panel, the up and down button can be pushed to generate request, and the algorithm mentioned before will distribute a proper elevator to answer the request.

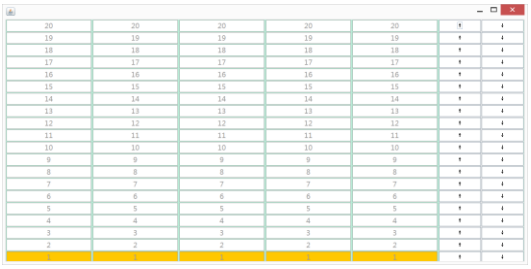

Figure 6. The main demonstration panel

To complete the composition, the *GridLayout* layout manager is most often used method. The button inside the elevator panel, the elevators and the floors are composed with this layout manager. Additionally, the *Border-Layout* manager is also used for com-position.

4. Multithreading

Since there are five elevators, we need to generate five threads. The *elevator* class implements the *Runnable* class, and the *run()* method is specified in this class. To start five threads, the corresponding part of code is shown:

```
for(int i = 0; i < numberOfElevator; i++)
    new Thread(elevatorArr[i]).start();
```

The scheduling algorithm in part 2 ensures that there will possibly not be any deadlocks. Each request, either is external request or internal request, will be pop out of the stack once corresponding elevator decide to answer the request, so that the elevator will not run for the same request and crash the program.

5. Result and Analysis

When we run the elevator scheduler simulator, all the elevators are initially at the first floor, as shown in figure 6. We first try whether the elevator will stop when we first request floor 20 inside the elevator (internal request) and then push the up button at floor 12. The demo video showed that the elevator will first stop at floor 12 to answer the same-direction request and then go to the target floor – the 20th floor.

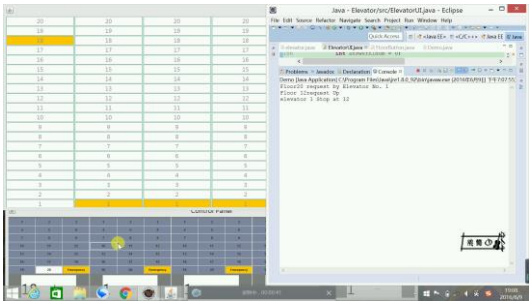

Figure 7. Taking on-route request while going upward

Since the internal requests and the external requests are stored in two different stacks, and we have a proper algorithm to deal with these two kinds of request (mentioned in part 2), so the elevators is capable of handling all the corresponding requests.

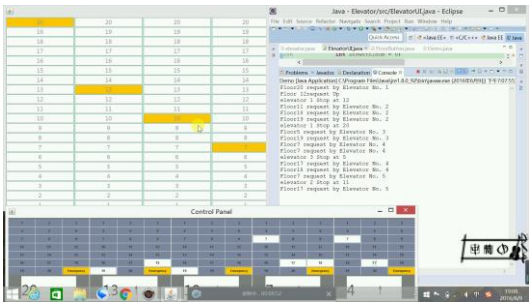

Figure 8. Taking internal and external requests

Plus, if the elevator have on-route request to take, the elevator will stop at the request floor and the corresponding feedback information will be displayed at the CLI (*Command-Line Interface*). Additionally, only the on-route request floor will be displayed while the elevator moving, which means the “final destination” will not be displayed, but can viewed directly on the demonstration panel, since the corresponding elevator has already stopped.

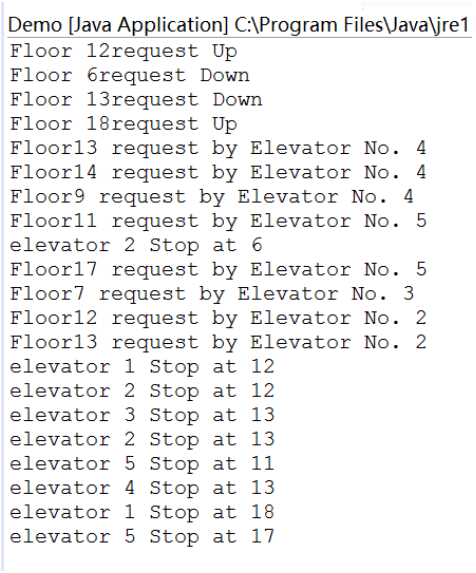

Figure 9. Corresponding feedback information on the CLI (Command Line Interface)

6. Conclusion

Planning ahead and have a clear design of the system before coding is very helpful. When coding the GUI, I need to consider how to let the button give the feedback information, so I extend two classes out of the *JButton* class, which is the *ElevatorButton* class and the *FloorButton* class. They have different functionalities and different properties so that the *e.getSource()* can return the needed object for further using. In addition, the color of the GUI is also one of the considerations. I went to Adobe's Kuler to have get a better color scheme for the GUI. Deep down to the algorithms of the elevators, all the requests must be taken into consideration in the designing stage. When debugging the program, try to simulate the real-life request and see whether some new problems arises.

This elevator scheduler is not perfect for sure. Here are one possible improvement:
To make it more practical, the load of each elevator can be take into consideration and the scheduler will not distribute request to an overloaded or full-loaded elevator.

Overall, designing such a scheduler required foreseeing problems and account all possibilities during the first stage. This will help in the later stages and have fewer re-doing works.