TTK4145 Elevator project

Group 55

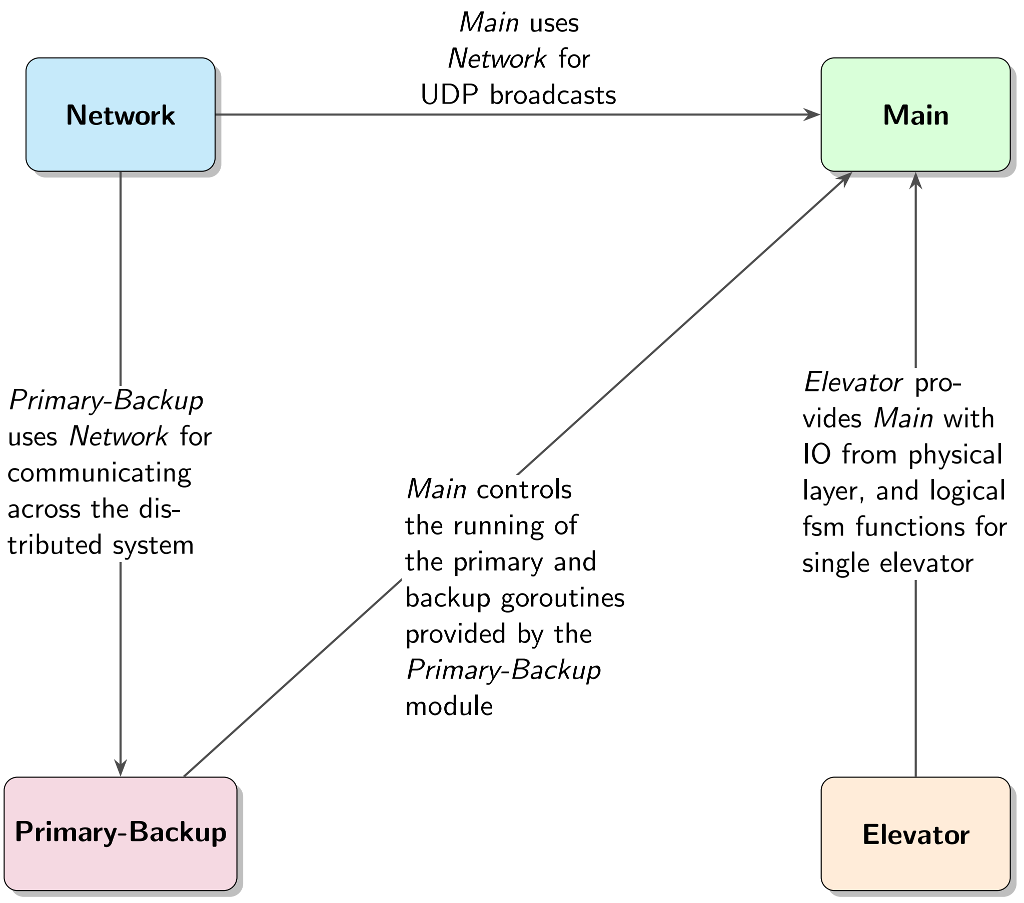
written by
Mats Heide : matsheid@stud.ntnu.no
Martin Hillestad: martihi@stud.ntnu.no
Jesper Lybeck : jespernl@stud.ntnu.no

May 1, 2025

1 Design Documentation

The system aims to provide fault tolerance based on redundancy. This is implemented through a primary-backup design. One node serves the role as primary, with responsibility for assigning orders to the other nodes. The other nodes on the network serves the role of backup, storing the orders in case a node is lost. This ensures that every time a buttonlight is lit, the order is saved by multiple nodes ensuring the order is completed, even if the responsible node fails. The approach for handling failures are built on this fact. In the case of lost engine power or persistent obstruction, the node throws an unhandled error resulting in the process terminating. When the node is lost, its hall calls are distributed to other nodes. Since every node is running a process pair, the crashed node is restarted, and its cab orders are restored from backup. The implications of this approach to handling faults will be discussed in more detail in section2.1.

Modules



Network
Main
Primary-Backup
Elevator
Main uses *Network* for UDP broadcasts
Primary-Backup uses *Network* for communicating across the distributed system
Main controls the running of the primary and backup goroutines provided by the *Primary-Backup* module
Elevator provides *Main* with IO from physical layer, and logical fsm functions for single elevator

Figure 1: The figure illustrates the dependencies between the core modules

	Responsibilities
Elevator	• Handling of IO operations • Pure functions for single elevator state machine
Network	• Provides networking utilities and functions for transmitting requests and orders with tolerance for network impairments
Primary	• Order assignment, redistribution of orders from lost nodes • Restoring cab calls for revived nodes • Merging of primaries • Broadcast state, including orders • Store elevator states from the other nodes on the network
Backup	• Monitoring primary health • Become primary if the current primary dies • Store state broadcast from primary
Main	• Manages running of goroutines • Binds together IO and logical functions for single elevator control • Broadcasts the local order queue for a single elevator • Interfaces a physical elevator as a node that can be connected to the network

Table 1: Module Overview

Service guarantee and light synchronization

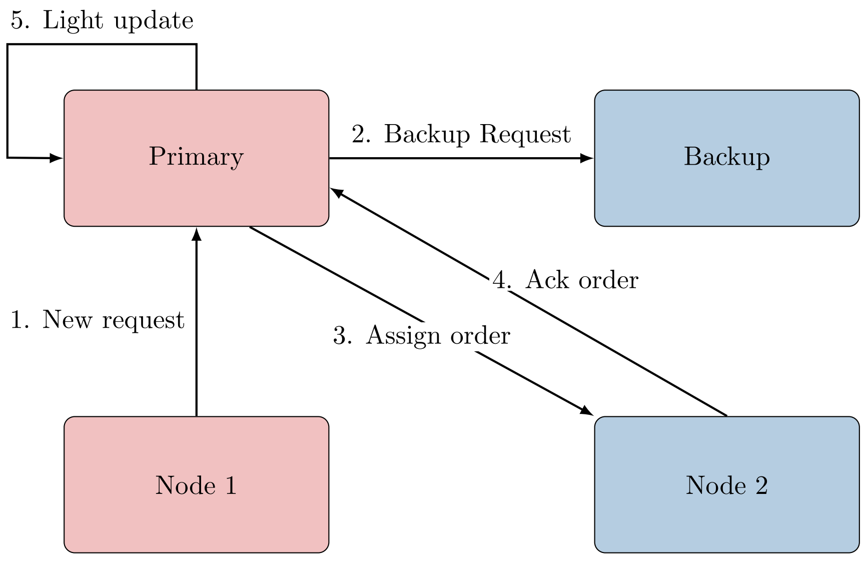


Figure 2: Communication flow between primary, backup and elevator nodes

After a request is sent to the primary, it is backed up and distributed. After acknowledgment is given, lights are updated. This is our approach to the button-lights being a service guarantee. The *Primary* and *Backup* blocks, in figure 2, represent individual processes, not complete network nodes.

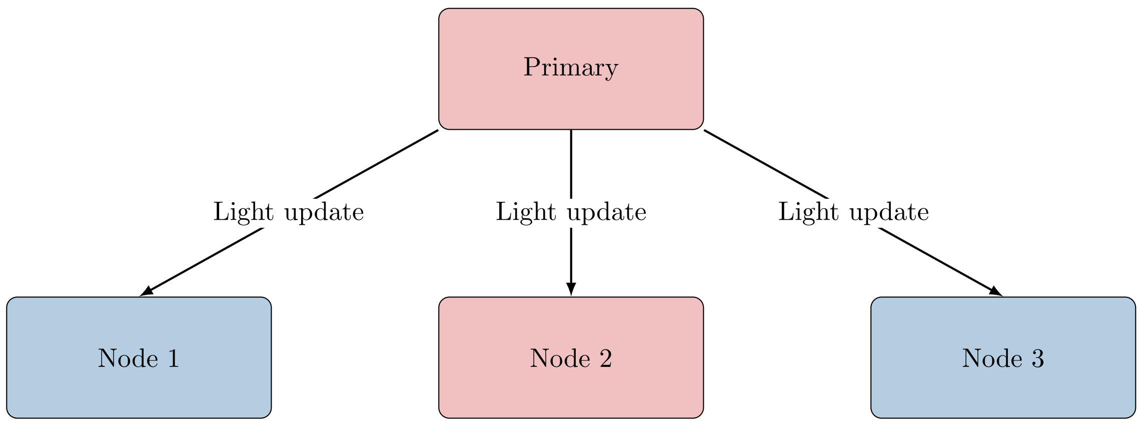


Figure 3: The primary broadcasts light updates to all nodes. Hall lights are shared, cab lights are individual

When a node receives a light update, it compares it with the current lights, and updates if something is new. In this way all hall lights are synchronized across nodes 3.

Merging of primaries

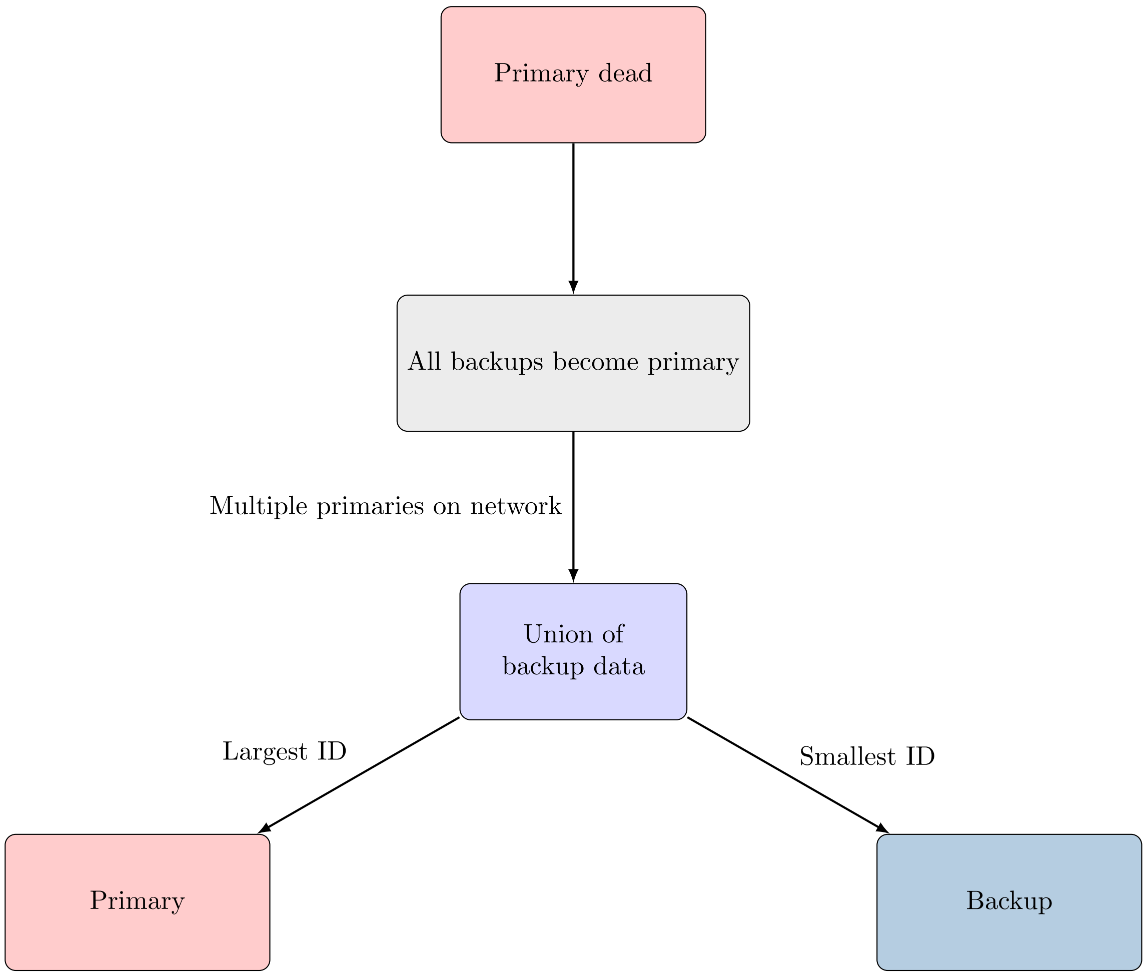


Figure 4: Routine of multiple primaries on network. This happens when primary dies or a new node enters the network

If multiple primaries are on the network, we need to merge them into a single primary. Since the primary nodes broadcasts their state, multiple primaries are detected if a primary node receives a broadcast from a different primary. This is solved by electing which node is to be primary, and demote all others. The orders from the different primaries are merged by union, ensuring that all orders are kept. This event will also occur when a isolated node is reconnected to the network. This is illustrated in 4.

Failure procedure

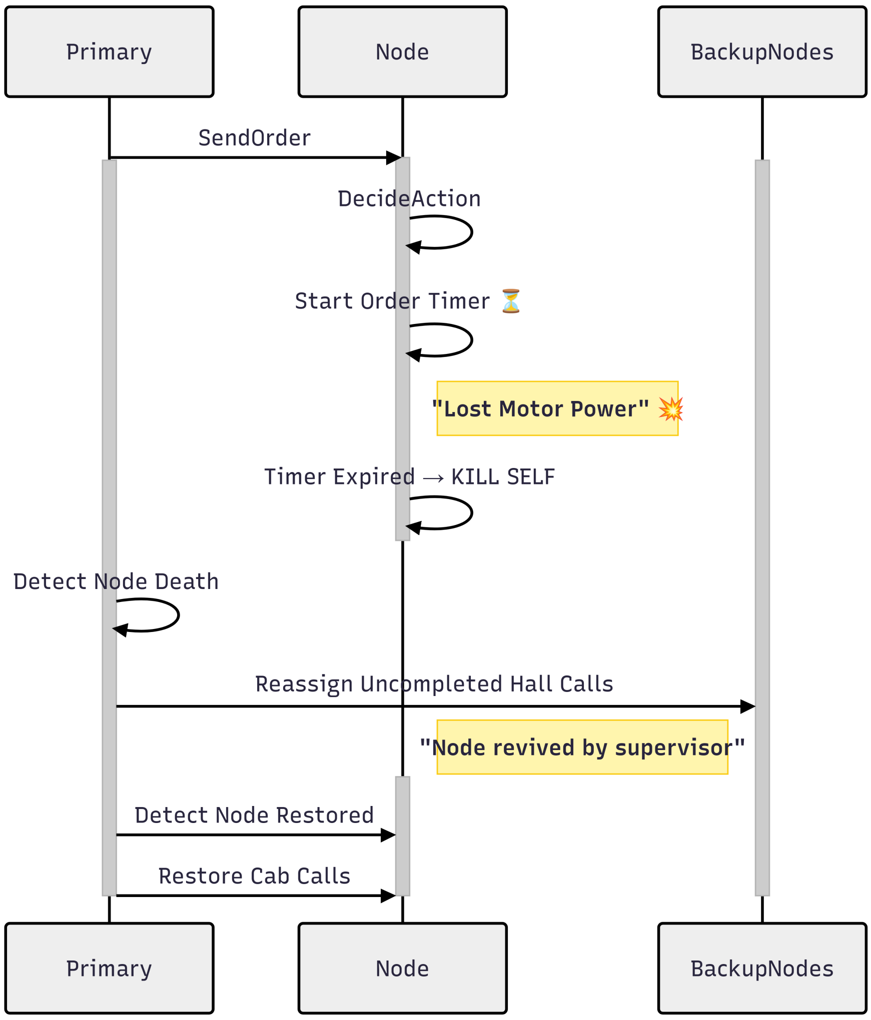


Figure 5: Sequence diagram illustrating a node failure with active calls.

Illustrated in 5, the node crashes and starts the reboot procedure automatically. Hall calls are distributed to spare nodes, and when reboot is complete and the node re-enters the network, cab calls are retrieved. This ensures that no calls are lost and orders are delegated if a node fails.

Communication procedure

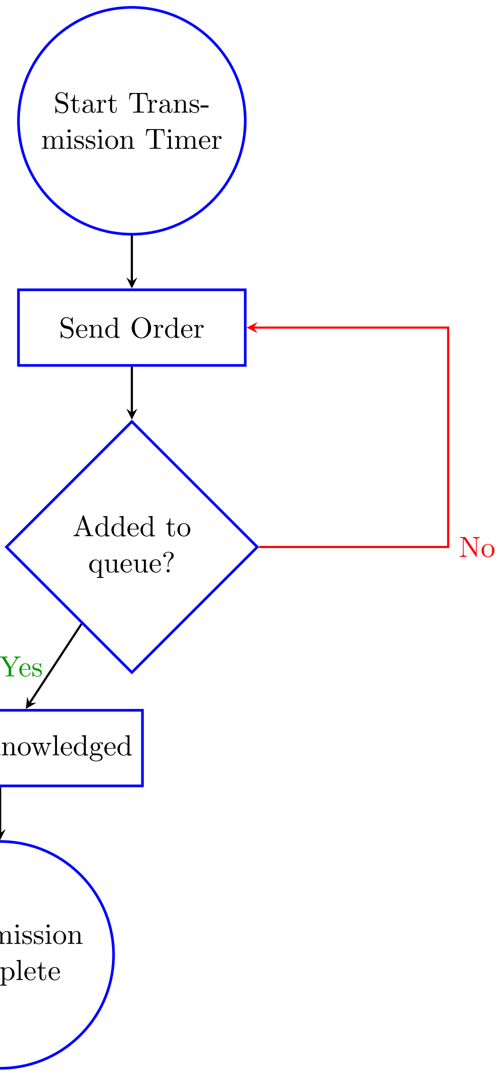


Figure 6: Flow chart of transmission during order distribution.

6 Illustrates the communication procedure. All nodes broadcasts their states, so this is used to acknowledge when a order is successfully transmitted. The transmitter keeps resending the order until it sees the order has been added to the receiver's status broadcast. In addition a timer is started by the transmitter. If no acknowledgement on the order has been given within the deadline, the transmitter reassigns the order to a different node. A similar approach using this implicit way of acknowleding messages is used for notifying the primary of completed orders.

2 Consequential Decisions

2.1 Error handling

From the system specifications, four different failure events are outlined:

- Network lost
- Obstruction
- Power loss
- Program crash

For each of the failures our interpretation of the project requirements was that hall calls should be redistributed to functioning nodes. Depending on the failure, additional considerations might also be necessary. For network loss we made it enter a *alone-on-network* mode, only servicing cab calls. For software crash, process-pairs is used to restart. Approaching the obstruction and power loss failures we came up with two different solutions:

Go offline after failure

The first approach was based on the idea that a elevator in a failure mode was unable to complete orders. As a result the program would self terminate, so the hall calls would be redistributed. In order to deal with reccurring failures, the node would upon restarting enter a *offline* mode. Until the node was able to close doors or move, it would not communicate that is was available. This offline mode made sure new orders would not be assigned. An overview of error handling is displayed in Figure 7.

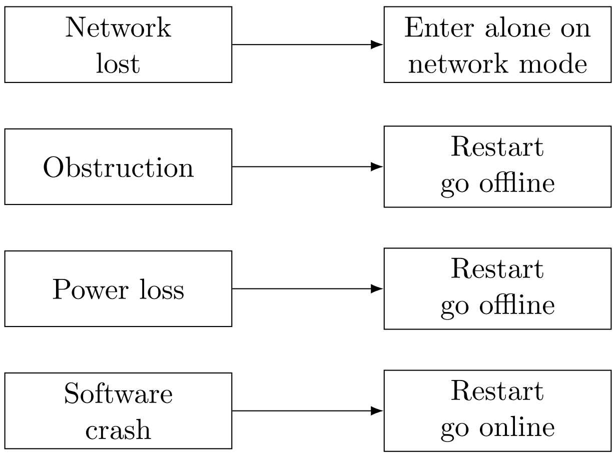


Figure 7: Overview of system behavior following different types of failures.

Remain online after failure

A different approach we considered was that motor-power loss, or obstruction failures was communicated to the primary node. The node would remain online, but signal the primary that it is in a failure mode. With this information the primary could avoid assigning orders to the nodes in failure modes. This is illustrated in 8

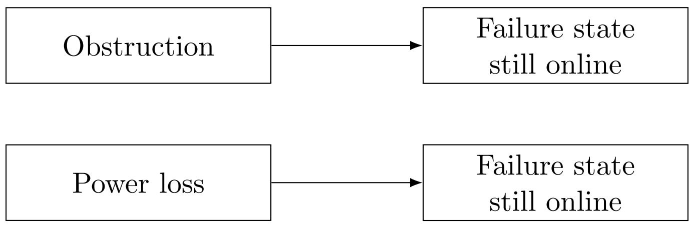


Figure 8: Possible solution to handle failure, the system remains online when possible.

When considering the solutions, we decided option 1 made for a simplier implementation. This approach meant we could rely on a single approach for all the failures outlined in the project requirements. If internet failed, it would not be able to broadcast its presence. If door was obstructed, or the motor was without power, the elevator would stay in offline mode until this was resolved. This unified approach was appealing.

After the FAT we realized that this decision did in fact not fulfill the project specification. An important detail was not taken into account. If the failure did not affect network or program flow, the node should still be able to receive hall calls. Going into offline mode in option 1, no transmission of hall calls to primary is possible. Option 2 would satisfy the specification and is in hindsight the clear choice between the two.

This mistake could have been avoided if we had discussed the project specifications with teaching assistants or the lecturer. It underlines the importance of a thorough understanding of the project and its demands, prior to the implementation phase.

2.2 Goroutine management

There are several ways to implement a primary-backup mechanism to ensure fault tolerance in a distributed system. We wanted to utilize the concurrency strengths of go in the form of goroutines to create this mechanism. The plan was to have one goroutine for the primary and one for the backup, each executing code involving their given task. We proposed two solutions to solve this problem.

Goroutines running in parallel

This solution initializes both the primary and backup goroutine when running main. Only one executes code, while the other *stalls* at the top. Initially, the backup is active and the primary is paused. When a takeover occurs, the roles switch: the backup pauses and the primary starts executing. This can alternate ensuring that a given node only executes one of the roles at a given point in time. A takeover is illustrated in Figure 9.

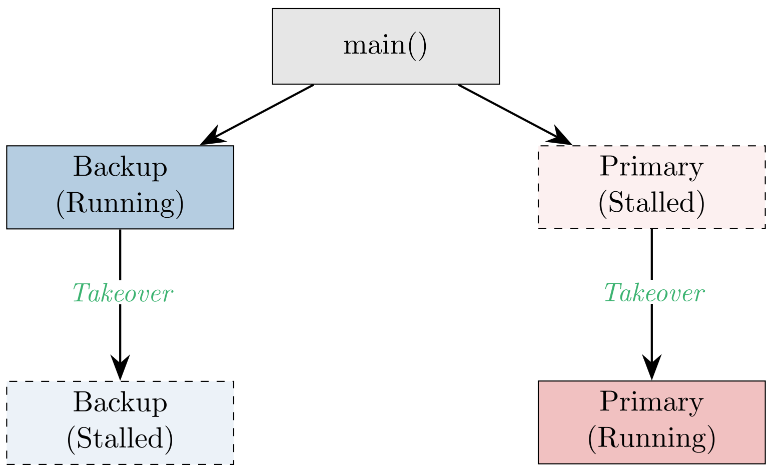


Figure 9: Solution 1, where both goroutines are running in parallel.

Implementing this would require a somewhat complex control structure. Including nested loops and conditional jumps, in order to switch between stalling different parts of the program, when a takeover occurs. This would solve the problem of only running either primary or backup routines at a given time. However the unusual program flow would mean that If something went wrong, it would be a challenging to debug. Understanding where something went wrong would be traceable, but not easy. A beneficial side to this approach is that it avoids running temporary goroutines. A single goroutine for the primary-backup logic, would follow the node through its full runtime.

Only one goroutine running at a time

The second solution we proposed was having only one of the backup or primary goroutines running at a time. To use the same example again, the backup is running and a takeover happens. This time a takeover message is passed to main and the backup goroutine terminates. The main receives the signal and starts the primary goroutine, and the takeover is completed. A figure illustrating a takeover is displayed in Figure 10.

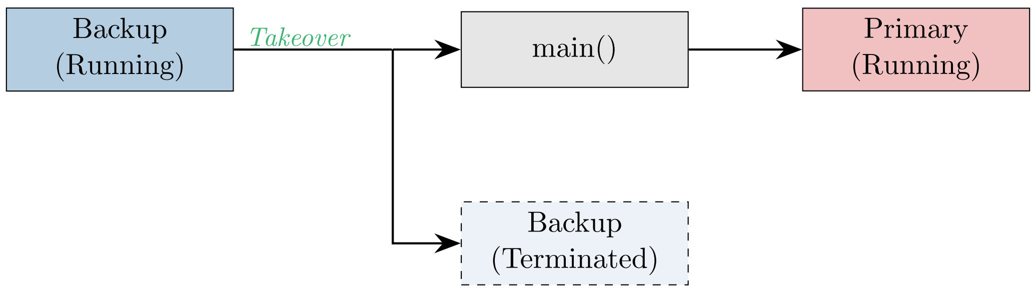


Figure 10: Solution 2, where only one go routine is running at a time.

The implementation of this solution makes both go routines consist of a single forloop; the inner one containing a for-select structure. If the only place in the code were a takeover signal is sent, is right before terminating the goroutine, traceability increases. As the takeover event should not happen all the time, The termination and restarting of goroutines would not happen to a problematic extent.

After carefully evaluating the solutions we ended up choosing the second option. This solution should be more traceable in the event of a bug, and it more clearly communicates the exclusivity of being either primary, or backup. This descision was critical at the time it was made. At this point the primary and backup goroutine had shared global variables. We later eliminated the use of global variables all-together, but the clarity of the signal-based management of goroutines still had its benefits in code readability. It also served the purpose of increasing our awareness with respect to how we used goroutines. We moved away from starting goroutines without consideration, to using them more cautiously.