Week 3 - Multi-agent Systems

Jasper Linmans [10249060], Jasper Driessens [11349026],
Diede Rusticus [10909486]

February 2017

Introduction

This week's goal was to implement a system in which the buses communicate to make decisions. We decided to use communication to update the belief of buses concerning the busyness of the track they're driving on (the tracks are still fixed schedules, like last week).

At any moment, a bus can determine it is *overloaded*. This is a local notion of busyness. When this happens, the bus would like to buy a new bus for the track it's driving on. However, it does not know the global busyness of the track. i.e., there might be other buses on the track that don't consider themselves overloaded. In the latter case, a new bus should not be bought (yet).

In order to determine this, we have implemented *protocols*, principled ways for buses to communicate. Implementation-wise, for each protocol there is a *lead procedure* and a *follow procedure* defined. The lead procedure determines whether a bus should take action on a protocol proactively and is called at each tick. The follow procedure is called whenever a bus receives a message belonging to a certain protocol, and determines how to react.

Control loop

At each tick, the control loop for each bus is the following:

1. Determine desire. Currently either `routine` (transport passengers along track) or `idle` (doing nothing when there are no passengers).
2. Update intentions. If desire is `routine`, this can be `drive`, `drop_off` or `pick_up`.
3. Perform actions.
4. Follow protocols. Check inbox for new messages and react.
5. Lead protocols. Initiate or continue according to protocol states and beliefs.

Protocols

Protocols consist of *states* and *messages*. In addition, buses keep track of *protocol-specific variables*, for instance timers. Messages can either be *broadcast* to all buses, or *sent* to a specific bus.

A message contains the *protocol* it belongs to, a *body* and optional *details*. Each protocol has a predefined set of message bodies, all with a specific semantic. The optional details are to carry variable information, such as an amount of something.

In the code, if `foo` is some protocol, its states look like `foo:SOME_STATE`, its message bodies like `foo.some_message`, and its protocol-specific variables like `foo/some_variable`.

The `overloaded` protocol

Currently the only protocol we have in place is `overloaded`. As mentioned in the intro, buses on the same track use it to make purchasing decisions.

Each bus keeps track of the following protocol-specific variables:

`overloaded/state` Stores the current state of the protocol this bus is currently in.

`overloaded/timeout` A constant; determines how long to wait since the protocol was last time finished. Currently set to 50 ticks.

`overloaded/timer` Keeps track of the wait.

`overloaded/dice` Stores the result of the dice-throw if two buses tried to buy at the same time.

Using those variables, these are the states and messages:

`overloaded:UNINITIATED` The state if the bus is currently not involved in this protocol.

If the bus gets overloaded, and `overloaded/timer` is zero, it broadcasts a `overloaded.want_to_buy` message, containing its track as a detail. It then goes to state `overloaded:ABOUT_TO_BUY`.

If the bus receives a `overloaded.want_to_buy`, and the detail of that message is its track, and it is not overloaded, responds with a `overloaded.do_not_buy` (and stays in this state).

`overloaded:ABOUT_TO_BUY` The state if the bus has just initiated the protocol.

If the bus receives a `overloaded.do_not_buy`, sets `overloaded/timer` to `overloaded/timeout` and goes to `overloaded:UNINITIATED`.

If the bus receives another `overloaded.want_to_buy`, and the detail of that message is its track, then there is a collision - two buses want to buy, but only one is needed. Now, the bus generates a random number, stores it in `overloaded/dice`, sends it as detail in an `overloaded.we_collided` message, and goes to `overloaded:COLLISION`.

If the bus receives nothing, apparently there is no other bus on its track that still has capacity, so it buys a new bus, sets `overloaded/timer` to `overloaded/timeout` and goes to `overloaded:UNINITIATED`.

`overloaded:COLLISION` The state if the bus knows another bus on the track also wants to buy.

If the bus receives a `overloaded.we_collided`, it checks whether the value in the detail is lower than `overloaded/dice`. If it is, it won the dice-throw, so it buys a new bus. If it's, not, it loses the dice-throw. Either way, it sets `overloaded/timer` to `overloaded/timeout` and goes to `overloaded:UNINITIATED`.