**Ex 4: From Prototype to Production**

1. Don't overengineer.
2. Always design properly.
3. Minor detail change will ruin your perfect model.
4. Always prototype first.
5. You will only know what you didn't know on the second iteration.
6. There is only budget/time for the first version.
7. The prototype will become production.

**Network Module**

This exercise aims to produce the "v0.1" of the network module used in your project and at the same time prepare you for the network part on the design review.

You should start by taking a look back at [the beginning of Exercise 1](https://github.com/TTK4145/Exercise1/blob/master/Part1/README.md), and reevaluate them in the light of what you have learned about network programming (and - if applicable - concurrency control). At the same time you might want to look into what kind of libraries that already exist for your chosen language.

* [C network module](https://github.com/TTK4145/Network-c)
* [D network module](https://github.com/TTK4145/Network-D)
* [Go network module](https://github.com/TTK4145/Network-go)
* [Rust network module](https://github.com/edvardsp/network-rust)
* [Distributed Erlang](http://erlang.org/doc/reference_manual/distributed.html)

By the end of this exercise, you should be able to send some data structure (struct, record, etc) from one machine to another. How you achieve this (in terms of network topology, protocol, serialization) does not matter. The key here is *abstraction*.

Don't forget that this module should *simplify* the interface between machines: Creating and handling sockets in all routines that need to communicate with the outside world is possible, but is likely to be unwieldy and unmaintainable. We want to encapsulate all the necessary functionality in a single module, so we have a single decoupled component where we can say "This module sends our data over the network". This will almost always be preferable, but above all else: *Think about what best suits your particular design*.

**Design questions**

To get you started with designing a network module and/or the application that uses it, try to find answers to the questions below:

* Guarantees about elevators:
  + What should happen if one of the nodes loses network? All nodes share a life-signal counter to show they are still working. If they do not (after a security time), it is assumed this node cannot work anymore (so one lift out of service). In such a case, the other nodes should work without considering it any longer, until it replies back again (for UDP comm).

NOTE: With TCP, it is required to know both IPs to establish a connection, but with UDP only the server IP is necessary.

* + What should happen if one of the computers loses power for a brief moment? In this case applies the same situation as the previous point (loss of network).
  + What should happen if some unforeseen event causes the elevator to never reach its destination but communication remains intact? All lift movements are expected to be done in a specific time period. If one never arrives to destination, the affected lift is assumed not to work, so it is not considered any longer, until there is a re-validation.
* Guarantees about orders:
  + Do all your nodes need to "agree" on an order for it to be accepted? In that case, how is a faulty node handled? There is a peer-to-peer comm. by UDP protocol, so all nodes share some data that goes into a list. In this way all of them know the next orders and there are no conflicts.
  + How can you be sure that at least as many nodes as needs to agree on the order, in fact agrees on the order? Every node should send a message back after accepting/rejecting any new order.
  + Do you share the entire state of the current orders, or just the changes as they occur? For simplicity, only the changes that occur.
    - For either one: What should happen when an elevator re-joins after having been offline? It should go automatically to the start position, and then establish connection again. In this way we know its position after a network problem.

*Pencil and paper is encouraged! Drawing a diagram/graph of the message pathways between nodes (elevators) will aid in visualizing complexity. Drawing the order of messages through time will let you more easily see what happens when communication fails.*

* Topology:
  + What kind of network topology do you want to implement? Peer to peer? Master slave? Circle? Peer to peer, as there is no hierarchy and it is simple to implement.
  + In the case of a master-slave configuration: Do you have only one program, or two (a "master" executable and a "slave")? In any chosen option, there will be just one program code, considering all possibilities to assign in each specific case.
    - Is a slave becoming a master a part of the network module? Not applicable, as master-slave config has not been chosen.
* Technical implementation:
  + If you are using TCP: How do you know who connects to who? Both IPs must be known to establish a connection through TCP protocol. That means, if they are unknown or could change, the own IP should be broadcasted first through UDP protocol.
    - Do you need an initialization phase to set up all the connections? For using TCP yes, handled by the main thread and before to parallelize the code, yet not necessary for UDP.
  + Will you be using blocking sockets & many threads, or nonblocking sockets & select()? Blocking sockets block the thread until the read/write operation has been done (so one socket per thread would be necessary) and Non-blocking ones let the thread go on (so one single thread for everything could be done). Blocking sockets & select (controlled by parallel threads) are chosen because it is much simpler to work with.
  + Do you want to build the necessary reliability into the module, or handle that at a higher level? The reliability should be applied into the module itself.
  + How will you pack and unpack (serialize) data?
    - Do you use structs, classes, tuples, lists...? Classes, because they have OOP approach, offering the possibility to join different value types, functions, inheritance, polymorphism.
    - JSON, XML, or just plain strings? Plain strings for easiness.
    - Is serialization a part of the network module? No, it is not. Network module should only transmit/receive the data, previously prepared by another module.
  + Is detection (and handling) of things like lost messages or lost nodes a part of the network module? Yes, as those problems would lead to re-send messages or changing the hierarchy of the network.
* Protocols:
  + TCP gives you a data stream that is guaranteed to arrive in the same order as it was sent in (or not at all)
  + UDP might reorder the packets you send into the network
  + TCP will resend packets if they're dropped (at least until the socket times out)
  + UDP may drop packets as it pleases
  + TCP requires that you to set up a connection, so you will have to know who connects to who
  + UDP doesn't need a connection, and even allows broadcasting
  + (You're also allowed to use any other network library or language feature you may desire)

We work with UDP protocol because it is easier to work with, and we do not need the extra capabilities that TCP offers.

**Running from another computer**

In order to test a network module, you will have to run your code from multiple machines at once. The best way to do this is to log in remotely. Remember to be nice the people sitting at that computer (don't mess with their files, and so on), and try to avoid using the same ports as them.

* Logging in:
  + ssh username@#.#.#.# where #.#.#.# is the remote IP
* Copying files between machines:
  + scp source destination, with optional flag -r for recursive copy (folders)
  + Examples:
    - Copying files *to* remote: scp -r fileOrFolderAtThisMachine username@#.#.#.#:fileOrFolderAtOtherMachine
    - Copying files *from* remote: scp -r username@#.#.#.#:fileOrFolderAtOtherMachine fileOrFolderAtThisMachine

*If you are scripting something to automate any part of this process, remember to****not****include the login password in any files you upload to github (or anywhere else for that matter)*

**Extracurricular**

[The Night Watch](https://web.archive.org/web/20140214100538/http:/research.microsoft.com/en-us/people/mickens/thenightwatch.pdf) *"Systems people discover bugs by waking up and discovering that their first-born children are missing and "ETIMEDOUT" has been written in blood on the wall."*

[The case of the 500-mile email](http://www.ibiblio.org/harris/500milemail.html) *"We can't send mail farther than 500 miles from here," he repeated. "A little bit more, actually. Call it 520 miles. But no farther."*

[21 Nested Callbacks](http://blog.michellebu.com/2013/03/21-nested-callbacks/) *"I gathered from these exchanges that programmers have a perpetual competition to see who can claim the most things as 'simple.'"*