Comparison of elevator scheduling algorithms for interfloor traffic

Kristoffer Rakstad Solberg - A0179890J
Mobile: 91406456
Group 24

April 18, 2018

Department of Electrical Engineering

Abstract

Scheduling algorithms are a governing part of real-time systems and there exists various algorithms for various needs and requirements of a real-time system. In this paper we will discuss the performance of two chosen on-line scheduling algorithms, *nearest car logic* and *fixed sectoring common sector logic* for scheduling passenger requesting pick-up service from elevators and dispatching in a multistory building. For the purpose of simulation an elevator *RTOS* was created using C++ and *robot operating system (ROS)* in Linux Ubuntu. Performance metrics of average customer travel-, response- and waiting time were monitored. Simulation results demonstrate that the proposed design facilitates acceptable customer waiting and service times, however the algorithms perform differently depending on the intensity and magnitude of passenger flow.

Note

The complete program code can be found on my Github page, with all the commits, history and a highly detailed README-file. The hyperlink is provided in the References at last page [2].

Nomenclature

Car - Individual lift	*RTOS* - Real-time Operating System
Call - Request for elevator	*ROS* - Robot Operating System
Terminal floor - Bottom floor	*NC* - Nearest Car
Master elevator - An elevator scheduler	*FSO* - Fixed Sectoring Common Sectoring
FS - Figure of Suitability	*Off-peak* - little traffic
Sector - Area of responsibility	*Down-peak* - all calls going down
Interfloor traffic - Calls from all floors	

1 Introduction

1.1 Motivation

The number of people waiting for an elevator usually varies throughout the day, with much of the traffic occurring in the early morning hours on a weekday. If people working in office buildings feel like they have to wait for a long time requesting service from elevator, this might be an indication that a new elevator system should be considered. Elevator traffic systems provide an ideal arena for the application of scheduling and queuing theory. Applying queuing theory to elevator traffic system can be very insightful providing the designer with a 'macroscopic' view of the operation of the system and the inter-relationship between different performance parameters such as the queue length, the waiting time in the queue and the service time.

1.2 Problem formulation

The problem at hand is the task of scheduling between two elevators a flow of passengers arriving at all floors in a 8-story building. From a contractor we have been asked to implement one of two on-line algorithms for customer pick-up service and efficient dispatching, but first we need to perform a simulation and analysis of their performance to make a right decision of which algortihm to implement.

1.3 Objective function

Supposed we know each passengers arrival time t_i, current floor r_i and destination floors f_i, both algorithms seek to minimize the average waiting time for the passengers until the elevator arrives. This is defined as (1.1).

$$\underset{w}{\text{minimize}}\; J = \frac{1}{N}\sum_{k=1}^{N} w_k \tag{1.1}$$

Where N is the total number of passengers, and w_k is each individual waiting time. Performance metrics such as average travel time and tverage tesponse time will also be recorded. Definitions of the metrics are shown below.

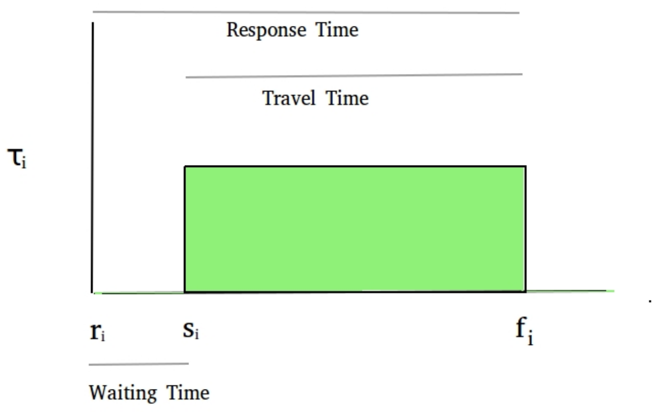


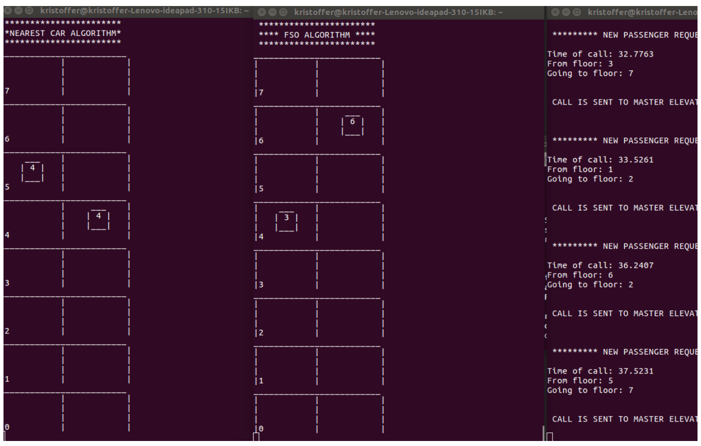
Figure 1: Illustration of the recorded metrics

1.4 Constraints

For the purpose of simulation the objective function is subject to a adjustments in terms of number of people asking for elevator service as well as the rate of their arrivals. Number of floors are constrained to 8 floors and number of elevators are 2.

2 Description of input data

All input data are generated from an homogeneous Poisson process where the arrival times are exponentially distributed based on the Poisson arrival rate λ. A particular customer also comes with a current floor and a desired destination floor, these two parameters are generated using *rand()*-functions in C++ 11. In the publishing node *poisson_call_generator.cpp* you can change parameters average passenger flow rate λ and the total number of passengers generated. Each simulation will be unique and all calls are independent of each other. In the most-right window in figure (2) you can see the generated calls being published according to the times in figure(3). The state of each elevator is then animated on the screen in real-time.


kristoffer@kristoffer-Lenovo-ideapad-310-15IKB: ~
kristoffer@kristoffer-Lenovo-ideapad-310-15IKB: ~
kristoffer@kristoffer-Leno

```
**********************
*NEAREST CAR ALGORITHM*
**********************
7
6
5   4
4        4
3
2
1
0
```

```
**********************
**** FSO ALGORITHM ****
**********************
7
6         6
5
4   3
3
2
1
0
```

```
********* NEW PASSENGER REQUE
Time of call: 32.7763
From floor: 3
Going to floor: 7

 CALL IS SENT TO MASTER ELEVAT


********* NEW PASSENGER REQUE
Time of call: 33.5261
From floor: 1
Going to floor: 2

 CALL IS SENT TO MASTER ELEVAT


********* NEW PASSENGER REQUE
Time of call: 36.2407
From floor: 6
Going to floor: 2

 CALL IS SENT TO MASTER ELEVAT


********* NEW PASSENGER REQUE
Time of call: 37.5231
From floor: 5
Going to floor: 7

 CALL IS SENT TO MASTER ELEVAT
```

Figure 2: Screengrap from simulation showing two the real-time simulation of elevators and flow of incoming calls

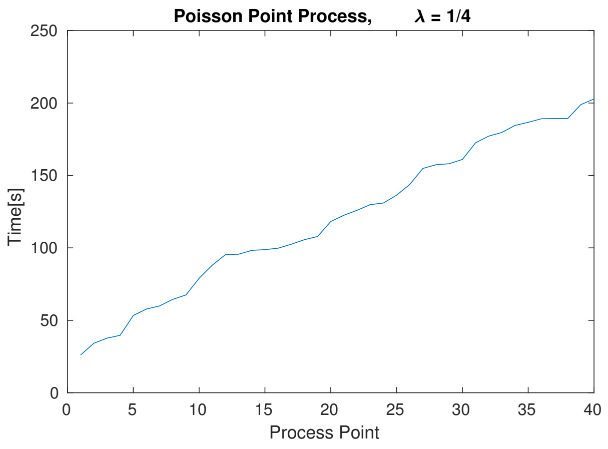


Figure 3: Homogeneous Poisson Process used for simulating passenger flow

3 Algorithm description

The first proposed algorithm, *Nearest Car Scheduling Algorithm* [1] will study the problem of efficient dispatching of people in off-peak traffic by scheduling elevators in terms of their suitability to handle a call. Calls are assigned to the elevator based on a function calculating the *Figure of Suitability* (FS) in its current state. As the elevator call is registered the elevator with the largest FS number is set to handle the call. The algorithm assumes that both elevators start in the terminal floor and operate at all floors.

The second proposed algorithm, *Fixed Sectoring Common Sector Scheduling algorithm* is devised for dealing with off-peak traffic [1]. It divides the building into equally sized static sectors, where every sector has its own elevator assigned to it. Every car will have the resposibility to serve customer calls in its assigned sector. If the sector above is considered vacant (the car responsible of that sector is answering to another call), the car is allowed to enter another sector to answer that call. In a special case of an elevator being the least occupied sector, it will be allowed to answer a call in a vacant sector below. The algorithm assumes that each elevator starts in it assigned sector (Elevator 1 - floor 0 - 3 and Elevator 2 - floor 4 - 7). An elevator car that is unassigned moves to the nearest vacant sector and parks there.

4 Simulation Results and Discussion

From figure (4) you can see the the flowchart of the simulation program. The simulation program uses a ROS framework in Linux Ubuntu for communication and C++ is the programming language of choice. Robot operating system (ROS) provides services designed for heterogeneous computer cluster such as message-passing between processes, and package management. Messages about passenger calls and running time are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. the protocol used by ROS is called TCPROS, which uses standard TCP/IP sockets[3]. Roscore is a collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate. It is launched using the roscore command.

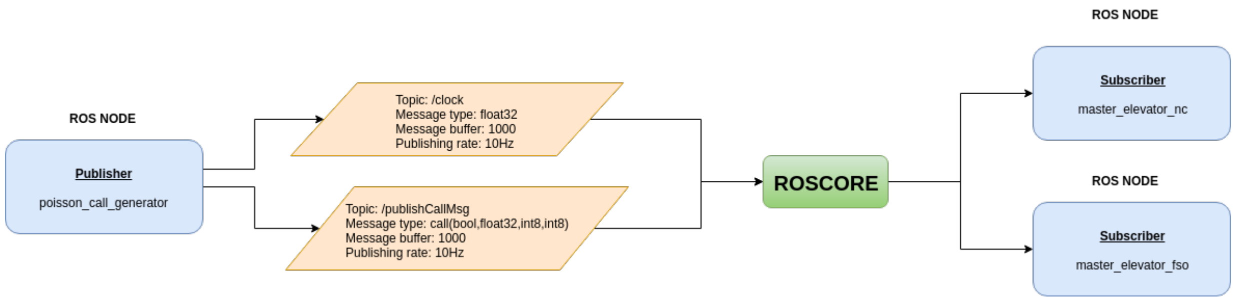


Figure 4: Flowchart of publisher and subscriber node architecture

4.1 Simulation architecture

The implemented algorithms are priority-driven, where as the on-line scheduler makes a decision without knowledge about the jobs that are released in the future. The *Master Elevator* -scheduler determine the priority of queued floors based on the current state (floor and travel direction) of the elevators. Scheduler decisions are made when events such as releases and completions of jobs occur. Therefore, priority-driven algorithms are event-driven. All the elevator jobs are *aperiodic* and are activated upon the arrival of an event.

The Event-driven simulation ”jumps” between occurring events, updating all objects to the current state. This is done in the *elevatorStateController()*-routine. Event-driven simulation method is more advanced in its implementation, but is suited for simulations with a long time period and big number of objects to be updated. The simulation ends when the time in the simulation reaches a certain value or the simulation reaches a certain state.The time period investigated in this project is short and the number of objects in the simulation is of manageable size. This means the objects can be iterated over without significantly affecting the runtime of the simulation program. Therefore the time-driven simulation will be used.

At Simulation start-up, the poisson_call_generator node will start publishing call messages at the *publishCallMsg*-topic with a rate of 10Hz. This topic has a meesage buffer with the capacity o housing a total of 1000 messages. As the subscriber *message_elevator_nc* or *message_elevator_fso* becomes available the messages will be picked from the buffer according to the first-in-first out (FIFO) principle. As the message is registered, a *Master Elevator*-routine will - based on the algorithm (NC or FSO) - make a decision which of the two elevators that is most suited to handle the job-call. In the next stage the job will be pushed in the back of the Ready-Queue (pick-up) for the respective elevator. A routine - *elevatorStateController()* - determines based on the current travelling direction and current floor what calls in the Queue that should be visited first. At the event of reaching one of the scheduled pick-up floors the job will perform a so called *Queue migration* where the job is redistributed and pushed from the Ready-Queue (pick-up) to the Ready-Queue (drop-off) that holds all jobs for passenger drop-offs. Figure (5) shows a illustration of the distributions of jobs.

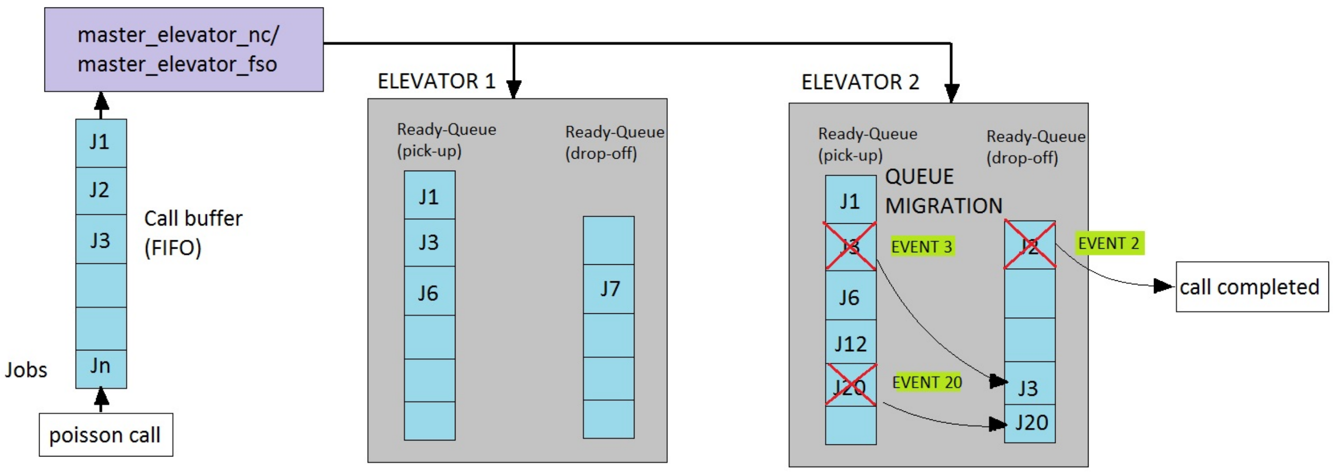


Figure 5: Queues

4.2 Results

The presented plots are constructed through data collection from 24 separate simulations carried out in 12-hour span. The simulations started with a passenger flow of 5 people with an high intensity arrival rate of $\lambda = 1$ for peak traffic, then gradually increasing the number of arriving passengers with an increment of 5 all the way up to 60 passengers. The simulations where then carried out once again, this time with an low intensity arrival rate of $\lambda = 1/8$ for off-peak traffic. The plots in figure (6),(7) and (8) show the total passenger flow for a simulation plotted against the Average Waiting Time, Travel Time and Response Time for the particular simulation. The purpose of this data collection is to analyze the robustness of the algorithms in terms of passenger flow.

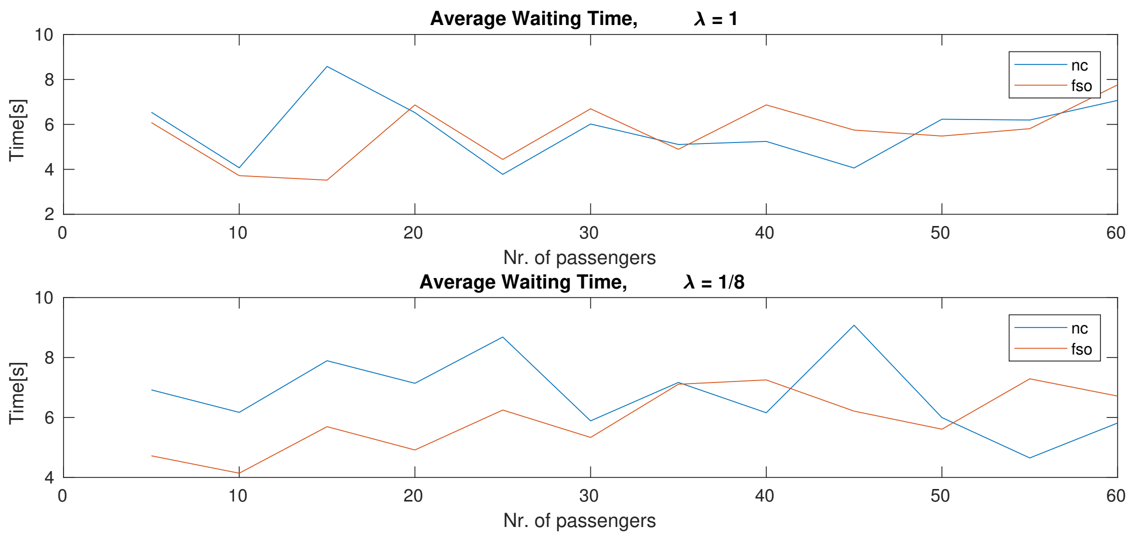


Figure 6: Simulated waiting time

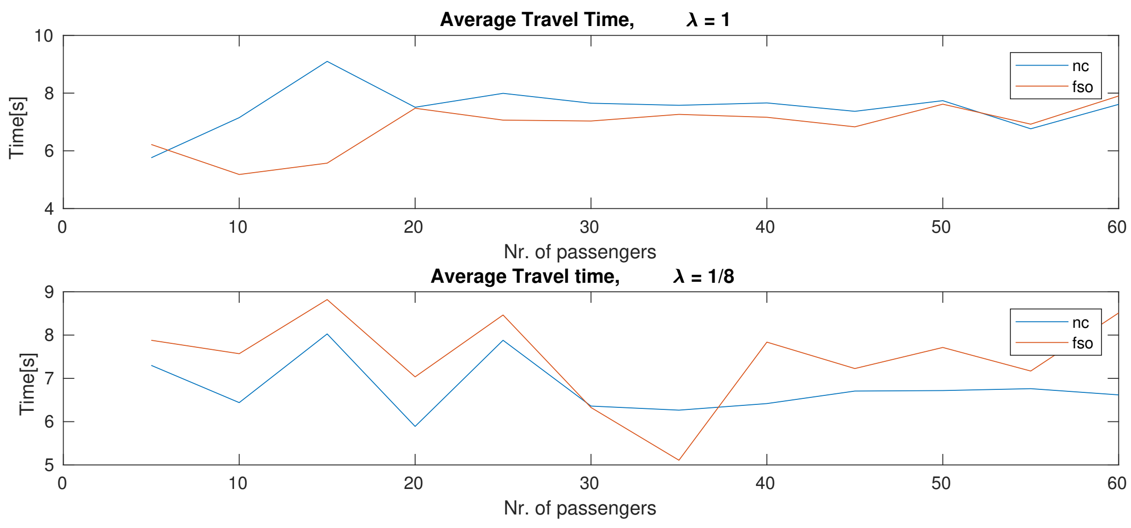


Figure 7: Simulated travel time

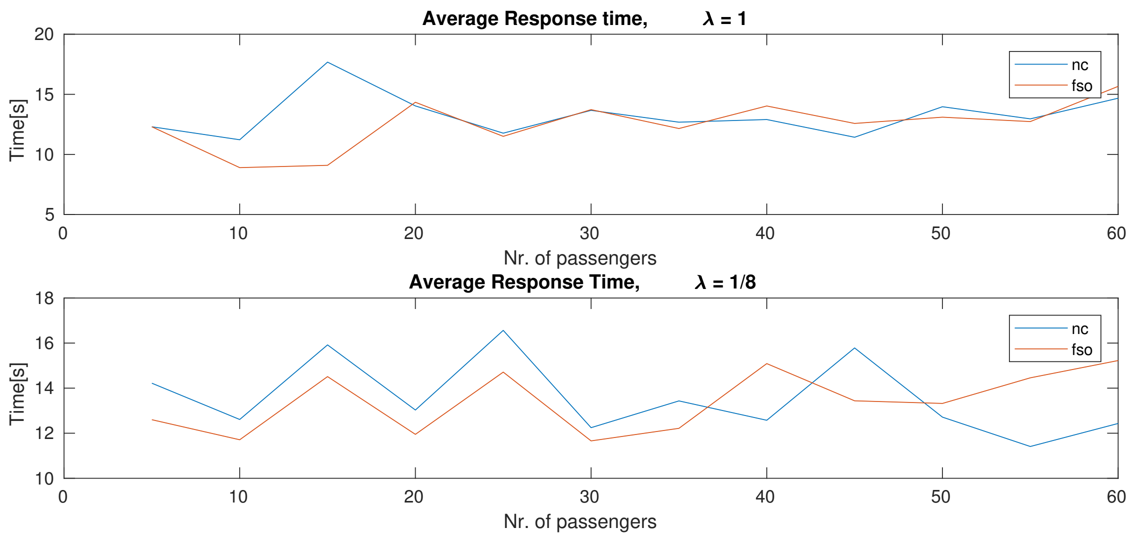


Figure 8: Simulated response time

4.3 Discussion

4.3.1 Average Waiting Time - Objective function

From figure (6) the plots show that the performance of the *Nearest Car* and *Fixed Sectoring Common Sector* algorithm are pretty similar for a peak-traffic at $\lambda = 1$ (in average one person arrive each second). Up until a workload of 20 passenger the NC-algorithm has almost twice the average waiting time than the FSO-algorithm. For a off-peak traffic at a $\lambda = 1/8$ (in average one person arrive each every 8 seconds), the NC-algorithm has almost twice the average waiting time for a passenger load lower than 30 people. For a passenger flow of 50 people up to 60 people the performance seems to turn and NC outperform FSO in terms of the objective function with about 3 seconds difference in average waiting time in favor of the NC-algorithm.

4.3.2 Average Travel Time - additional recorded metrics

At $\lambda = 1$ FSO outperforms NC up until a passenger flow of 20 people, then after that the average travel time is pretty stable at about $7[s]$ from pick-up floor until destination. This is the case for both algorithms. At $\lambda = 1/8$ the NC-algorithm outperforms the FSO-algorithm for all simulated passenger flows.

4.3.3 Average Response Time - additional recorded metrics

At $\lambda = 1$ FSO is better up until 20 people passenger flow, then after that they have pretty similar behaviour and stay more or less constant at an average reponse time of about $13[s]$. For at arrival rate of $\lambda = 1/8$

FSO seems to have a the best performs for small passenger flow and NC has the best performs for large passenger flow.

4.3.4 Strengths and Pitfalls

The first thing that is worth noticing is that the implementation of both algorithms for simulation were really time consuming and demanding because as an on-line algorithm they both required that the state of the elevators were continuously updated. To handle this an elevator controller was implemented so that the travelling speed, dwelling time at each floor, direction and current floor are all taken into account when the elevators travel from one floor to the other, this is also set in accordance with the global time reference published on the topic *clock*.

From the testing it was observed that both algorithms avoid the highest waiting time by having a fixed dwell time and therefore not taking into account the passenger clustering. However one of the backsides with both algorithms is that they are static systems meaning that once a new call is scheduled and added to the queue of one elevator, then that call cannot migrate to the another elevators queue if however the circumstances change.

Lastly, the basis on which the authors of the degree paper conduct their experimenting seems to be a bit lacking. In a similar matter they also create an actual elevator controller and have a fixed number of floors. However instead of having the passenger flow as a parameter, they rather choose to set number of elevators as a parameter and fix the passenger flow. Neither due they generate a Poisson Process for the incoming traffic. The number of elevators in an office building is usually very constrained [4], and if we could just simply increase the number of elevators in a building the demand for elevator scheduling and queuing wouldn't be necessary because everyone would be served with their own private elevator. It seems more right to fix the number of elevators and rather adjust the passenger flow to check the robustness of each algorithm.

5 Conclusion

The results for the comparison of the two algorithms NC and FSO show that they perform differently depending on the passenger flow for a interfloor traffic pattern. This is also an oberservation done by the authors of the degree paper. However, they tested their algorithms for different traffic patterns (up-peak, down-peak and interfloor traffic), but with a fixed passenger flow. In terms of the objective function average waiting time the FSO-algorithm seems to perform better than the NC-algorithm for both off-peak and peak traffic. But if also taking into account the average travel time and average response time the NC perform best when the passenger flow is large and FSO seems to perform best at a smaller passenger flow.

References

[1] Edlund. *Constructing a scheduling algorithm for multidirectional elevators*. http://www.diva-portal.se/smash/get/diva2:811554/FULLTEXT01.pdf. 2015.

[2] *Github source code*. https://github.com/Sollimann/simulation. Accessed: 2018-04-17.

[3] *ROS Techical overview*. http://wiki.ros.org/ROS/TCPROS. Accessed: 2018-04-15.

[4] *What Determines How Many Elevators a Building Will Have?* https://www.huffingtonpost.com/quora/what-determines-how-many_b_3421978.html. Accessed: 2018-04-17.