CptS 483/580 Concurrent Programming
Assignment: Lift Modeling

Joe Armstrong,
Christian Schulte
Seif Haridi
Carl Hauser

March 12, 2014

Due date: Monday April 7, 2013 11:59 PM

This lab assignment uses agents for simulating a simple lift (also known as "elevator" scenario. It exercises modeling with agents, concurrency, and message sending. Don't be too intimidated by this project: the individual agents are not all that complicated although there are several of them and care must be taken with each one to fully understand the messages it must process and state changes it must make.

I suggest an approach of attacking each agent type, one by one, understanding what it must do, coding it, and testing it in isolation (build a testing skeleton for each one – you can include this in the program file with the agent code).

The skeleton files referred to below are all contained in the file
`http://www.eecs.wsu.edu/~hauser/cs580/handouts/lifts.tgz`
(and also `.zip`). In addition to the places in the files explicitly marked for you to add code you may add or change anything else in the files (and will probably have to!). Files ***`agent.erl`***, ***`simulation.erl`*** and ***`cabin.erl`*** do not require any changes. Note also that the skeletons may not completely specify the contents of the state for all the agents – it's just an initial suggestion.

Submit your code as a zip file by the due date to the Lifts Assignment dropbox on Angel.

The Lift Scenario

Assume we have a building with n lifts and m floors. When a lift is called, we wish to service the request in a reasonable manner: the lift will be selected that can service the request as soon as possible.

Imagine for the time being a building with three lifts ($n = 3$) and five ($m = 5$) floors. Floors one and five have a single call button, the remaining floors have two call buttons each for up and down respectively.

A lift is requested by pressing one of the buttons. A request is served by selecting a lift that stops at the floor. In addition to that, each lift has five buttons (or, in general m buttons) inside labeled with the floor numbers. These buttons will stop the lift at the appropriate floor.

Our task is now to write a program that controls the lift system. We will construct a solution that models lifts and floors by agents that react to incoming messages and possibly send messages to other agents. We model each problem agent by one program agent.

Scheduling Lifts

Suppose the up button is pressed on the fourth floor. We have to decide which lift should stop at the fourth floor. The following algorithm is one possibility:

- The agent for floor 4 sends a message to each of the lift agents asking them how long it would take them to get to floor 4.
- Each lift receiving this message works out how long it would take for it to get to floor 4, given its current workload, and replies with a message containing the time (the *waittime*).
- The floor agent chooses the lift with the minimal proposed waittime and replies with a `reserve` message. All other lifts are informed by a `reject` message.

Initially, assume that a lift agent is locked after it has sent a time estimate and remains locked until it receives either a `reserve` or `reject` message (that is it will only accept a `reserve` or `reject` message when in this state). This strategy works and does not lead to deadlocks if all request messages sent to lifts are received by the lifts in the same order. Why? Does your implementation meet the requirement that all request messages sent to lifts are received by the lifts in the same order? If not, you'll have to rethink this design.

Modeling

We model the scenario with three different types of agents which include of course the lift and floor agents mentioned above together with agents who model the actual lift cabins' behavior.

The Floor Agent. The only message a floor agent receives, initially, is that a button has been pressed. This is modeled by either a message `up` or `down` depending on which button has been pressed. When a button press is received, the best lift is selected as described above.

Initially, the state of a floor agent contains its floor number and a list of all lift agents. The state is encoded as tuple `{N, Lifts)` where `N` is the floor number and `Lifts` is the list of lift agents.
Extra credit: once you have the basic logic working, extend the state of the floor agent to include lights on the buttons and to accept arrival messages to know when the button lights should be turned off.

The Lift Agent. A lift agent understands the following messages:

- `{request, FloorNum, FloorAgent, Direction}` which is sent by the floor agent for floor `FloorNum`. `Direction` is either up or down depending on the button pressed at floor `FloorNum`.

  The lift agent responds with a tuple `{propose, T}` where `T` is the waittime.

  The floor agent responds with either `reject` or `reserve`. The best lift agent gets a `reserve` while the others get a `reject`. If the lift agent gets a `reserve`, it must include the floor number `FloorNum` in the floors it must visit (the list of floors to be visited is called *StopList* – see below for the description of the Lift Agent state).

- `{arrived, FloorNum}` is sent by the cabin agent when the cabin has reached floor `FloorNum`. The lift agent now must determine which action the cabin agent takes next by sending to the cabin agent either:

  1. `stop`: the cabin will open and close the doors. This takes five time units.
  2. `up`: the cabin moves up one floor. This takes one time unit.
  3. `down`: the cabin moves down one floor. This takes one time unit.
  4. `wait`: the cabin stays where it is for one time unit.

  Notice that all of the time delays are implemented by the cabin agent, not the lift agent. After the cabin has executed its action, it sends another `arrived` message to the lift agent.

- `{stop, FloorNum}` requests the lift to stop at floor `FloorNum` (coming from the buttons inside the lift).

  The state of the lift agent is a tuple `{Now, StopList}` where `Now` is the floor where the lift agent is currently located (this is known from messages sent by the cabin agent) and `StopList` is the stoplist. (You may find it useful to add a variable bound to either the atom `goingUp` or the atom `goingDown` to keep track of the lift's current direction of travel.) The StopList is a list of integers describing at which floors the lift agent must stop. Initially, the StopList is empty. Floor numbers are inserted into the StopList whenever a lift agent has received a request message and its offer has been accepted (with a reserve message) as being the best. Closely related to managing the stoplist is computing the waittime. The waittime is determined by where the floor would be inserted into the stoplist and considering the times that the cabin agent will pause the lift for each stop as well as the times taken to travel between floors.

1 Agents With State

We use the `agent` module to create agents. It implements an agent as a process that serves incoming messages.

```
-module(agent).
-export([newAgent/2]).
newAgent(Update, InitState) ->
    spawn(fun() -> serve(Update, InitState) end).
serve(Update, InitState) ->
    receive
        Message -> serve(Update, Update(Message, InitState))
    end.
```

Here the behavior of an agent is given by an arity-2 function `Update`, taking a `Message` and an `InitState` as input and returning a new state. This code is available as `agent.erl`.

A Simple Memory Cell Agent. As an example of a simple agent consider the memory cell programmed as follows:

```
-module(memoryCell).
-export([newMemoryCell/1]).
memoryCell({read, Requestor}, State) ->
     Requestor ! State,
     State
;
memoryCell({write, NewVal}, _State) -> NewVal
;
memoryCell({swap, Requestor, NewVal}, State) ->
    Requestor ! State,
    NewVal
.
newMemoryCell(InitVal) ->
    agent:newAgent(fun memoryCell/2, InitVal)
.
```

The state is implemented as a single value.

The code for the memory cell agent is in `memoryCell.erl`. It is only an example, not part of the programming assignment.

2 Inserting Floors into the Stoplist

Implement a function `insert(Stoplist, Now, Dir, Floor}` that returns a new stoplist. Here `Stoplist` is the current stoplist, `Now` is the floor where the lift agent is currently located, `Dir` is either `up` or `down` depending on which button has been pressed at floor `Floor`.

It is important to compute a new stoplist that minimizes time until the lift arrives at the desired floor. Take the following possibilities into account:

- `Floor` is already contained in the stoplist.
- The lift is already at `Floor`.
- If the lift moves from floor `A` to `B` with `A<Floor<B` and `Dir` is up, insert between `A` and `B`.
- If the lift moves down from floor `A` to `B` with `A>Floor>B` and `Dir` is down, insert between `A` and `B`.
- Insert `Floor` as much to the front of the stoplist as possible while following the above rules.

For example,

```
insert([], 3, up, 6)
```

returns `[6]`, whereas

```
insert([5, 7, 9, 3], 4, up, 6)
```

returns `[5, 6, 7, 9, 3]`, and

```
insert([5, 7, 9, 3], 4, down, 6)
```

returns `[5, 7, 9, 6, 3]`.

Store the function in the file `liftUtils.erl`.

Hint. In order to test your scenario you might want to consider a simple implementation of `insert` first.

3 Computing the Waittime

Implement a function `waitTime(Stoplist, Now, Dir, Floor}` that returns the waittime before a request can be served. Here `Stoplist` is the current stoplist, `Now` is where the lift is currently located, `Dir` is either up or down depending on which button has been pressed at floor `Floor`.

As mentioned above, stopping the lift takes five time units whereas moving the lift by one floor takes one time unit. The waittime must be computed with the same rules as for insertion into the stoplist.

For example,

```
waitTime([5, 7, 9, 3], 4, up, 6)
```

returns `7`, and

```
waitTime([5, 7, 9, 3], 4, down, 6)
```

returns `23`.

Store the function in the file `liftUtils.erl`.

4 Pressing a Lift Button

Implement a function `stopAt(Stoplist, Now, Floor)` that returns a new stoplist when a button for `Floor` is pressed inside the lift. Here `Stoplist` is the current stoplist, and `Now` is the floor where the lift agent is currently located. Follow the same rules to insert `Floor` into the stoplist as for Insert.

Store the function in the file `liftUtils.erl`.

5 The Lift Agent

Implement a function `liftProcess(Message, State)` for the lift agent which returns a new state.

The file `lift.erl` already contains code fragments to give you a start.

6 The Floor Agent

Implement a function `floor(Message, State)` for the floor agent which returns a new state.

The code for selecting the lift with shortest waittime (given that you know the waittime for each lift) is outlined in the **`selectBest`** function in this file.

The file `floor.erl` contains code fragments to give you a start.

7 Putting Everything Together

The provided files `simulation.erl` and `cabin.erl` contain the remainder of the simulation. The file `simulation.erl` creates a complete lift simulation scenario for a given number of floors and lifts. Feed the following to erl:

```
S = simulation:newScenario(3,5). % creates the simulation
% now you can feed other statements like those found at the bottom of
% simulation.erl to drive your lifts
```

You can simulate pressing buttons by sending the appropriate messages to the lift or floor agents. simulation.erl and cabin.erl in the .tar file provide a GUI for observing the behavior of the lifts. It should work on either Windows or Linux.

Acknowledgments

The problem has been taken from: Joe Armstrong, Robert Virding, Claes WikstrÄom, and Mike Williams, Concurrent Programming in Erlang, second edition. Prentice Hall, London, UK, 1996, and adapted for Oz by Christian Schulte, re-adapted for Erlang by Carl Hauser.