**Exercise 5: Call for transport**

The elevator hardware on the lab is controlled via a National Instruments PCI Digital I/O device, using the Comedi driver. An "elevator" abstraction has been created, that exposes a few functions in C that lets us use this I/O card. However, using this presents a few challenges for a project like this:

* Calling C code from other programming languages is sometimes a bit of a hassle
* The driver only works on Linux, which you might not use when working elsewhere than the lab
* Very few of you have an elevator

In order to alleviate the lack of elevators, a simulator was created. In order to use the simulator, you need to see what it does and input "button presses" to it, which means it has to run in a separate terminal window. Then in order to a) make interfacing with the simulator and the real elevator as similar as possible, and b) eliminate the need to call C code, both the simulator and the hardware elevator expose a network-based interface using TCP. In this way, you can seamlessly swap between the simulator and hardware elevators.

This means we have a simple client-server structure to the elevator:

* Two possible servers:
  + [The Elevator Server](https://github.com/TTK4145/elevator-server)
  + [The simulator](https://github.com/TTK4145/Simulator-v2)
* [Language-specific clients](https://github.com/TTK4145?q=driver)
  + Choose the one you need for the language you are using on the project
  + (If none exist for your language, ask for help and we'll add it once it works)

You may want to modify the client end of the driver, or possibly create your own from scratch. There is no particular requirement or recommendation involved here.

*(The*[*low-level C driver*](https://github.com/TTK4145-students-2019/exercise-5-ttk4145_group14/blob/master/driver)*for the elevator hardware is included in this repository for completeness, but using it is not recommended.)*

**Up and down**

Download the driver (for the programming language you are doing the project in), and test it on both the hardware elevator and the simulator.

* Using the hardware elevator
  + Download and run the [elevator server executable](https://github.com/TTK4145/elevator-server/releases/latest). It's likely already installed on the lab computers, try just running ElevatorServer from the terminal
  + If an ElevatorServer is already running, the new server will not be able to bind to the socket. If you need to kill it, you can do so by calling pkill ElevatorServer
* Using the simulator
  + Download and run the [simulator executable](https://github.com/TTK4145/Simulator-v2/releases/latest)
  + In order to run multiple simulators on the same computer, you will have to change the port on both the simulator (with --port) and in the driver (likely in a call to some init-function or in a config file)

**Up and away**

The elevator project can be roughly divided into two parts: Distributing the incoming requests (hall and cab calls) to the elevators, and then servicing those requests. At this stage you don't have the functionality implemented for the first part, but the latter part was a project in TTK4235. Since not all of you have taken this course, we'll have to get you up to speed on both the solution to this problem, and the preferred implementation pattern.

The relevant part of that project is documented in [Project-resources/elev\_algo](https://github.com/TTK4145/Project-resources/tree/master/elev_algo).

Implementing the "single elevator control" component as a state machine is the preferred pattern, to the point where we might even dare call it the *definitively correct* approach. The details and analysis of this pattern are covered in greater detail in the lectures, but here is the short version on how to follow it:

* Analysis:
  + Identify the inputs of the system, and consider them as discrete (in time) events
  + Identify the outputs of the system
  + Combine all inputs & outputs and store them (the "state") from one event to the next
    - This creates a combinatorial "explosion" of possible internal state
  + Eliminate combinations that are redundant, impossible (according to reality), and undesirable (according to the specification)
    - This should give you a "minimal representation" of the possible internal state
  + Give names to the new minimal combined-input-output states
    - These typically identify how the system "behaves" when responding to the next event
    - Leave any un-combined data alone
* Implementation:
  + Create a "handler" for each event (function, message receive-case, etc)
  + Switch/match on the behaviour-state (within each event-handler)
  + Call a (preferably pure) function that computes the next state and output actions
  + Perform the output actions and save the new state

You are encouraged to try to trace the analysis steps for the elev\_algo code linked above, but I also find that the vastly less rigorous approach of intuition quickly overtakes the methodical one. But for the implementation side you should take a much closer look, especially on why we consider events first then state (as opposed to state-first), and where the divide goes between "code that is directly in the event-handler" and "code that is in a function called by the event handler".

**Doing it yourself**

You should now implement some way to control a single elevator, as a part of the elevator project. This is where you get started "for real", so set up your environment (build tools, repository, editor, etc) the way you like it before you begin.

Since you don't have any way to distribute requests yet, you should use the button presses directly. This will have to change later, so keep code quality in mind.