Project

ETH Zurich

Hand-out: 17 March 2011
Team registration: 24 March 2011
Due: 10 May 2011

1 Synopsis

You work for the Isis elevator company, developing controls systems. You have received a contract to build a control system for the elevator system in a new building. To become more familiar with the project, you will be building two prototype implementations in two different models (Java and SCOOP). In each language you will build a simulator which will receive elevator requests and will decide how to react to these events.

2 Requirements

At its core, the problem is to receive events, and then decide based on the system state appropriate actions in response to these events. The actions are commands dispatched to concurrently operating elevators.

2.1 Events

A request consists of two integers, separated by a single dash. The request represents a passenger's current floor and their target floor:

$$Request ::= Int - Int$$

Next to request events, there is a *Done* event which signals the last event. Hence, the set of all possible events is:

$$Event ::= Done \mid Request$$

Every input to the system will be of the form of *Event*. An example input may be:

```
3-2
5-1
Done
```

This input first has a request for a trip from floor 3 to floor 2. After some indeterminate amount of time, there is another request to go from floor 5 to floor 1. Finally the end of the events is signaled with *Done*.

2.2 Elevator Requirements

When invoked, the program will accept three command line arguments in order: n, *lower*, and *upper*. There are n elevators in the system, where $n > 0$ and the floors range from *lower* to *upper*, where $lower \leq 0$ and $upper \geq 0$. All elevators are initially at floor 0.

There must be a one-to-one relationship between the elevators and the unit of control (thread or processor). Additionally, the control system must have its own unit of control.

The elevator is governed by the following behaviors:

- The elevator travels at $1m/s$.
- The floors are $2m$ apart.
- The elevator experiences instant acceleration (poor passengers).
- An elevator stop lasts $1s$.

A passenger will enter an elevator as soon as one stops at their floor. Additionally, a passenger will only exit the elevator at their destination floor.

After a passenger made a request, the elevator must eventually visit the floor from which the passenger made the request from. After the passenger has entered the elevator, the elevator must eventually visit the passenger's destination-floor.

The system receives events through the standard input. Every line of standard input will correspond to a single event. The input is as described in the request language. The *Done* event signals the last external input. After the *Done* event has been received, and the final passenger in the system has reached their destination, then the simulation can safely exit.

The total waiting time is the time between when a passenger indicates a request and when they arrive at their destination. An efficient implementation may also try to minimize the average total waiting time of passengers.

2.3 Visualization Information

The visualization does not have to be real-time with respect to the input. It can be constructed after the *Done* event has been sent.

One simple way to construct the animated visualization can be using a QML as part of the Qt SDK. It allows simple descriptive explanations of moving shapes, and is suitable for displaying the behavior of the elevator system. An introduction to QML can be found at `http://doc.qt.nokia.com/4.7-snapshot/qdeclarativeintroduction.html`.

Other approaches to visualization are also acceptable, including using other formats (animated PNG, Flash, etc), or even a real-time visualization using the appropriate graphical library (Swing, EiffelVision2, OpenGL, etc).

3 Deliverables

3.1 Report (18 points)

The project report must include the following elements:

- An overview of the design of both the SCOOP and Java implementations of the system. This should include a high-level explanation of how SCOOP and Java were used to ensure correctness in the face of concurrency. (8 points)
- An explanation of how you attempted to minimize some or all of the waiting times in the system. If you run into troubles, explain the difficulties. (3 points)

- A detailed comparison of how SCOOP and Java concurrency shaped your design. Provide an analysis of which aspects of your solution each language handled elegantly or effectively, and also which aspects of your solution were difficult to express in each language. (4 points)

The report will also be expected to have a clear structure and proper grammar and spelling. (3 points)

3.2 Design and Implementation (32 points)

Two implementations are required: one in Java (using Java Threads) and the other in SCOOP. Design and implementation of both solutions should adhere to the requirements and address the problem of minimizing total waiting times (14 points for the design and implementation in each language, 4 points for the visualization).

4 Teams

You can work in teams of up to three persons. Please send an e-mail to one of the assistants (benjamin.morandi@inf.ethz.ch, scott.west@inf.ethz.ch) to register your team. The due date for the registration is indicated at the top of this assignment.

5 Support

You have the possibility to ask questions during the exercise sessions. If you need immediate help, you can also contact the assistants directly.

6 Submission

Please submit a zip file that contains the source code and the report in PDF format by e-mail until the due date. The SCOOP submission should be *without* the EIFGENs directory (which contains compiler-generated files).

Your zip file should contain two folders: a *source* folder for the source code and a *documentation* folder for the report. The layout of the source-code directory should contain two subdirectories, one for each of the SCOOP and Java code-bases.