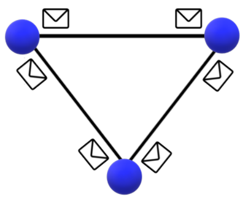
TTK4145, exercise 4: **Network design**

Hanne-Grete Alvheim, Maren Keini Haugen,
Iselin J. Nordstrøm-Hauge

February 2021



Contents

1 Guarantees about elevators

1.1 Loss of network connection

If one of the nodes loses its network connection it will continue to operate as normal, and keep sending messages. When an elevator sends and assigns an order to another elevator, it expects a fast reply confirming the order has been placed. If there is no reply, the elevator takes the order itself. If the elevator is disconnected, it will not receive any confirmation messages and always place the orders in its own queue. Also, when there is no network the elevator will not be updated on the shared information, meaning a list of structs containing information about each elevator. This results in the disconnected elevator losing updates on the information about the other elevators, but it will not be a problem. As the elevator connects to the network again it will be able to get the most recent updates on the information about the other elevators. The most recent update is what's needed as this is what is used for calculating the cost efficiency of placing orders.

1.2 Loss of power

If one of the nodes loses power for a brief moment, the elevator will not accept any more orders assigned to it from other elevators. This is done by not sending the placed-message. If the elevator gets new orders on its own hall-panel, it will send them to the other elevators while the power is out. Another elevator may have lost its network during this period. If that is the case, the elevator without power will after a few seconds of not receiving a placed-order send the order to the third elevator, regardless of the cost-function.

1.3 Unforeseen events

To prevent orders getting lost due to a unforeseen event, we want to implement a clock or timing function which keeps track of how long an order has been in the common information slice. Normally, when an order has been executed, we want an indication that it has happened. This is done by the elevator updating the common information slice with a new version of its struct, which do not include the already executed order. If an order has been in the common information slice for a given period without being executed (and thereby removed), the order will be reassigned to another elevator to ensure it is completed. This is done by the elevator which sent the order.

2 Guarantees about orders

2.1 No ”agreeing” on orders

Our approach to the design is to have one designated decision maker for each placed order. The decision-making role will be passed around from node to node as described in section 3. When the decision maker has made its decision, it will send the order to the node that will be executing it. This node will not have an ”opinion”, except when there is a loss of power (section 1), it will always place the order in its queue without ”hesitation”.

2.2 Losing packets containing order information

To handle losing packets containing order information between the nodes, we will implement the already discussed placed confirmation message. This message will be sent from the node that got the order, to the node that made the decision to give the node the order. If the decision making node does not get this message, the order message or the placed/accepted message is lost. If this happens, the decision-making node will place the order in its own queue. This might result in duplicate orders, but it is more critical that no orders are lost than two elevators going to the same floor. Losing updates on the common information slice is not critical either, worst case scenario is an efficiency cost.

2.3 Sharing the entire state

In our implementation (almost) all information will be common information. All nodes will have the ability to access the common information slice, so to be aware of critical sections will be important. By using goroutines we share memory through communication (channels) which makes handling this a bit easier. The shared information will include the elevators whereabouts, current sets of existing requests, and availability or failure modes of the elevators. These are the entire state of the current orders. Unassigned requests will only be available to, and handled by, the elevator that got the given requests. Why this is not necessary to share, and the rest is, is a result of our chosen topology. If an elevator re-joins after being offline, it has to update the common information slice. This by updating the information about its current orders etc. After this is done, the network of elevators should keep working as if the disconnect never happened.

3 Topology

3.1 P2P network topology, and the decision maker

We want to implement a peer-to-peer (P2P) network topology. All nodes will be peers, switching on the role as the decision-making node. The node that gets the order (by its hall button being pushed), will be the decision maker. The decision will be made by the node running a cost function on the shared information, and thereby designating the order to the most cost efficient elevator. It is always the elevator that gets the order that decides the order assignment.

3.2 Handling same button pushed on two hall panels at once

If the same order is placed at two separate elevator hall-panels at the same time, both nodes will become decision-making nodes. The nodes will execute the same cost function, and the data the function operates on will be the same since it is shared information used at the same time. The only part of the data needed for the cost calculation that is not shared is the unassigned request, and this request is the same for both elevators (same order placed on elevator hall-panels). Hence the nodes will come to the same conclusion, and duplicate orders will be assigned to one of our elevators. By including a deletion of duplicates in our code, any problems with this case will be avoided. Or will it? We have shared information being used for separate, but identical, calculations at the same time. What if there is a slight difference in the data being picked up by the two nodes? One might have missed an update on the common information slice. This may result in a race condition, and thereby different conclusions on which elevator that should get the order. Even though this happens, worst case scenario is that two elevators take this order, which is not a critical operation fault.

4 Technical implementation and module boundary

4.1 UDP and the Go net package

We are planning on using UDP to send packages via the network, as it provides a fast and efficient way of transporting messages. We do not find it necessary to have all of the other features of TCP, as we have found necessary solutions to our problems regarding the UDP and our elevator system, and we do not want excess overhead. The library we will use is the standard package net, already included in Go. It provides ways to communicate between computers, as well as TCP and UDP.

4.2 UDP broadcast

Broadcast will be used to send updates on the common information slice. In our UDP broadcast function, the broadcasting elevator's ID will be sent with the message, thereby differentiating messages from different nodes. Specific sockets on the different nodes will be used to assign orders and send placed confirmation-messages. These sockets are specified beforehand, and enable two-way communication between pairs of elevators.

4.3 Lost nodes, packages and duplicates

Since it focuses mostly on speed, there are a lot of things UDP does not handle. For instance, the messages may not be sent in the correct order. Messages can be duplicated and UDP does not detect package loss. Unordered messages are probably not a problem in the elevator system, because it does not matter much in which order an elevator receives first, considering the specifications. Lost messages might be a problem, but the planned solution to this is given in section section 2. If an entire node is lost, the UDP does not, as far as we know, have a way to detect this either. But as described in section section 1, the elevator will just continue working as normal. Duplicated messages might be a problem, as discussed in section section 3. As a result, the duplicates will be deleted.

4.4 Handling reliability

When a message is not received because it's lost, late or due to a network error, we want to handle it in a module. The node which sent the order has a timer to detect this happening, and can then start handling the situation.

4.5 Using structs and slices

All the elevators must share information in order for the cost function to work. We will use structs and slices to do this. We plan on using a slice with three structs, one for each elevator, to contain the information.

4.6 JSON and serialization

Another issue we must take into consideration, is how we want to structure the messages which are to be sent - the data must be serialized. If we were to do this using a string, it would use a lot of memory. Instead we are planning on using a package in Go called encoding/json. XML was also considered, however the output using JSON seemed tidier. Additionally, it could read slices , which will come in handy as we are planning on structuring the information in a slice. JSON will serialize the message to be sent in a string, and when it reaches its receiver it can deserialize it to get back the original object. Using pure strings, we would need a function to do this, but using JSON we can just use Marshal and Unmarshal to respectively serialize and deserialize the data. Since encoding/json uses serialization, and we plan to use this package (separate package from the go network package), serialization will be a part of our project.

A Networking overview, illustration

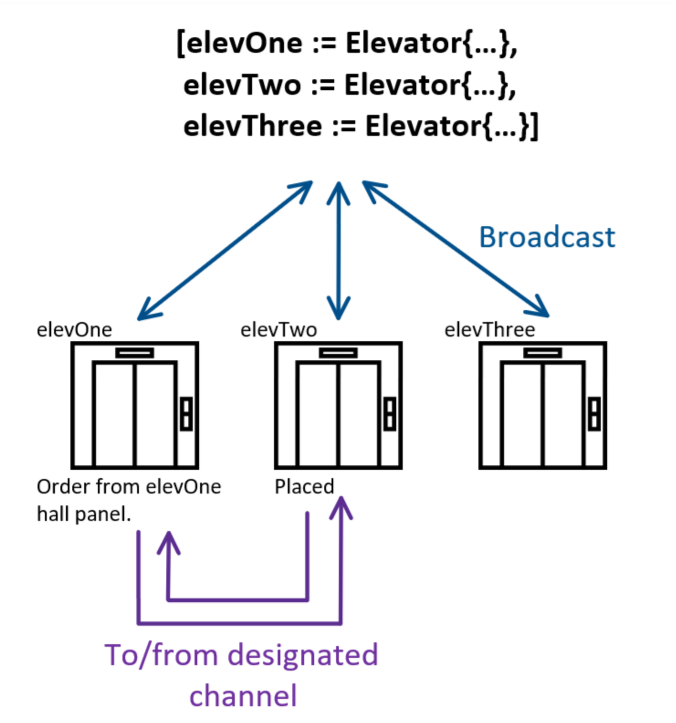

[elevOne := Elevator{...},
elevTwo := Elevator{...},
elevThree := Elevator{...}]
Broadcast
elevOne
elevTwo
elevThree
Order from elevOne hall panel.
Placed
To/from designated channel

Figure 1: Brief illustration of planned networking for the project. Message passing through UDP is illustrated with arrows for broadcast and for specific node-sockets. The latter is just illustrated for an instance where elevator one gets an order on its hall panel, and places and sends the order to elevator two. The shared information slice, with structs as elements, is also illustrated.