NTNU

Department of Engineering Cybernetics

TTK4145 - Real Time Programming

Elevator Project Report

Eirik Runde Barlaug
Kristian Hope
Jon Torgeir Grini

April, 2022

Table of Contents

1 Introduction

This project report from the course TTK4145, describes a real time program for controlling n elevators working in parallel across m floors. The report consists of two main parts, namely Design Documentation and Case Studies. In the first part, the over all system and important implementation details are covered. It is expected that the reader possesses some skills in `Golang` and has some knowledge of real time systems. In the second part, three case studies are presented. These are pivotal moments in the development process that proved to be of great importance for the outcome. The report is written concisely, so the reader's comprehension of the system should be improved if accompanied by the code.

2 Design Documentation

2.1 Implementation choices

This section highlights some parts of the implementation we consider especially relevant for understanding the workings of the system.

2.1.1 Network Topology

For this project we decided to implement a peer-to-peer network topology. This implies that all nodes on the network communicate with each other. As there is no designated master node, the master responsibilities are shared between the nodes. A peer-to-peer solution makes broadcasting with UDP a natural choice for communication. The topology is visualized in Figure 1.

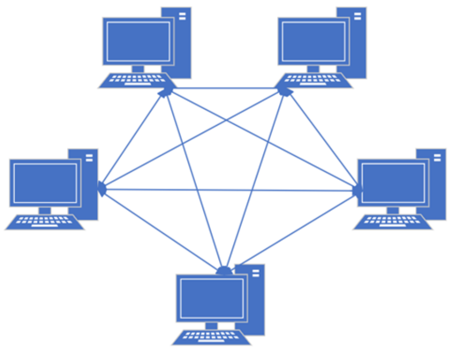

Figure 1: Peer-to-Peer Network Topology

2.1.2 Network Reliability and Robustness

As a measure to improve reliability the state of each elevator is broadcast periodically. This is an effective way of avoiding problems that can occur due to packet loss. If a package never arrives, a new package with the same or updated information will arrive soon after. There are three different messages being sent on the network:

I) A heartbeat signal to indicate that the sender is available for order assignment.

II) A struct containing the state of a peer, so that each peer has the information necessary to assign local button presses to the most eligible peer. Information about peers is also necessary so that an elevator can retrieve its cab calls after power loss.

III) Hall calls for distribution and to guarantee execution of orders.

2.1.3 Order Cycle

To guarantee service of orders it is essential to keep track of the execution of all hall calls. As all the elevators are working together to service hall calls, it is crucial that they are synchronized on the order status. Even if some elevators were to fail, all hall calls must be executed. Therefore, all peers maintain a hall calls table, which is a #floors x 2 matrix with all current hall calls in the system. Each entry of the matrix is of type `HallCall` on the form {Executing elevator, Order state, Acknowledgement list}. The four possible order states and possible transitions between them is visualized in Figure 2. The hall calls matrix is broadcast periodically to the other peers. When receiving from the network, the local and remote (incoming) hall call tables are compared and updated according to the scheme in table 1.

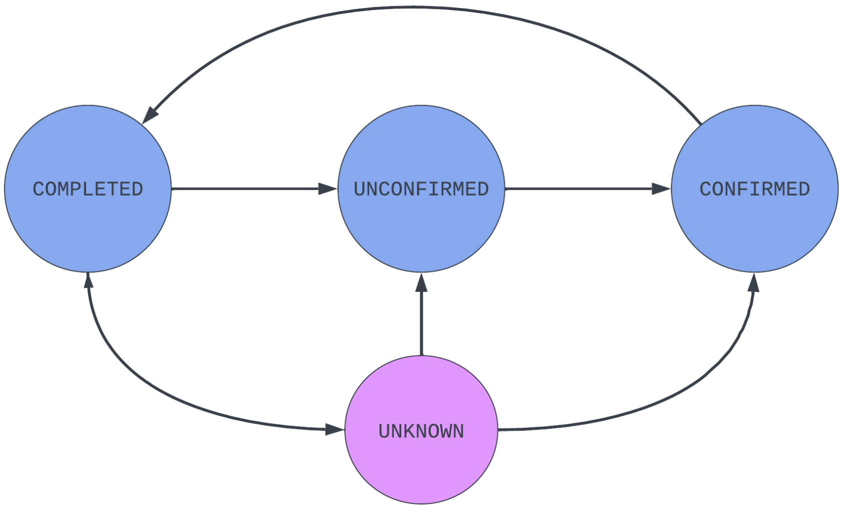

Figure 2: Order Cycle

Remote → / ↓ Local	**Completed**	**Unconfirmed**	**Confirmed**	**Unknown**
Completed	-	Unconfirmed +Ack	-	-
Unconfirmed	-	-	Confirmed	-
Confirmed	Completed	-	-	-
Unknown	Completed	Unconfirmed +Ack	Confirmed +Ack	-

Table 1: The acknowledgement- and order state update scheme in the `Distributor` module. "+Ack" represents adding the local ID to the acknowledgement list.

The state `Completed` represents a completed order. In this state there is no active order. When an elevator receives a local hall call, it determines the most eligible elevator, and transitions the order to `Unconfirmed`. Since the hall calls table is sent periodically, the other peers will soon read the order and update their hall calls table according to Table 1. Only when all elevators has acknowledged the order, the order transitions to `Confirmed`. Consequently, the corresponding hall light corresponding is turned on for all elevators. After order execution, the state is updated to `Completed` and lights are turned off. As the new order state is distributed on the next periodic send, the other peers updates their hall call tables and turns off their lights. Under "normal" elevator behaviour, the `Unknown` state is unused. However, it comes in handy when new peers connect to the network. The new peer can not know if there are active hall calls present for the other peers. Therefore, the new peer has its inactive hall calls in the state of `Unknown` to avoid disturbing the cycle of the other peers. As soon as the new peer reads any remote hall call tables from the network, it updates its own hall call table according to Table 1.

2.2 Modules

This section describes each module's responsibility and functionality. The module division and information flow is visualized in Figure 3. Each container in the diagram contains a list of input and output channels for the corresponding module. To easily see the flow of information, the number of a channel can be matched to a line entering or exiting the module. Network communication is handled entirely inside the scope of the modules, but is added to clarify which modules are communicating on the network.

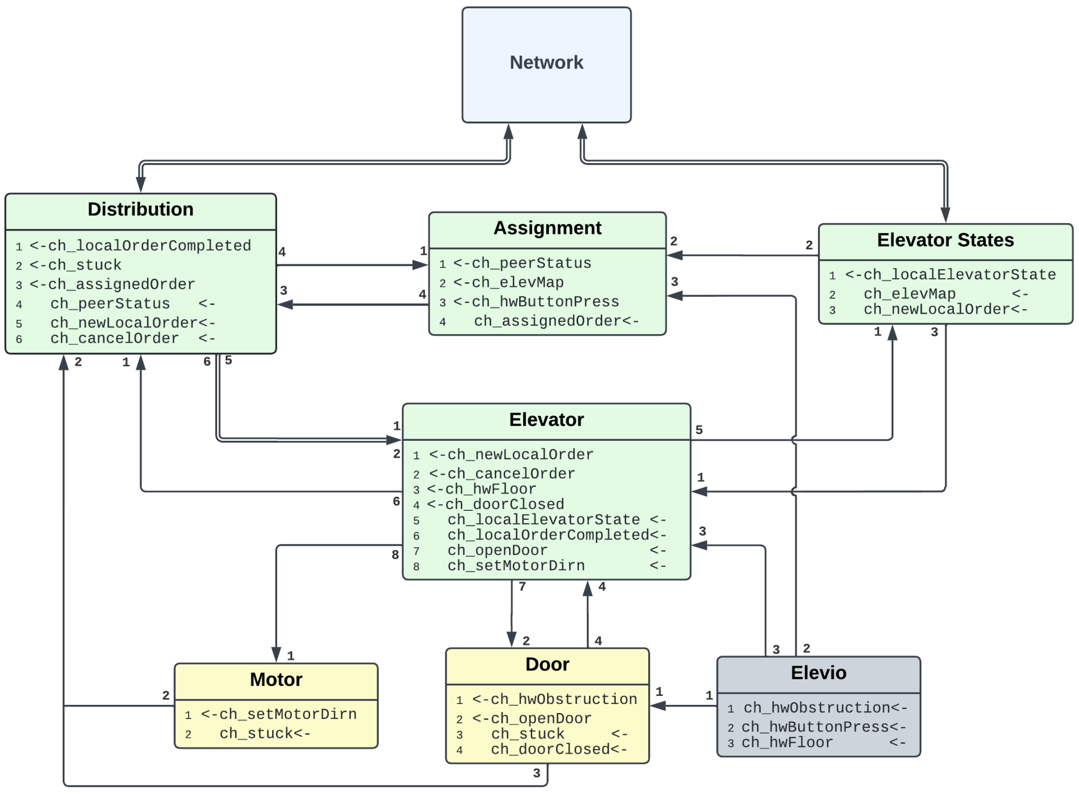

Figure 3: Module diagram showing the flow of information

2.2.1 Local Elevator

`Local elevator` is responsible for controlling the movements of the elevator and consists of the four following sub modules:

- **Elevator**: Contains a finite state machine (FSM) for the local elevator. The state of the elevator is determined by its current floor, direction, behaviour (door open, idle or moving) and a matrix of local requests. The FSM ensures correct elevator behaviour based on the current elevator state and the inputs. This sub module communicates with the outer modules by I) receiving orders and II) sending the current state and completed orders.
- **Door**: Contains a FSM for the door, and is responsible for setting the door light (i.e. opening and closing the door) and registering obstructions. If the elevator is obstructed longer than

some predefined duration, this module tells `Distribution` that the door is stuck. Door states are: `Open`, `Closed` and `Obstructed`.

- **Motor**: Sets the motor direction and contains a watchdog over potential motor failures. Tells `Distribution` in case motor failure is detected.
- **Elevio**: Contains functionality for polling and writing to the hardware of the elevator. That is: the buttons, obstruction switch, floor sensors, lights and motor direction.

2.2.2 Elevator States

This module contains a map of all the available elevators on the network on the form {ID: Elevator state}. It is responsible for I) updating the map when it receives a new state from some other elevator, II) broadcasting the local elevator's state to the others periodically and III) broadcasting the state of a peer that has fallen of the network and comes back on, which is useful for retrieving cab calls. This module takes the local elevator state and remote elevator states (from network) as input. As output, the elevator states map is sent to `Assignment` and retrieved cab calls sent to `Local Elevator`

2.2.3 Assignment

This module is responsible for assigning each incoming hardware button press to the most suited elevator, more specifically the elevator that has the lowest execution-cost for the given order. The cost is an estimate of the time passed before all orders of an elevator, including the order currently being assigned, is executed. The elevator with the lowest estimated time is assigned the order. In order to be able to calculate the most eligible elevator for an order, the module must have information about the states of all the elevators, and which elevators are available. These are inputs to the module. The output of the module is the assigned order, which is sent to `Distribution`.

2.2.4 Distribution

The area of responsibility for the `Distribution` module is distributing orders. Here the hall calls table, discussed in section 2.1.3, is stored and maintained. The module has three inputs coming from the local system and two inputs coming from the network, as shown in Figure 3. Consequently, the hall calls table are updated with both local and remote changes. `Distribution` also takes care of order acknowledgements, discussed in Figure 2. To be able to reassign orders in the event that a peer is lost, the module updates a list of available peers by receiving a heart beat from the network. When being sent an assigned order from assigner, `Distribution` updates the hall calls table, and immediately forwards the order to `Local elevator` if the local elevator was assigned. The whole sequence from button press to order completion is visualized in Figure 4. When signaled that the elevator is stuck, `Distribution` stops sending the heart beat on the network to inform the other elevators that this elevator is not able to service orders. Furthermore, in cases where the local elevator has had a temporary failure and is restored, `Distribution` informs the local elevator if any orders were serviced by another peer. Finally, the module also broadcasts its hall calls table periodically on the network, in addition to its own heartbeat signal.

2.2.5 Network

This module contains functionality for connecting to, receiving from and transmitting to the network. Especially `peers` and `bcast` are essential in the peer-to-peer communication utilized in this system. `peers` contains a struct, `PeerUpdate`, containing all peers currently on the network, as well as new and lost peers. `bcast` holds functionality for UDP broadcasts.

2.3 Order Sequence

In Figure 4 we have implemented a flowchart loosely describing the life course of an order under ”normal” circumstances. The chart visualizes how assignment interacts with distribution, and how transitions and acknowledgment of orders are done. The figure also displays how we set lights for hall calls, which is only done for `Confirmed` orders. Given that information about the hall calls are updated and transmitted periodically, events do not necessarily happen in a fixed sequence. We found it somewhat challenging to visualize the timing of events, but think this flowchart does a decent attempt without excessively complicating things. The diagram should be read from right to left. The color green signifies an input on a channel, and the color red signifies an output or the end of a case.

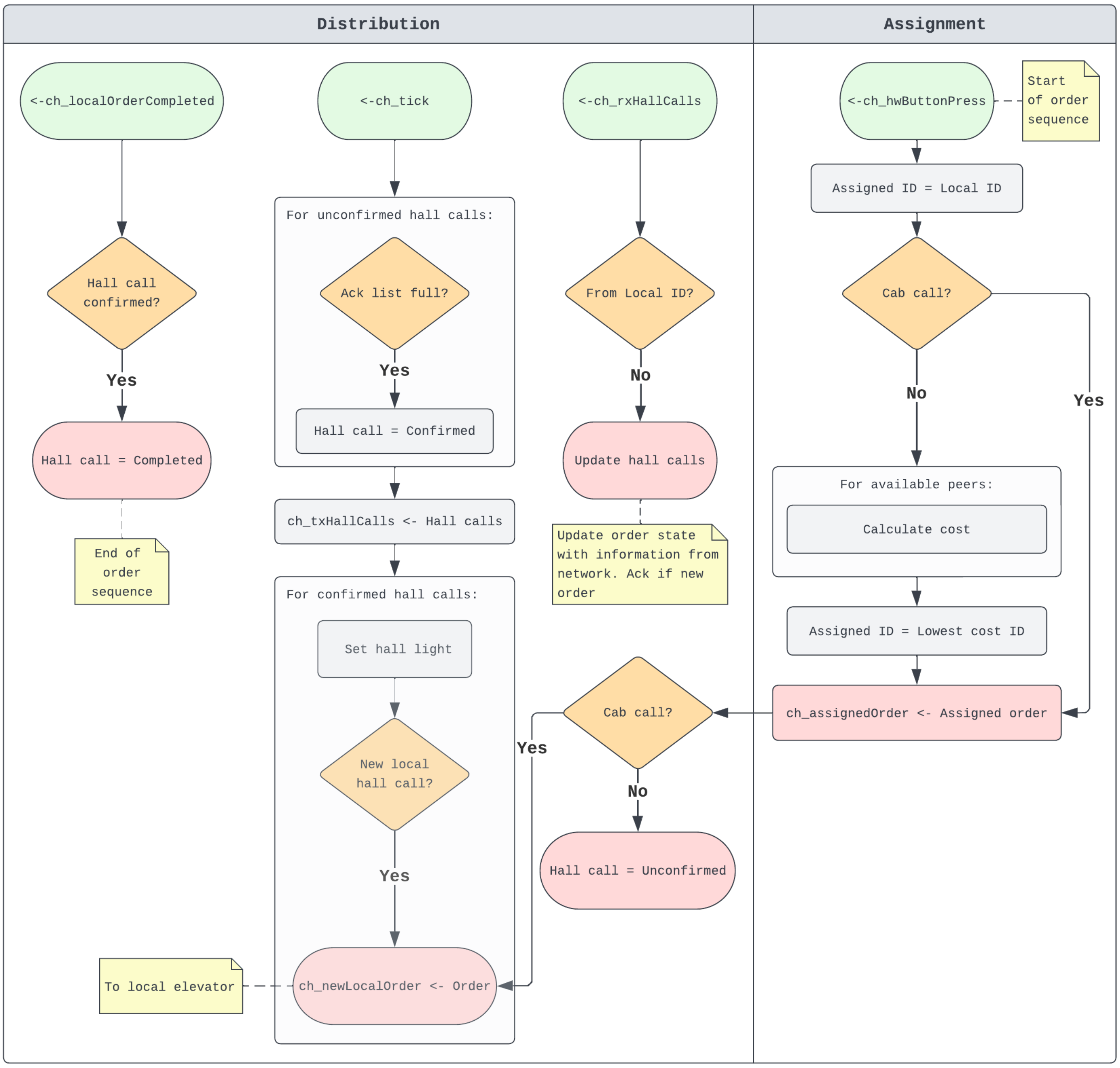

Figure 4: Order sequence

3 Case Studies

In this section, three cases of trade offs in the system, as well as some lessons learned in software design, are presented and reflected upon.

3.1 Network topology

Our first, and arguably most consequential design choice was the network topology. As stated in Section 2 we chose a peer-to-peer topology. Leading up to this decision we discussed the pros and cons of the possible topologies. The choice was however quickly narrowed down to either master-slave or peer-to-peer, as other solutions, such as ring topology seemed less applicable or complicated to implement.

The main reason we chose peer-to-peer ahead of master-slave was reliability. Although a master-slave topology has the advantage of simplicity and a clearer division of responsibility, it has a single point of failure (SPOF) in the master node in the absence of any redundancy solutions. SPOFs are undesirable in distributed systems such as our elevator system, where the goal is availability and robustness. Since the master node in a master-slave system acts as a communication hub, you run the risk of reading different information in the slaves, because the changes might not have fully propagated yet. Furthermore, a failure in the master node will cause all communication in the network to cease, leaving the slaves with outdated information. The consequences of such an event are mitigated with a master backup solution, but any information not transferred to the backup when the master fails is lost nonetheless. The question of who becomes the master when the failed master is restored is also an issue that needs to be addressed in a master-slave system.

A peer-to-peer solution rectifies certain issues of a master-slave system, by not having a designated master node, thereby no SPOF. Instead of having one node keep and convey information, all the nodes are equal and essentially possess the same information. As a consequence, the loss of one node will not cause any harm to the system as a whole, allowing for better reliability. Since all nodes are allowed to send updates, peer-to-peer communication does require more complex synchronization techniques as opposed to master-slave communication, where the master node authoritatively determines what information the rest receives.

Choosing a peer-to-peer topology made UDP a logical choice for communication protocol as every node needs to broadcast local changes on the network. In a master-slave solution TCP would be a viable option, but this introduces another layer of complexity. Choosing peer-to-peer resulted in a reliable system that handles synchronization and node failure quite elegantly.

3.2 Periodical broadcast

Another pivotal decision was that we decided to broadcast information periodically instead of on every state change. For the system to perform correctly, it is essential that all peers have equivalent and correct information about the other peers. If for instance one elevator is to calculate the execution costs for all other elevators when it receives a local order, it is crucial that the elevator states map contains the most recent states of all peers on the network. If not, some sub-optimal elevator may receive the order, and the system will not behave as efficiently as possible. An even worse scenario would be that an unavailable elevator is assigned the order, making the system violate the service guarantee.

The above scenarios may happen during packet loss, and since packet loss is considered probable in this project, it makes the periodic sending advantageous. On the other hand, broadcasting at state changes could have been the best solution if for instance limited band width was an issue, or if it for some reason were a goal to keep the network load low. It would also make it easier to trace the information and make the over all network message passing more uncluttered.

3.3 Redistribution of orders

When an elevator loses power, gets stuck due to hardware failure or simply disconnects from network we need to redistribute its active orders. For us, a pivotal decision was to retreat from the solution where each order was reassigned and redistributed to the single, most eligible elevator. As we have a peer-to-peer system, where the peer that receives the button press acts as the assigner for that order, it is not obvious whose responsibility it is to reassign orders after a peer disconnects. Although different solutions exist, they involve some cumbersome logic. One solution is that we let the peer with the lowest ID take the role as assigner, and redistribute the orders. Another solution is that all the active peers calculate the most eligible peer. Hopefully, they'll agree on the outcome as the calculations all should be based on the same information. However, in case they shouldn't agree, some kind of voting system must've been implemented.

The solution we ended up implementing chooses simplicity and reliability over efficiency. We decided that all the remaining peers assign the orders of the newly unavailable peer to itself. In this way, it is safe to guarantee that the orders are executed, and it provides a robust and easy-to-implement solution. The main disadvantage however, is that it is not very efficient. Having all the remaining elevators execute the same orders is time consuming. Hopefully, power loss, hardware failure and network disconnect does not occur too often outside our simulated environment.

3.4 Lessons learned

This section highlights some of the lessons we learned during the development of the elevator system. At an early stage in the development, the single-responsibility principle for modules was brushed aside in favor of fast progression. Our system seemed to be functioning well, but the modules were getting increasingly large and cluttered. It was getting difficult to pinpoint what part of the functionality each module encapsulated. Furthermore, small changes in a module often had far reaching repercussions, making it difficult to expand the program. For example, `Distribution` and `ElevatorStates` were at first one unified module that dealt with distribution of orders as well as updating the elevator states map. Later on we realized that these responsibilities were entirely unrelated, and that splitting the module would benefit the coherence of the program. When we began enforcing the single-responsibility principle in our modules the program became more predictable with less unexpected bugs from changes in the code.

Another lesson we learned was the importance of deep copying slices when sending them over channels. Our ignorance of this caused a number of inexplicable bugs early in the development process. To our great despair, some modules mysteriously knew about changes happening outside their scope. After struggling to understand the origin of this bug for quite some time, we realized that we were actually passing pointers between the modules. Passing a slice over a channel in `Golang`, creates a copy of the slice, but a slice is essentially just a pointer to the memory location of its first element. Copying a pointer does not copy the data being pointed to, but rather the memory address of that specific data. Passing slices around therefore enabled modules to read and modify memory they were only supposed to have received a copy of. This problem was fixed by deep copying slices before sending on channels. Our struggles encouraged us to minimize the passing of important data structures over channels, and further stressed the importance of module responsibility.