NTNU
Kunnskap for en bedre verden

TTK4145 - Real-time Programming

Elevator Project

Group 16:
Lisa Blaser
Tarek El-Agroudi
Finn Gross Maurer

April 2022

Table of Contents

1 Introduction

In this project, the goal was to create software for controlling multiple elevators working in parallel across multiple floors in accordance to the specifications listed in Appendix A. This report contains a design documentation, which presents how the project specification has resulted in a set of modules, and summarises how these modules function. Thereafter, three core pivotal decisions are discussed and justified. Finally, a few potential future improvements are outlined.

1.1 Project specification

The main points of the project specification from Appendix A are listed below.

S1 The button lights are a service guarantee

S2 No calls are lost

S3 The lights and buttons should function as expected

S4 The door should function as expected

S5 An individual elevator should behave sensibly and efficiently

S6 Calls should be served as efficiently as possible

These specifications should hold in the following failure states: (1) motor power loss, (2) network connection loss, (3) jammed doors, (4) software crash and (5) computer power loss. Packet-loss is explicitly not stated as a failure state and is to be expected. Three assumptions to make life easier are that (1) at least one elevators is not in any failure state, (2) cab call redundancy with a single elevator is not required and (3) no network partitioning occurs.

2 Design documentation

The elevator project is implemented in the programming language Go. Elevators are started by simply running the program with a unique ID, and there is no need to specify the number of elevators on startup. The distributed system has a peer-to-peer topology, which is discussed further in Section 3.1. Network messages are sent periodically over UDP.

2.1 Module overview

Unpacking the design specification has revealed a series of necessary and fundamentally different parts of the project, and this has motivated the selections of modules. Figure 1 shows the resulting module configuration as well as their interactions for a single elevator.

The specifications S1, S2 and S3 suggest that, once the light for an order is turned on, it must be served no matter which failure mode discussed in Section 1.1 occurs. Hence, lights cannot be set immediately when a button is pressed, since the elevator responsible could crash causing the order to be lost. A module is needed to process order information from all elevators and decide when an order is ready to be assigned. The responsible module in our implementation is called Order Redundancy, and is presented in Section 2.2.

The specifications S4 and S5 require a module that perform the actual execution of orders, after these have been assigned to the elevator. This is implemented as the modules Elevator Control and Elevator IO. Furthermore, in order for the elevators to actually have orders to execute, a module is required to assign confirmed orders to the different elevators. This is done by the Order Assigner module, presented in Section 2.3.

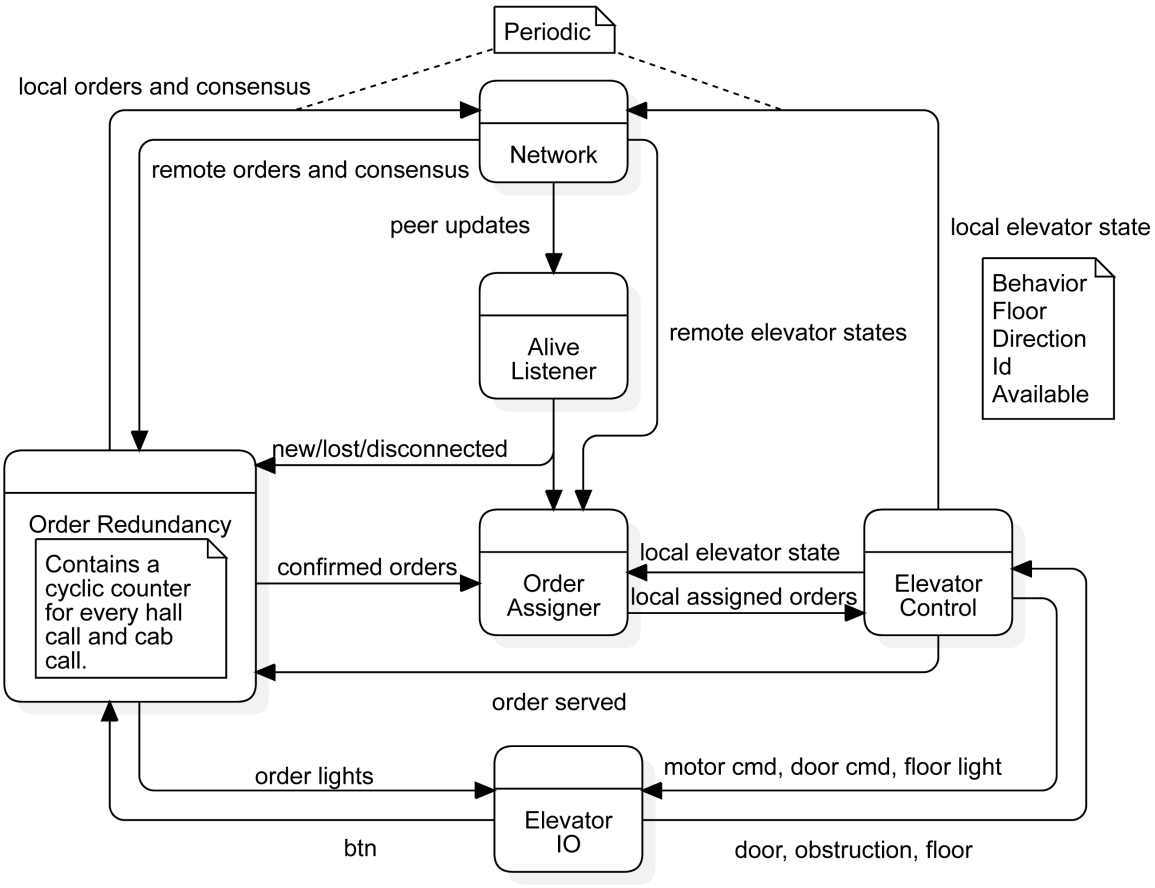

Figure 1: Module interaction diagram. Modules are run as go-routines and are connected through go channels or direct function calls to Elevator IO.

The need for communication with other elevators motivates the Network module, and the need to know what elevators are active for redundancy resulted in the Alive Listener module. These are presented in Section 2.5. The system is made robust towards packet loss by sending periodically over UDP. Hence, loss of packets is fine because a new message is likely to arrive shortly after.

Finally, the specification S6 is a general trait sought throughout the modules. Specifically, a good assigning algorithm is of course crucial, and this affects the choice of algorithm described in Section 2.3.

We now present the individual modules.

2.2 Order Redundancy

As was introduced above, the purpose of the Order Redundancy module is to ensure that any order whose light is activated is guaranteed to not be lost before it is served. Because of the assumption that all elevators except one may fail, there needs to be agreement or *consensus* between all elevators that an order exists before its light is set.

The module takes in orders and a "consensus list" from the network, as well as button presses from the elevator IO module. The consensus list is simply a list of all elevators that have registered the order as "Unconfirmed" (see below). It runs a modified cyclic counter state machine, documented in Figure 2, to update the state of each order. These states are described below, and the stars in the diagram denote the differences between hall calls and cab calls. As an output, the module sends its order and consensus information to the network, sets the lights for confirmed orders and

sends these to the Order Assigner.

To enable this formalism, the following order states were defined: *Unknown, Confirmed, None, Unconfirmed*. An order is in the "Unknown" state if the elevator has not received any information on the state of an order. This is the case whenever an elevator has just started, or when it has been disconnected from the network. Orders that are "Confirmed" have been registered by all elevators on the network, and are therefore "safe" (in accordance with the assumption that all but one elevator may fail) to be sent to the Order Assigner and have their lights set. Orders are in the "None" state when they have been served, and in the "Unconfirmed" state when they are registered, either due to buttons or a message from a different elevator.

The most important transition in the state machine is from "Unconfirmed" to "Confirmed". This transition must only occur when there is peer consensus, that is that all elevators have the order in the "Unconfirmed" state. This means that all elevators on the network agree on the existence of the order, which was the definition of the "Confirmed" state.

The distinction between "None" and "Unknown" is necessary because there is an important transition related difference between them. If an order is in "None", the elevator is on the network, and should not be placed in the "Confirmed" state directly if it receives this state from remote elevators. This is because "None" indicates that the order already has been served, and placing it back in "Confirmed" would initiate a possibly infinite loop. However, an order should be able to transition from "Unknown" to "Confirmed". If the order execution is in progress, this will allow the assigner to give the new or reconnected elevator the order. If the order is completed and the message comes from a network-delayed elevator, the state will simply transition to "None" once the network messages are exchanged a few times.

As for the difference between hall calls and cab calls, the main distinction is that for hall calls the elevator has to be connected to at least one other elevator to confirm an order, while for cab calls it is only necessary to be connected to the owner of the cab call. The same difference appears in the definition of "disconnected" in Figure 2.

2.3 Order Assigner

The Order Assigner takes as input the confirmed orders and elevator states from the Order Redundancy and Network modules, runs an assigning algorithm for all elevators, and outputs *locally* assigned orders to the Elevator Control module. The assigner algorithm [4] reassigns *all* orders every time a new elevator state or confirmed order list is received. The algorithm uses information on the time taken between floors and the time taken to open the door, and uses simulations to determine an assignment. It was provided as an executable. Cab calls are always assigned to the elevator that "owns" them.

This module runs on all elevators, and nothing is done with the orders assigned to other elevators. Inconsistencies between the elevators are expected, but not a problem due to the peer-to-peer topology and the concept of eventual consistency, which is discussed in Section 3.1.

2.4 Elevator Control

The Elevator Control module receives local assigned orders from the Order Assigner, and executes these orders. It is responsible for calling the required motor, door and floor light commands. It periodically sends it's state to the network and Order Assigner, and notifies the Order Redundancy module when an order is served. The basic logic was provided in the project [4] and hence this documentation will only concern the additional modifications.

The largest change from the provided implementation is that orders are no longer directly taken from call buttons, but come from the assigned orders list from the Order Assigner.

Furthermore, two boolean variables where added to the elevator state: "obstructed" and "motor_failure". When the state is sent to the other elevators, these flags are combined to a single

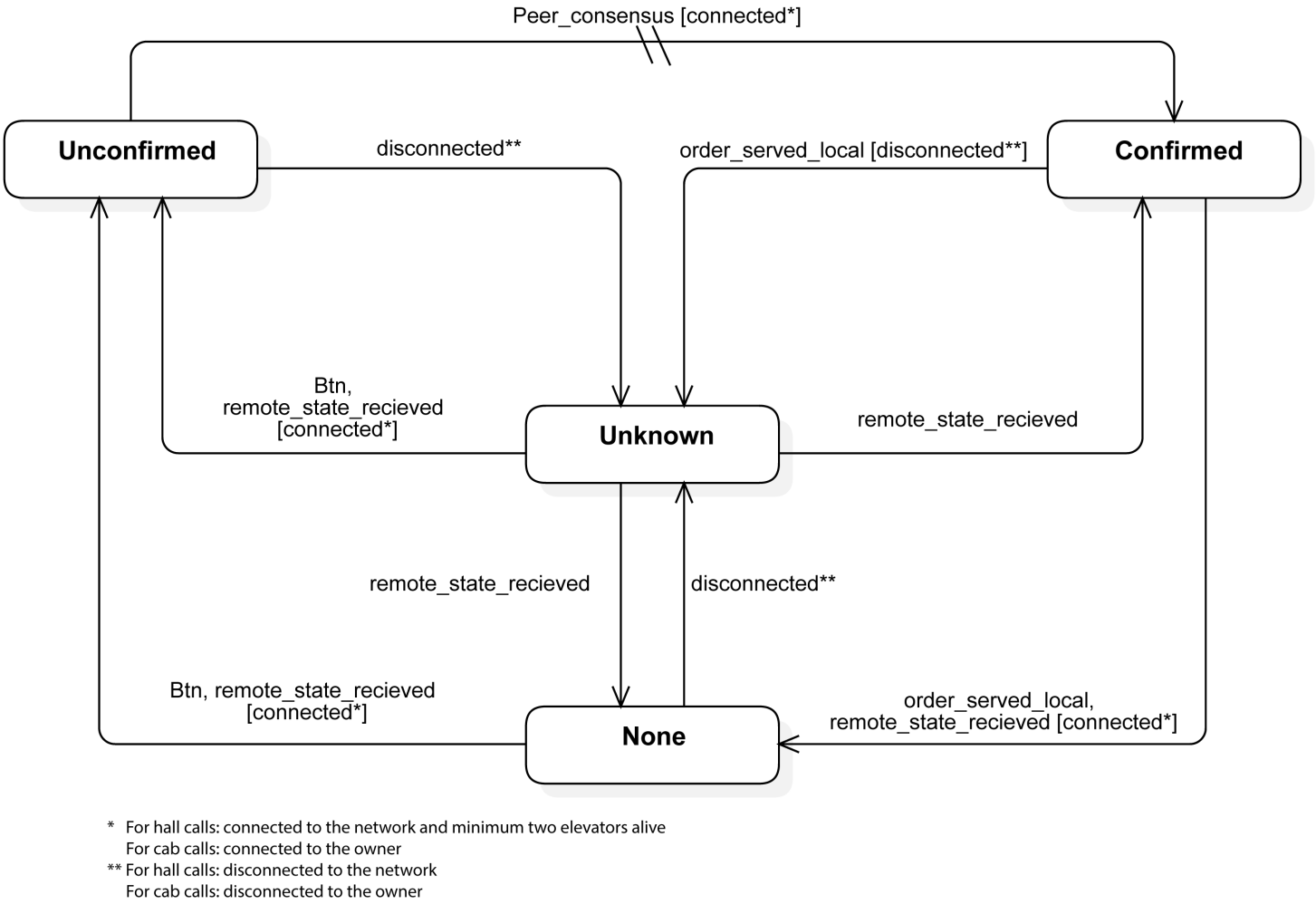

Figure 2: Cyclic counter state machine for order states.

"available" flag which, if set, prevents the assigner from assigning to that elevator. The fact that an obstructed elevator immediately is marked as unavailable is in part a solution to the "efficiently serve calls" part of S6. An obstructed elevator cannot take orders and, since we reassign all orders upon state or order changes anyway, orders should be assigned to other elevators in the meantime. The "motor_failure" flag is set by a timer on the "Moving" state of the elevator. If the elevator fails to reach a new floor in a given time, the "motor_failure" flag is set.

2.5 Network and Alive Listener

The provided Network module [3] enabled us to create UDP broadcasters and listeners. Hence the only amendments to the Network module were to add broadcasters and listeners for elevator states, orders and the peer update channels, as these are all the messages we send over the network.

In addition, the provided network code contained a "peer update" routine, which allowed us to create a module that keeps track of new and lost peers, which we called the Alive Listener. The Alive Listener listens to peer updates, decodes them and notifies the Order Assigner and Order Redundancy module if a new or lost peer is detected, and if the elevator is disconnected from the network.

2.6 Elevator IO

The hardware module was provided. Code and usage examples can be found in [2].

3 Case studies

This section analyses three pivotal decisions made in the project. Trade-offs and past mistakes are discussed, and alternatives are considered.

3.1 Choice of topology

A central choice in the project regarded the topology for the distributed system. At the time of selection, the considered options were mainly a master-slave topology and a peer-to-peer topology, of which the latter was selected.

Initially, the peer-to-peer topology was discarded. The reasoning was related to order assignment and can be summarised as follows: Each elevator assigns all confirmed orders to all elevators based on the elevator states. If, due to delays or packet loss, the elevators don't have the same state related data, they may come to different conclusions and end up assigning each other in a way that some orders are effectively not taken. This inconsistency, we thought initially, would have to be explicitly controlled by a complicated consistency module. In a master-slave topology, where only one elevator assigns orders and simply transmits this information, this is not necessary.

However, this seeming downside is solved by the subtle but simple idea of *eventual consistency*. Briefly, it is OK if inconsistencies result in certain orders to not be assigned. Once elevators have executed their orders and stand still, the periodic sending will eventually ensure that all elevators agree on state information. Since the order assigner reassigns *all* orders consistently on every state and order change, this ensures that the order that was not initially assigned is assigned. An interesting implication of this, however, is that it assumes the assignment algorithm to be deterministic, as the alternative no longer guarantees consistent outputs for the same inputs. This would prevent us from extending the algorithm to one with random elements, which is a possible disadvantage of a peer-to-peer implementation.

However, a series of significant advantages of the peer-to-peer topology for this project motivate the choice. For the purpose of full order redundancy, it is necessary for all modules to communicate

anyway, as is discussed in Section 2.2. So if any master slave solution would need to have the same distributed communication system as a peer-to-peer solution anyway, what benefit is gained from having a master? Peer-to-peer also has a conceptual advantage. Since all elevators on the network are to be considered as equals and there is no inherent priority in the system, this is nicely reflected by an implementation where all members of the distributed system er identical. It is also extremely easy to add elevators to the network, as these are immediately integrated without any extra protocols.

One alternative that was not thoroughly considered, but that bears considerable merit is a circle topology, where all elevators are connected unidirectionally in a circle. Confirming orders is now easy: if an elevator sends an order out, and it returns to itself, every other elevator has to be aware of that order. The sharing of order states however quickly becomes more complicated, and would probably require the addition of a second topology on top for the elevator state sharing. Hence the topology was not further considered.

In the end it boils down to the distribution of logic between the modules. In a peer-to-peer solution it is more complex to verify distributed orders (the Order Redundancy module), but far easier logic in the rest of the modules, which is partly why we opted for this solution.

3.2 Combined vs separate network message

Although not being a detrimental decision with regards to overall design, the choice of whether to transmit the state and order as a single network message or as separate messages over the network was one of the hard decisions, and is therefore included as a case study.

Initially, we wanted a solution where order and state are sent as a single message for the following reason: The order assigner assigns orders every time it receives different states or confirmed orders. Initially, if the messages are sent separately, one may encounter a situation where only either the state or the order has been received over the network. If, for instance, only orders have been received, the state variable is empty, but a difference in confirmed orders is detected and the assigner is called on an empty state, causing it to fail. If the state and order is always sent together, one can guarantee that the assigner is always called on a valid combination of data.

However, several disadvantages of this approach, and a simple remedy lead us to rather choose separate sending. Firstly, there is a code quality related issue with sending data together that is not immediately used together. While state is directly sent to the order assigner, orders are first sent to the order redundancy module before they eventually show up at the assigner as confirmed orders. Hence, they should not be sent as the same message. Furthermore, a module would be required to combine the order and state into a single message, as well as a receiver module that parses this message and redirects the state and order to different sections of the system.

Implementation-wise, the issue described above can easily be solved by simply preventing the order assigner from running if it does not have both state and confirmed orders available. Hence, the only possible disadvantage of separate sending is the increased number of network messages, which is not a problem if it enforces the separate role of the messages.

3.3 Choice of hardware redundancy

One of the requirements of the project specification was robustness and redundancy to hardware errors such as motor failure. Initially, this was implemented as a timer on orders. We considered that, if an assigned order does not cause the elevator to reach a new floor or to clear that order within a time frame, *something* must be wrong. This was of course only if the order was not reassigned, in which case the timer was reset.

However, after some thought, a much simpler solution was opted for. In the Elevator Control module, a simple timing of the “Moving” state sufficed to the same purpose. This initial mistake and correction has taught us to always consider the core purpose of an implementation and which

modules should be involved in accordance with this. If the purpose is to time hardware failures, the Order Redundancy and Order Assigner modules should not be involved, as these in essence are independent of the hardware!

4 Future improvements

A possible future improvement of general redundancy and fault tolerance could be to implement a set of process pairs [1] for each elevator of the system. This ensures that we have a mechanism to handle also unknown and unexpected errors. If the main program, or primary, fails to send an i-am-alive message to a backup, the primary is terminated and the backup takes over such that it starts from a consistent state. This would also require the inclusion of acceptance tests in the program, which is beneficial from a fault tolerance perspective anyway.

Another point of improvement relates to the readability of the Order Redundancy module. Currently, it takes the form of a for-select loop with seven cases on certain channels being received. The problem with so many cases is that it is harder to easily verify that the module is *complete*, i.e. does everything it should do. A remedy is based on the realisation that three of these channel cases - newElevDetected, elevsLost and disconnected - all hierarchically handle the same case, namely a change in the list of alive elevators. If the three cases were replaced by a single case receiving the alive elevator list whenever it changed, the Order Redundancy module would have one channel case per module it takes input from and it would be easier to, at a glance, verify completeness. This improvement also demonstrates that the entire Alive Listener module is unnecessary.

Implementation-wise, a point of improvement is to use slices instead of arrays. Array length has to be given as constants, such that global constants are needed for both number of floors and number of buttons. If slices were used, these configuration parameters could have been placed in a configuration file, like yaml or json, and then be passed to the respective modules from main, omitting the need for global constants.

Bibliography

[1] R. Anders Petersen. *4: Designing for crashability*. NTNU Lecture. 2022.

[2] R. Anders Petersen. *driver-go*. 2022. URL: https://github.com/TTK4145/driver-go.

[3] R. Anders Petersen. *Network-go*. 2022. URL: https://github.com/TTK4145/Network-go.

[4] R. Anders Petersen. *Project-resources*. 2022. URL: https://github.com/TTK4145/Project-resources.

Appendix

A Specifications

Downloaded from: https://github.com/TTK4145/Project.

S1 The button lights are a service guarantee

- S1.1 Once the light on a hall call button (buttons for calling an elevator to that floor; top 6 buttons on the control panel) is turned on, an elevator should arrive at that floor
- S1.2 Similarly for a cab call (for telling the elevator what floor you want to exit at; front 4 buttons on the control panel), but only the elevator at that specific workspace should take the order

S2 No calls are lost

- S2.1 Failure states are anything that prevents the elevator from communicating with other elevators or servicing calls
  - S2.1.1 This includes losing network connection entirely, software that crashes, doors that won't close, and losing power - both to the elevator motor and the machine that controls the elevator
  - S2.1.2 Network packet loss is not a failure, and can occur at any time
  - S2.1.3 An elevator entering the network is not a failure
- S2.2 No calls should be lost in the presence of failures
  - S2.2.1 For cab calls, handling loss of power or software crash implies that the calls are executed once service to that elevator is restored
  - S2.2.2 The time used to handle (compensate for) these failures should be reasonable, i.e. on the order of magnitude of seconds (not minutes)
- S2.3 If the elevator is disconnected from the network, it should still serve all the currently active calls (i.e. whatever lights are showing)
  - S2.3.1 It should also keep taking new cab calls, so that people can exit the elevator even if it is disconnected from the network
  - S2.3.2 The elevator software should not require reinitialization (manual restart) after intermittent network or motor power loss

S3 The lights and buttons should function as expected

- S3.1 The hall call buttons on all workspaces should let you summon an elevator
- S3.2 Under normal circumstances, the lights on the hall buttons should show the same thing on all workspaces
  - S3.2.1 Normal circumstances mean when there are no active failures and no packet loss
  - S3.2.2 Under circumstances with packet loss, at least one light must work as expected
- S3.3 The cab button lights should not be shared between workspaces
- S3.4 The cab and hall button lights should turn on as soon as is reasonable after the button has been pressed
  - S3.4.1 Not ever turning on the button lights because "no guarantee is offered" is not a valid solution
  - S3.4.2 You are allowed to expect the user to press the button again if it does not light up

S3.5 The cab and hall button lights should turn off when the corresponding call has been serviced

S4 The door should function as expected

S4.1 The "door open" lamp should be used as a substitute for an actual door

S4.1.1 The door should not be open (light switched on) while the elevator is moving

S4.1.2 The duration for keeping the door open when stopping at a floor should be 3 (three) seconds

S4.2 The obstruction switch should substitute the door obstruction sensor inside the elevator

S4.2.1 The door should not close while it is obstructed

S4.2.2 The obstruction can trigger (and un-trigger) at any time

S5 An individual elevator should behave sensibly and efficiently

S5.1 No stopping at every floor "just to be safe"

S5.2 Clearing a hall call button light is assumed to mean that the elevator that arrived at that floor announces "going up" or "going down" to the user (for up and down buttons respectively), and users are assumed to only enter an elevator moving in the direction they have requested

S5.2.1 This means that a single elevator arriving at a floor should not clear both up and down calls simultaneously

S5.2.2 If the elevator has no reason to travel in the direction it has announced (e.g. a both up and down are requested, but the people entering the elevator all want to go down), the elevator should "announce" that it is changing direction by first clearing the call in the opposite direction, then keeping the door open for another 3 seconds

S6 Calls should be served as efficiently as possible

S6.1 The calls should be distributed across the elevators in such a way that they are serviced as soon as possible