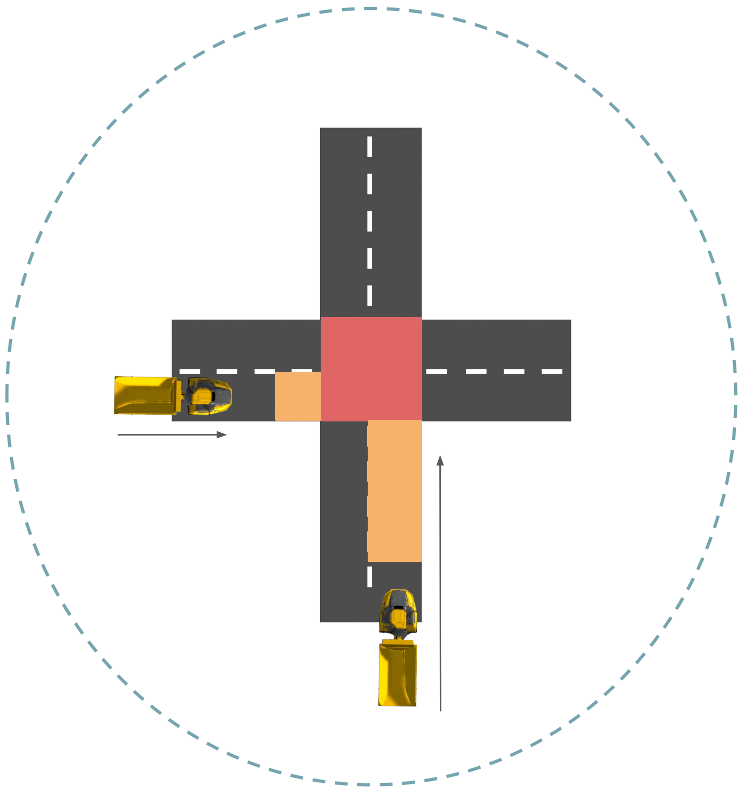
CHALMERS
UNIVERSITY OF TECHNOLOGY



Arbitrating Intersection Crossing

using V2I communications

Master's thesis in Computer Systems and Networks

MATHIAS ANDRÉASSON & TOMAS HASSELQUIST

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

Master's thesis 2018

Arbitrating Intersection Crossing

using V2I communications

Mathias Andréasson
Tomas Hasselquist

Department of Computer Science and Engineering
Chalmers University of Technology

Arbitrating Intersection Crossing
using V2I communications
Mathias Andréasson
Tomas Hasselquist

Supervisor: Elad Michael Schiller, Computer Science and Engineering
Advisor: Marco Admyre, CPAC Systems AB
Examiner: Vincenzo Gulisano, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration for crossing an intersection, depicting individual virtual brake zones and the critical area as the vehicles approach.

Typeset in LaTeX
Gothenburg, Sweden, 13th of June 2018

Arbitrating Intersection Crossing
using V2I communications
MATHIAS ANDRÉASSON & TOMAS HASSELQUIST
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

In the domain of automated driving, virtual traffic lights (VTLs) are key to making efficient intersection crossings. However, the design of an automated VTL remains an open challenge. This dissertation provides the design and implementation of a VTL prototype that can tolerate a multitude of failures, such as positioning system inaccuracy, packet omission and communication delays.
As such, our fault tolerant VTL can be a critical building block for future automated driving systems, completing existing autonomous driving efforts. We also present a prototype implementation and evaluate the protocol through extensive simulations. From our evaluation we establish interesting trade-offs in the system design. One such trade-off is observed when stopping vehicles further from the intersection, this reduces the negative impact of positioning system inaccuracies. Another example is that raising the frequency of message transmissions both improves recovery time of the system as well as acting as an effective remedy for severe packet omissions.

Keywords: Computer science, engineering, V2I, vehicles, virtual traffic light.

Acknowledgements

The biggest thank you goes to Elad Michael Schiller, our supervisor. Elad who worked hard to both guide and collaborate with us to make this a great project. Also a big thank you to Marco Admyre for all his help and kind words. Thank you to Anders Ek for arranging this opportunity to work with our thesis together with CPAC. Thank you to Johan Angenete for exploring further use cases and potential of this thesis. Thank you to Håkan, Caroline and Marcus for helping us improve our presentation skills. The final big thank you goes to all the other people at CPAC for all their help and the free coffee.

Mathias Andréasson & Tomas Hasselquist, Gothenburg, June 2018

Contents

1
Introduction

Maneuvering vehicles on regular roads introduces challenges such as crossing intersections. More specifically, the problem of agreeing who crosses an intersection first, but also efficiently as possible. A typical way to solve this problem is by using traffic lights in all directions, When the light is green, a vehicle can cross the intersection. The same task could be done by a virtual traffic light (VTL) which would enable the possibility of including automated driving into the intersection. This thesis considers different possible designs of a VTL and proposes a fault-tolerant version that would be able to deal with failures such as positioning inaccuracy, packet omission and communication delay.
We also consider environments that are not built up by regular roads, such as construction sites and quarries where it is not always clear who will cross the intersection first since it is not always feasible to put up traffic light systems. This infeasibility could have multiple causes, such as difficult terrain, or terrain that often changes, or being a remote area, among other problems. The only alternative is in some scenarios to slow down and wait until there is a visual confirmation that the vehicle can cross.
In recent years VTLs and cooperative driving have been introduced to solve the issue of arbitrating intersection crossing. Some examples of these VTLs or systems that are similar to VTLs can be seen in [1, 40, 26, 11].

1.1 Motivation

In some of the previous work, communication failures were not considered, in others a solution for a complete system was not given. Combining a complete solution with a system that can still be robust in the presence of failures would be the next step. In reality there are a number of failures that can affect the behavior of a VTL-like system. These all range from physical errors such as violations of traffic rules and accidents to errors in the software or environment such as communication failure or GPS-inaccuracy.
Thus leading us to our thesis which is a solution that can assist both human and autonomous drivers. This is an important step to help areas where there would potentially be both human drivers and completely autonomous vehicles. The problem lies in arbitrating an intersection where the vehicles have to come to some conclusion on what vehicle would cross in what order. This can be achieved by implementing a queue system, and suggesting a vehicle, or a group of vehicles that can cross first. This will be the base of our solution, with the addition of being a robust solution

to handle failures.

1.2 Problem Description

The main challenge from the application perspective, is to supervise the entry of vehicles into intersections. An intersection is not necessarily two roads that meet but can also be a broad range of different configurations of interconnecting roads. An example of this is a two-way narrow section where only one vehicles from one direction should enter at any given time.
It is important to keep in mind that the intersection arbitration should still be achieved even in the presence of failures. In Section 3.3, the considered failures are listed. While more failures exist they are outside the scope of this project.
There could be many issues such as deadlock where vehicles that have gained access are stuck behind vehicles that do not have access, or even violations of the requirements where other vehicles than the allowed have access to the restricted area.
The algorithm thus needs to handle a specifically defined area in which only a number of vehicles would be allowed into at a certain point in time while allowing every vehicle access to this area at some point, thus arbitrating the intersection.

1.3 Use Cases

More so than the traditional intersection, this algorithm in theory supports multiple variants of intersections. This includes intersections that have both more and less amounts of roads leading in and out of it. An example of this would be a narrow road segment where only vehicles may pass in one direction at a time. Another example would be a 5-way intersection. The reason for the flexibility and supporting of many different scenarios is the membership service that the implementation entails. It is simply a queue tag system which grants certain clients access based on certain criteria, while having sophisticated failure detection and recovery attributes.
Other application scenarios would be places where roads and traffic rules are not clearly defined, such as in a quarry or a construction site. In these cases having a traffic light that indicates who has the right of way can be very helpful. This especially goes for areas where the visibility is low.
This would not be limited to cars, since other vehicles interact in the same way. This means that any such vehicle can make use of the same algorithm as long as the areas in which vehicles would have access are defined correctly. This includes boats, trains, planes and various other ground vehicles.

1.4 Evaluation Criteria

In this project the evaluation criteria are chosen to represent potential problems the algorithm is built to handle as well as specifically problematic errors such as messages omission or message delays.

These criteria were chosen to be relevant while comparing related work, and also to make a system that was as usable as possible in a real scenario. The evaluation criteria focused on making a system that could tolerate a number of failures, such as packet omission, delay and positioning system errors. The main goal was that the system would still be functioning if these failures did occur, but still have a focus on researching how the intersection throughput would be affected if they did. An added functionality later emerged, this functionality followed naturally from the initial goal of being used as a real system. This system is the failure detection and failure recovery mode. All of these criteria were measured and tested through an implementation in the Veins simulator. A more detailed listing of these criteria, as well as a more detailed description is found in Section 3.6.

1.5 Related Work

The traffic light algorithm of the reliable intersection protocol proposed by Azimi [1] and Savic et al. [40] are used as the base for our algorithm. The algorithm we propose can be seen as a fault tolerant vehicle-to-infrastructure (V2I) inspired counterpart to the one by Azimi [1]. We entail a queue system with only a single grid square.
As with earlier examples, we aim to implement a system which behaves similarly to a virtual traffic light as explained by Hagenauer et al. [26]. This algorithm was evaluated using the vehicular network simulation framework Veins which was developed and published by Sommer et al. [43]. This tool can be used to evaluate our algorithm since it is specifically developed to handle the combination of network and vehicular simulations.
There are examples concerning cooperative collision avoidance between V2I through a centralized supervisor as explained by Colombo et al. [11]. This centralized supervisor, which in that case would be placed in the intersection, would gather information about the vehicles and then decide whether or not to adjust heading or velocity of the vehicles. However, since the communication could fail, i.e., a message was lost, there could still be a collision within the intersection. This could happen since the vehicles inside the intersection do not necessarily expect a message for adjusting themselves, and as such, will not be notified of a possible message loss.
Our algorithm has a recovery procedure that is influenced by the self-stabilizing attribute as seen in the algorithm by Dolev et al. [14, 42, 13, 23, 22, 21, 20]. This means that every time the system enters a new state of the recovery procedure, every other node in the system has to agree to and verify that all other nodes are entering the same state before actually committing to it.

1.6 Our Contribution

We are proposing a protocol and we detail its implementation. This protocol detects failures that are due to system and communication failures as well as state and message corruption. We study an automated fundamental driving component which is the VTL. We state a number of evaluation criteria that include a broad range of failures such as normal and abnormal positioning errors, packet omissions, and communication delays. The system will be evaluated via extensive experiments in the Veins (SUMO and Omnet++) simulator. When conducting experiments in the simulator, the following key trade-offs were established.

- Making vehicles stop further from the intersection reduces the negative impact of positioning system inaccuracies.
- A higher message transmission frequency acts as an effective remedy against severe packet omissions.
- Raising the frequency of message transmissions improves the recovery time of the system.
- Having a frequency of message transmissions that is too high causes network congestion that severely affects the performance of the system.

As such, our fault tolerant VTL can be a fundamental building block for future automated driving systems, completing existing autonomous driving efforts. To display this, we also present a prototype implementation with an evaluation of this algorithm.

1.7 Scope

The goal of this thesis is to provide an efficient and reliable group communication for a vehicle system that can deal with severe yet fair asynchronous messages passing. The system does not tolerate crashes of either the server or the clients. The motivation to our work is the application domain of traffic formation and intersections in particular. We aim to provide an efficient and reliable group membership and multicast services that depend on the availability of the asynchronous infrastructure. However, the system will still be available and fully functioning, albeit less effectively, while experiencing severe packet delay, positioning system inaccuracy and packet omission.

2
Background Knowledge

This chapter describes the background knowledge needed in order to more easily grasp the core ideas and concepts of this thesis. These include intersection crossing algorithms, communication between vehicles, the idea of the capture zone, creating agreement between client and server, fault models and fault injections. These sections each go into details surrounding their respective subjects as well as include the most closely connected references to give a quick overview of what to expect later on in the thesis report.

2.1 Intersection Crossing Algorithms

Just like in [1], the paper by Savic et al. [40] presents a decentralized intersection crossing (IC) algorithm. While there is no intersection manager (IM) in this implementation both safety and liveness is considered. However, only two vehicles are considered. Yet, it is important to notice how they deal with a large unknown number of communication failures. What is really interesting is how we could combine the proposed vehicle-to-vehicle (V2V) algorithm with our own V2I algorithm. At least we consider the fact that there are multiple ways to make an intersection crossing more efficient, while still keeping the algorithm safety critical. Just like in [1, 40] the intersection could be built up by multiple parts to allow vehicles that are not heading in the same trajectory to cross simultaneously. Thus we aim to keep our algorithm as modular as possible to allow more efficient and more fault tolerant approaches to be used in combination with our algorithm.

2.2 Vehicle Communication

The choice of communication between vehicles can vastly affect the system. By ensuring fair communication between each vehicle and the server will increase the throughput of the algorithm while maintaining its safety criteria [41, 34, 29, 19, 18, 46, 27, 31]. However, the proposed algorithm is not limited to V2V communication and can utilize any form of communication that would allow each vehicle to communicate with the allocated server. In fact, this makes it possible for the server to not be present at the intersection at all. Communication failures can create safety concerns within the system [30, 16, 33, 2, 3, 8, 35, 10, 4, 36, 9]. As a result of this, the algorithm is designed to withstand many of these potential issues as described in Section 3.4 Requirements.

2.3 The Capture Zone

By building for the usage of a capture zone as described by Azimi [1], we can assist vehicles to get a better understanding of what velocity to keep when closing in on the intersection. It is therefore necessary to have a basic understanding of what the capture zone is used for to understand how we support it.
As such, the algorithm is built around supporting a capture zone as explained by Hafner et al. [25]. This would further help vehicles from accidentally entering the intersection and increase throughput.

2.4 Agreement Between Client and Server

A large part of our algorithm is based on the self-stabilizing technique as shown by Dolev et al. [14, 12], Brukman et al. [7] as well as the ones presented in [15, 39, 38, 17]. In particular, the segment about having all involved processes agree upon a state change before committing to the change. This is done by having every process first confirm they have seen the suggested change, after which every process verifies they have seen the fact that every other process have agreed to the change. Once every process has seen the last verification, they change into the new configuration.

2.5 Fault Models

Another important aspect is to understand the concept of a fault model as described by Raynal [37]. This is useful to get an understanding of failure models and what type of failures exist and how these models can be used. The fault model is the base upon which the project is built. This is used as a base line when developing and testing the algorithm as a means to see if the algorithm performs as planned. In this project, we will consider communication and process faults, both of which are described further in Section 3.3 Fault Model.

2.6 Fault injection

This system is tested by implementing the algorithm in a simulator that utilizes virtual traffic lights. This simulator has to support both network communication simulation and traffic simulation. Such a simulator is described by Hagenauer et al. [26] and Sommer et al. [43] where the concept of virtual traffic light computation and simulating such environments are described. This simulator is in fact built in three parts, one network simulator, one traffic simulator and an interface connecting the two to support to cooperation between them. This makes it possible for us to test our algorithm both with respect to potential network communication issues as well as the aspect of it being used by vehicles traversing an intersection.

3
The System and its Requirements

The following chapter is meant to provide a better understanding of surrounding factors, such as the system components, definitions we use, assumptions we make, the fault model we handle, requirements of the application, the evaluation criteria, test cases and demonstration tools.

3.1 System Description

We start by presenting different aspects of the system and its parts. This includes both the vehicles and the subsystems that manages the intersection crossing. Furthermore, we provide several definitions for important concepts that are central to both utilizing and understanding of the presented algorithm.

3.1.1 Definitions

The following section details definitions used in this thesis. These include ℓ-exclusion and the various zones used by the algorithm. The zones are a core concept of how the algorithm arbitrates vehicles. Each zone is connected to a specific purpose and vehicles will act differently depending on which zone they are currently in. The different zones can be seen in figure 3.1 where we display an example of an intersection.

ℓ-exclusion

At any time, at most ℓ vehicles are allowed to be in the intersection, and in extension the Capture Zone. Every vehicle has to eventually be allowed to enter the intersection and if ℓ is unbounded, a real time system cannot be implemented. The motivation for modeling this problem as an ℓ-exclusion task comes from the fact that a collision avoidance mechanism requires reliable high throughput and real-time message exchange in mobile ad hoc networks.

Registration zone

This is the area that surrounds the joining zone. It allows vehicle to both register when they entered this zone, and report that time to the server. Thus, the server can detect which order the vehicles arrived in. The main focus with this zone is that every vehicle should have been registered at the server before reaching the joining zone since at that point the server assumes it has all knowledge necessary to start organizing which vehicles should be allowed to traverse the intersection together.

Joining zone

This is a predefined area that surrounds the Critical Zone, Capture Zone and some surrounding area meant to let the vehicle be able to connect to a server and request entrance to the intersection. This area should be large enough for the vehicle to have enough time to get through the process of gaining entrance to the intersection while maintaining a flow of the system. The client will have an interface that returns true if the client is currently in joining zone. The Join Zone is represented as the road in Figure 3.1.

Critical Zone

This Critical Zone is the zone in which there are at most ℓ vehicles allowed. In other words, the algorithm is designed to be a resource sharing algorithm. In our examples the Critical Zone is represented as the central part of the intersection, where the lanes intersect. As a result of being designed as a resource sharing algorithm, the Critical Zone could be something else, such as a one-way road that can change which direction vehicles are allowed to go. The Critical Zone is represented as the striped square in Figure 3.1.

Capture Zone

This is a relative area in which size is dependant on vehicle velocity that measures distance to intersection of which the vehicle is not allowed to enter unless cleared. Once a vehicle enters the Capture Zone, it is not possible to guarantee a braking that will prevent the vehicle from entering the intersection area. In other words, entering the Capture Zone is no different than entering the critical section itself. Note that we can define the Capture Zone in way that considers convenient braking. Examples of Capture Zones can be seen in [40] and [25]. Just as with the interface for the joining zone, a client will also have an interface that returns true when the client is in either the Capture Zone or the intersection itself. These two share the same interface since the Capture Zone is essentially an individual extension of the intersection. The Capture Zone is represented as dotted rectangles in Figure 3.1.

Departure zone

This is a zone depicting the same area the joining zone could, with the exception that the vehicle is assumed to have visited the Critical Zone. Once the vehicle has accessed the Critical Zone, the vehicle will then exit into the departure zone. The departure zone is represented as the same circle as the Join Zone in Figure 3.1.

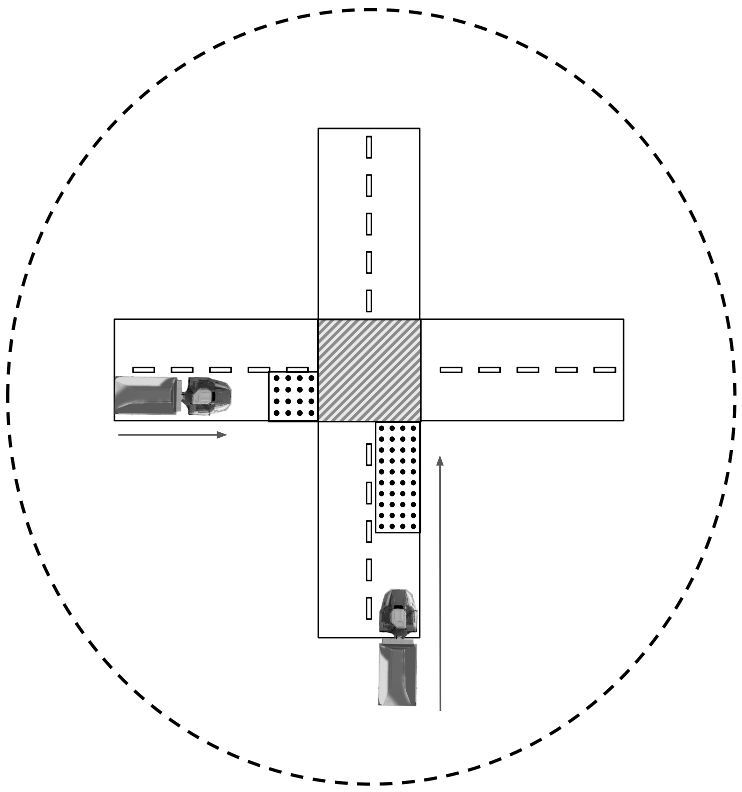


Figure 3.1: An overview of the zones. The dotted lines surrounding the intersection represents the registration zone and the start of the road represents the join- and departure zones. The dotted rectangles are the individual Capture Zones and finally, the striped square represents the critical section.

3.1.2 System Architecture

To utilize the centralized solution, there needs to exist clients and a server. The system is composed of vehicles that are trying to cross an intersection, thus, in this system they are the clients. We have chosen a centralized solution for this project where a server will grant these clients access to at most ℓ vehicles at a time. This server could in reality be a set of servers that work in a replicated manner to ensure availability. In this project however, we chose to work with a roadside unit (RSU) as the given server. This is because it makes the implementation in the simulator easier to handle since the RSU can easily be assumed to be a server handling the connections of vehicles entering and leaving the intersection. The server and client has a interface for them to interact through, which we depict in Figure 3.2.
Both the vehicle and the RSU have dedicated short-range communications (DSRC) devices, (IEEE 802.11p). These are used both by the vehicles to send queries to the RSU and for the RSU to respond with.
We assume that every radio unit have access to a unique network identifier and that the network identifier of the road side unit is well known to all vehicles.
Each vehicle is equipped with a positioning system that is precise in the sense that it has a known bounded error. The positioning system provides input to a region management component that uses a static map for notifying the client whenever

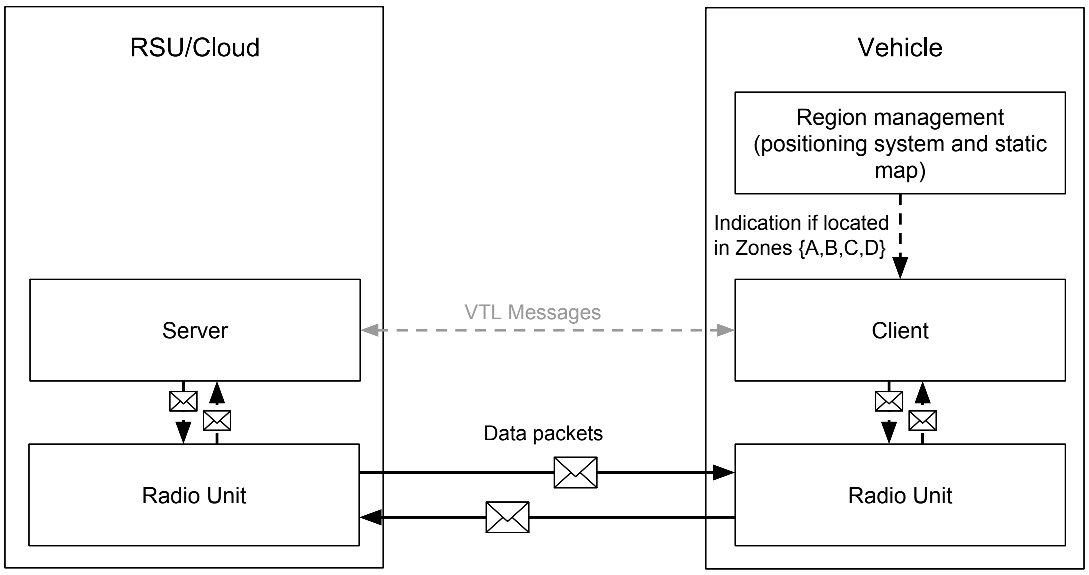

RSU/Cloud
Vehicle
Region management (positioning system and static map)
Indication if located in Zones {A,B,C,D}
Server
VTL Messages
Client
Data packets
Radio Unit
Radio Unit

Figure 3.2: Architectural overview of the subsystems and their data flows. The figure shows how the client and server exchanges data, while only the client uses the map and positioning system to determine what action to take.

it enters a new zone, which are the joining, critical and departure zones (Section 3.1.1). For example, this allows a vehicle to detect that it is in the joining zone and then to start contacting the server.
Recall that the zone interfaces are defined in a way that every vehicle moves through the zones in the following order: registration zone to Join Zone to Critical Zone to departure zone. Apart from this, the positioning system should show the correct position of a vehicle within a known amount of time. Breaking this time limit is considered as a failure (Section 3.3).
For the system to function there are a few components that are absolutely needed. Other than the server and client parts which interact there needs to be a interface for them to interact through. In Figure 3.2 there is a description of what each subsystem requires. It can be summed up to a computer system and a communication interface, in this case a radio unit. In addition, the vehicle needs to have an accurate positioning system with a static map. This map locates each (in the scope of a single intersection) each of the previously mentioned zones that are also defined in Section 3.1.1. These are the minimum architectural requirements in terms of a functional system.

3.2 Assumptions

We make the following assumptions about the system behaviour.

1. Communication Fairness. Suppose that a sender sends a packet infinitely often. The receiver will receive a packet from that sender infinitely often.
2. We assume that the server and clients are always alive and connected. There could still be failures and recoveries, but our assumption is that they will always recover eventually.

3. Only vehicles that are using the system are allowed to enter the intersection. This means that no unauthorized or unconnected vehicles will enter the intersection. This does not mean the connected vehicles respect the rules of moving through the zones. As long as the vehicle is connected, the algorithm is meant to mitigate further issues and self-heal in the case of such a violation.

3.3 Fault Model

We perceive failures as steps that the environment, rather than the algorithm, takes. In each such failure step, the environment causes the system state to change, i.e., it manipulates the messages in the communication channels or changes the state of a process. We define the failure model as a set of all possible failure steps. Note that when we are given a run of the algorithm, we can note faulty processes in that run when their behavior deviate from the one that the algorithms defines.
Below, the fault model is presented:

1. Considered failures in the **communication channels**:

   (a) **Packet omission.** We consider cases where a faulty process intermittently omits to send messages it was supposed to send or receive. This could for example be caused by a full message buffer or due to the nature of wireless communication, where the message might never reach the intended receiver. A common way to handle omission is the use of acknowledgements and re-transmissions, as used by protocols implementing ARQ[1]. An omission could however, still cause a delay without being considered as a normal omission.

   (b) **Packet delay.** We restrict the possibility of omissions using the assumption of communication fairness. That is, we assume that if a sender sends a packet infinitely often then the receiver receives that packet infinitely often. Thus, the packet omission would appear as an unbounded, yet finite message delay in our case.

   (c) **Packet duplication.** We consider the case in which the sender achieves reliable communication via re-transmission of packets until it receives acknowledgment (either explicitly via ARQ mechanisms, as it is in TCP/IP, or at the application layer). Such re-transmission can cause packet duplication and thus they are considered. If a system is not set up to handle the same message data, duplicated messages can appear to be two different requests which could incorrectly alter the state of the recipient. Re-transmission can be handled as packet delay as well.

   (d) **Incorrect message reception.** There is the possibility that a process receives a message not intended for itself but another process. This is due

[1]https://en.wikipedia.org/wiki/Automatic_repeat_request

to the fact that the system will use V2I and most likely 802.11p making every process within range receive the message. This will in turn delay the process from dealing with other messages. We can therefore translate incorrect message reception to be a message delay.

(e) **Corruption of message data.** We consider the case where the contents of a message is corrupted. We assume the availability of mechanisms for checking the packet integrity, e.g., checksums using CRC codes. We note that such mechanisms cannot guarantee data integrity in all cases and thus the application has to further test messages upon their arrival. This is why it is important to offer resilience to these kinds of errors, albeit rare. However, because of how the algorithm functions, this will in the end only cause packet delay since there will be some delay before the system works out what message to actually consider.

2. Considered **process** failures :

   (a) **Positioning system errors.** We consider the possibility for a client to be erroneous when calculating its position through either GPS or other means. However, we also make the assumption that the positioning data will eventually show the correct position. Since a strong signal is not always guaranteed, we have to consider the case where positioning through means such as GPS will become temporarily inaccurate. However, the initial GPS position of a vehicle is assumed to always be less than half the distance to the closest vehicle driving in the same direction. Without this assumption, vehicles can end up out of order, which is not considered when vehicles drive in the same lane.

   (b) **Illegitimate Capture Zone entry.** We consider the case where a vehicle enters the Capture Zone without permission, this can happen due to a human driver that does not respect the instructions of the the traffic light. This could also be due to an error of the automated driver as well as cases in which the vehicle experiences a brake failure and enters the Capture Zone when it is not allowed. It could also be caused by an incorrect positioning system. Both of these cases would be a violation of the assumption, i.e. an unauthorized vehicle enters the intersection. These are important to consider, since these are accidents where there is no way to guarantee that there are only ℓ vehicles at a time when an accident like this occurs. However, what we can do is mitigate these hazardous situations by first detecting that there is a problem, and then warn other vehicles in the system. We also want to stop any new vehicles from entering until the problem is resolved.

   (c) **State corruption.** We consider the case where the internal state of a process has been corrupted because of some arbitrary reason such as a corrupted message that was delivered or due to a process reboot, thus altering the internal state. Since this would change the behaviour of the system outside of the given system definitions, we need to make sure the system can recover from these kinds of errors.

3.4 Requirements

The algorithm has to support a set of requirements, some of them benign and some of them based on the fault model. The benign requirements explain what the system should handle when there are no failures while the fault model requirements will go into more detail of how the algorithm should handle the different faults listed in section 3.3.

3.4.1 Benign Requirements

The following section lists the requirements of our algorithm while there are no failures.

1. **Safety requirement:** While there are no physical faults, the intersection and Capture Zones should never have more than ℓ vehicles inside the Critical Zone.

2. **Liveness requirement:** The algorithm will eventually allow every vehicle requesting to cross the intersection to cross.

3. **Timing requirement:** If no failure occurs, messages will arrive within a known amount of time.

3.4.2 Fault Model Requirements

The following section is referencing to section 3.3 when explaining different requirements.

1. Communication channels:
   (a) Whenever an error in the communication channels occur, it can be summed up to be a message delay due to the design of the system. Thus, the requirements for this system is that each of the faults in the communication channels should lead to being treated as a message delay. The delay itself should simply be resolved when the next message arrives from either a client or the server. While the system may encounter these issues, it will keep vehicles from entering the intersection until it is resolved, thus we encounter a delay.
   The reason behind dealing with these issues as a message delay is due to the fact that in the case of a violation, the message will in most cases ignore the message and at worst lock down the system until every vehicle and server has reached agreement again after which is will resume normal traffic. This means that in any case, the possible faults can be translated to be message delays since there is a delay before either the server or

vehicle receives the initially intended message.

2. Process failures:
   (a) Whenever an illegitimate Capture Zone entry (2b) occurs, the system switches to recovery mode. During recovery mode, no new vehicles are allowed to progress through the intersection and all vehicles will attempt to leave or stay away from the Capture Zone. There are checks and mechanisms in effect to mitigate the effects of these errors but in the case where there is a violation, the system could enter recovery mode. The system returns to normal mode after all vehicles have left the Capture Zone and coherence is achieved.

   (b) When, (1e), (2a), and/or (2c) happens, the process should initiate the recovery procedure. It should stay this way until the issues are resolved and it is no longer detecting any errors.

   (c) If the combined faults of either (1a) or (1e) with (2a) occur, where a client incorrectly thinks it is in the wrong position while the messages to the server are corrupted or lost. In this case, the client would act according to its state while entering recovery mode and wait until it starts receiving the expected messages again. As described earlier, this case is also handled by checks and mechanisms to help mitigate many of these errors.

3.5 Evaluation Environment

We evaluate the algorithm using Omnet++[32], SUMO[45] and Veins[44] which allows us to simulate a traffic environment with network connections between vehicles.

To allow our simulation to run we have modified the veins framework to allow our own types of messages containing custom fields and data. This is done to make it possible for us to specify exactly what we wanted to send with each message and to store our own classes of data.

In each of the traffic simulation scenarios there is an intersection with traffic arriving and leaving in four directions. In our implementation we followed the idea by Hagenauer et al. [26] to first remove the dependency to traffic lights. This is achieved by telling vehicles to disregard a red light if the algorithm grants them access to the critical section. In Figure 3.3 the modeled intersection can be seen. The cars are represented by triangles. The cars approach the intersection, and when the server gives them permission they cross it. Once they have left the intersection the clients contact the server to remove its access once again, effectively leaving room for a new client. While SUMO handles how the cars behave in traffic OMNET handles the network simulation. Figure 3.4 shows each of the zones including the Capture Zone for the closest vehicle in each direction. It also shows the cars position and their current status.

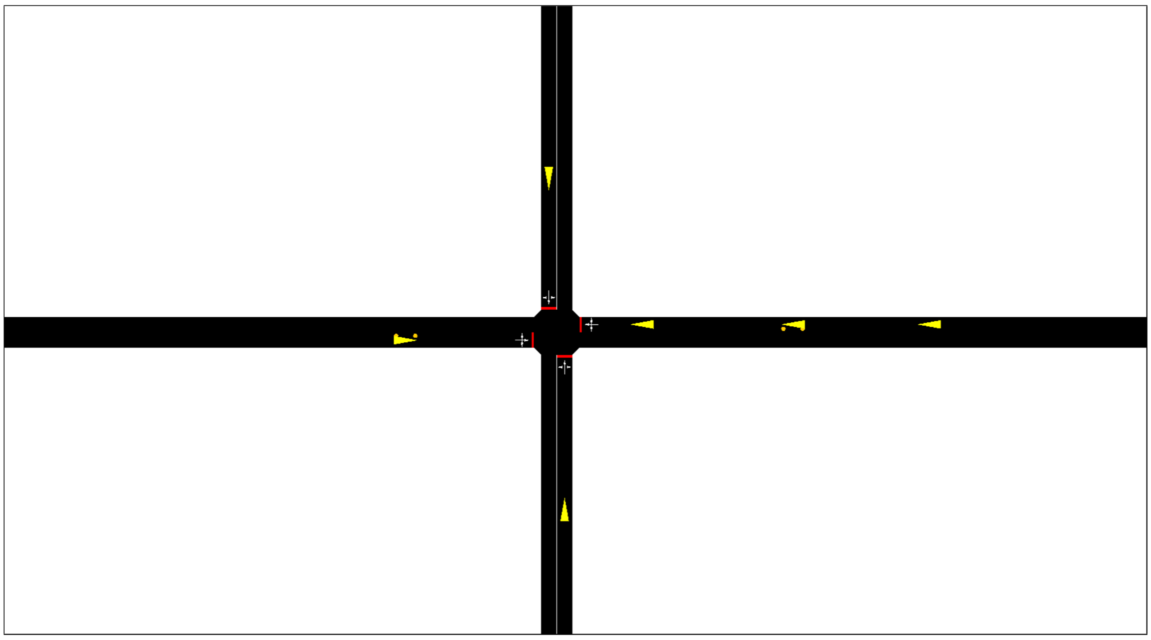


Figure 3.3: The intersection we are using to test the algorithm as it modeled in SUMO

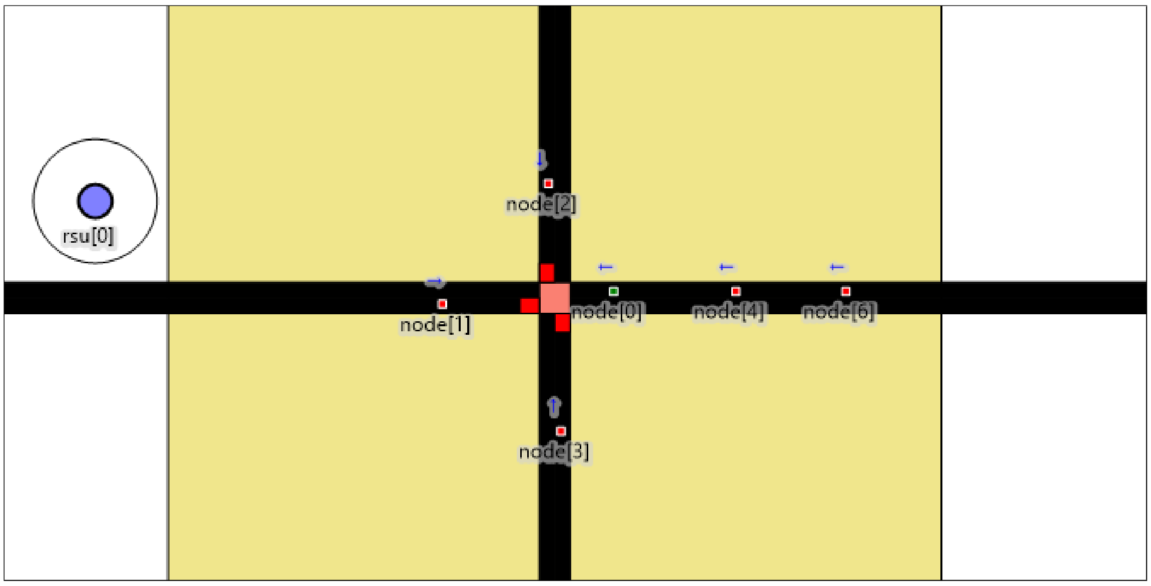


Figure 3.4: The intersection that is used to test the algorithm as it is modeled in OMNET++. The figure also shows the current state of each vehicle as a color, where red is a vehicle waiting to be granted access and green is a vehicle granted to cross the intersection, as well as the Join Zone and Departure Zone in yellow, the Critical Zone in pink and the Capture Zone in red

The algorithm is focused on using a centralized structure where each vehicle talks to a server before entering the critical section. However, as soon as a vehicle has been granted access to enter the critical section, it can not be revoked and the vehicle must be assumed to have entered.

The system needs to be able to recover from possible errors which are described in the evaluation criteria. While the algorithm is performing a recovery, it should not allow any normal progress for vehicles except let any vehicles already inside the critical section exit.

3.6 Evaluation Criteria

To evaluate the system there have been different criteria set for the system, and they are each listed below. These criteria exist to verify the functionality of the system, and to find interesting trade-offs in the system design.

- Measure the throughput of the intersection for twenty vehicles without any failures with different variables such as margins between the Join and Critical Zone. These margins are measured between 0 to 60 metres.

- Measure the recovery time from a triggered injection of the failure detection mode for different amounts of involved vehicles (5, 10, 15, 20). In other words, how long it takes from a first vehicle detecting an error in a message to the complete system returning to a normal state.

- Measure the throughput of the intersection with message delays where many different injected delays are measured between 0 to 6 seconds (On average 200ms between each measured value).

- Measure the throughput of the intersection with packet receive omissions where many different injected drop rates are measured between 0 to 90%. (On average 3% between each measured value).

- Measure the impact of transmission rate of packets on the system, both with and without failures. Locate the break-point for the system, when and how much does network congestion affects the throughput.

3.7 Test Cases

The testing is done with the Veins simulator, by setting up different scenarios to see how the algorithm handles them. In the case of a fault model violation, the system should still maintain as much safety as possible. Below, the different test cases are listed. They are each carried out in respect to the evaluation criteria.
Firstly, it is important to verify that the system works as intended by supporting the different criteria that we set for this project. Not only that, but also how well the algorithm handles different loads on the system. Here we will measure the throughput and verify that the system works in different configurations.
Secondly, the system has to satisfy the criteria and the requirements that is set up in accordance to the failure model. In addition to basic requirements, it should be able to handle a number of faults, as described in section 3.4. We want to verify that the system handles both delays and faults that would put it in recovery mode. Of

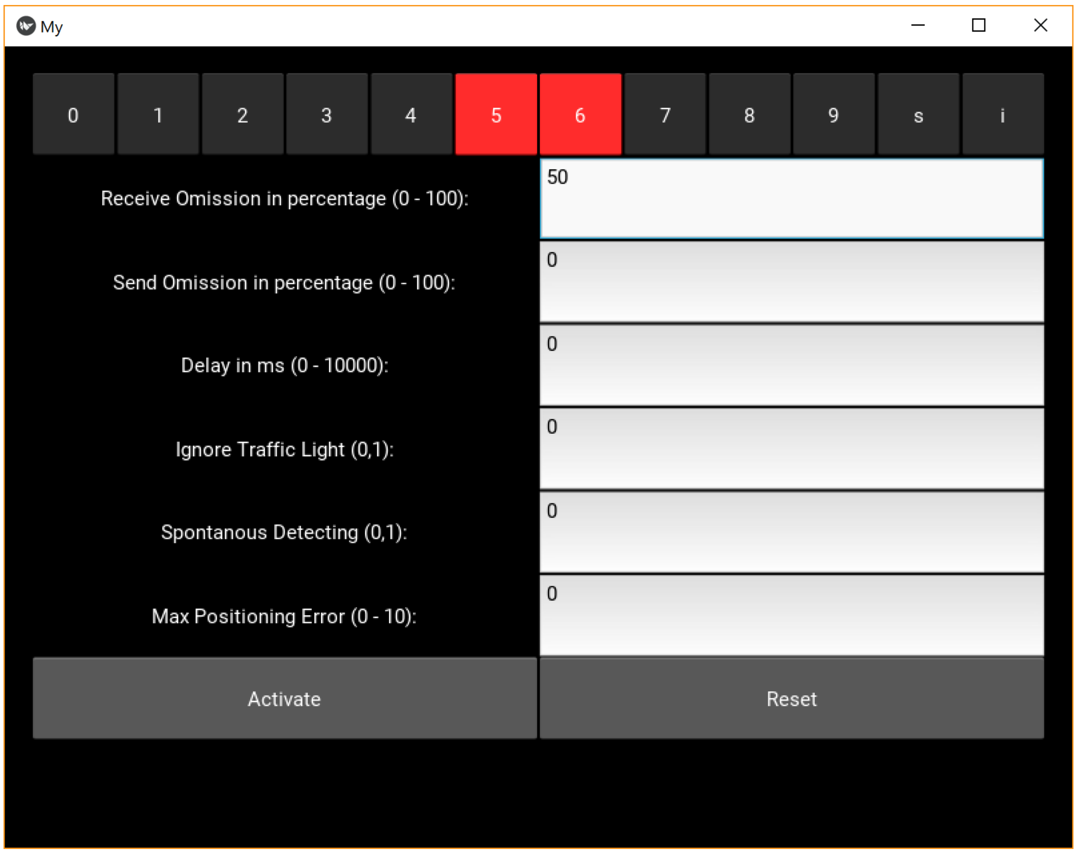
course we also measure recovery time from both types as well as how the throughput is affected. This is a great opportunity to verify that the system correctly recovers as it was designed to do.
Finally, there are also a few different demos that are covered in section 3.7.1. These demos focus on showing the capabilities in the system instead of being used as data collection points for the evaluation of the fault model.
Below, the test cases are summarized to give a brief overview of what each of the test groups entails. Each test segment contains more than one test and will be referring to the evaluation criteria in section 3.6. It is important to notice that these are all tested with a few variations of the actual ℓ-exclusion values.

- Verify that the algorithm works as specified and measure the throughput the system **without any injected failures.**

- Verify that the algorithm works as specified and how the system behaves **with different injected delays and a recovery mode-triggered failure,** and measure the time it takes to recover as well as the throughput.

3.7.1 Demonstration in Front of an Audience

Apart from our own run experiments, we have developed a small interface to demonstrate the properties of the algorithm. This interface combined with the simulation makes us able to change variables in run-time. Figure 3.5 shows the graphical user interface (GUI) that was developed to use when demonstrating.
This is useful to show how the algorithm reacts if something were to happen in the middle of a run, such as a vehicle which starts dropping messages and how the system works around that. This is done to graphically show the adaptation the system makes and the consequences that occurs as a result.
The demo interface itself is a tool built with buttons and text boxes that a user can use to change variables when running the simulation. The user can choose what variables should affect which vehicles/server and with what value.



Receive Omission in percentage (0 - 100): 50

Send Omission in percentage (0 - 100): 0

Delay in ms (0 - 10000): 0

Ignore Traffic Light (0,1): 0

Spontanous Detecting (0,1): 0

Max Positioning Error (0 - 10): 0

Figure 3.5: The Demo GUI with buttons and text boxes, allowing users to change variables at run-time for up to 10 vehicles plus the server. The example shown in the figure has the user selecting vehicles 5 and 6 while putting their receive omission at 50%.

4 The Protocol for Arbitrating Intersection Crossing

Facilitating a traffic light can be done by defining a queue of vehicles. Through this we can control the traffic flow together with ℓ-exclusion. Once the defined amount of vehicles for that specific capture zone enters, no other vehicle should enter and once a vehicle has left the capture zone again we can discard any information about that vehicle and allow new vehicles to enter. The following chapter explains how this can be achieved in more detail. First by going through an initial version and then adding layers of functionality on top.

4.1 Core algorithm

The initial algorithm is developed to resemble a normal traffic light, except that it is virtual. In other words, no physical traffic light infrastructure would be needed, instead wireless communication, positioning systems and computers act together to solve the same problem that a physical traffic light would. Of course there are some key differences and additions, but many of the systems are in place to act like a physical traffic light does in principle. These additions include the sensors that a physical traffic light uses, as well as the physical queues that essentially is a First-Come, First-Served (FCFS) that naturally happens because of the way cars line up in a lane. For instance the sensors that tell physical traffic lights about the presence of a vehicle is in our case replaced by predetermined GPS-coordinates that the vehicle compare to their own coordinates. To clarify, a vehicle detects when it enters the vicinity of the intersection and tells the server that it wants to cross.

The state transition of a client in a single server is shown in Figure 4.1 and the work starts with a fully centralized system where we expect no message drops, no packet loss delay, perfect positioning readings and full up-time for the server and clients. The modifications are focused on implementing a queue system which aims to let vehicles cross the intersection according to the ℓ-exclusion definition and all previously stated goals. To support this, we suggest a form of state transition system described by changing colors. This further supports the use of unique tags for each connecting vehicle that shows which order they connected to the server. These tags are used to determine a FCFS-queue system for the server to prioritize which vehicle is next in line to be allowed into the intersection.

4.1.1 Pseudo code

To explain the core algorithm more clearly we have written pseudo code representing the initial state of the algorithm. Both the server and client parts of the pseudo code can be seen in Algorithm 1.

Algorithm 1: Pseudocode for Server and Client

```
types:
$\mathcal{T} = \{0, 2^{64} - 1\}$;
$ID := \{0, N - 1\}$, where $N$ is an upper bound on the number of possible client identifier;
$Colors = \{red, green, blue\}$;
constants:
$myID$, the unique identifier for a process in the domain $\mathcal{D}$ of all possible identifiers;
$\ell$, the maximum number of vehicles that allowed to get a green light concurrently;
macros:
addQueue($Q, m = (t, s, j)$) := Adds element to Q. Will overwrite if tag $t$ or clientID $j$ exists in Q;
commitChange($m := (tag, (color, \bullet), clientID)$) := begin
    if $(color = red) \wedge$ clientID exists in Q then $tag \leftarrow$ (tag belonging to clientID in Q);
    else if $(color, tag) = (red, \bot)$ then $(tagCounter, tag) \leftarrow (tagCounter + 1, tagCounter)$;
    if $color \in \{red, green\}$ then $addQueue(Q, m)$;
    else if $color = blue$ then Remove everything with tag and clientID from Q and greenSet;
    return $m$;
interfaces:
$joinZone()$, true if the vehicle is in the joining zone, else false
$afterZone()$, true if the vehicle is in the after zone, else false
server part begin
    variables:
    $greenSet := \emptyset$, a set with all (tag, clientID) that are valid to be green;
    $Q := \emptyset$, a set containing information of all connected clients
      $(tag \in \mathcal{T}, color \in Color, clientID \in ID)$;
    $tagCounter := 0$, a long integer;
    $pending := \emptyset$, a list of messages;
    do forever begin
        let $toSend := \emptyset$;
        foreach message $(m_c, \bullet) := ((tag, color, clientID), \bullet) \in pending$ do
            $toSend \leftarrow toSend \cup \{q := commitChange(m_c)\}$;
        while greenSet is not full and there exists tags not in greenSet do Add smallest tag not
          in greenSet to greenSet;
        foreach message $m := (\bullet, clientID) \in toSend$ do  $send(m, greenSet)$ to $p_{clientID}$;
    upon arrival of message $m = pending \leftarrow pending \cup \{m\}$;
client part begin
    variables:
    $myTag := \bot$, a tag (a queue number at the server) where $myTag \in \{\bot\} \cup \mathcal{T}$;
    $myColor := red$, current color of the client where $myColor \in Colors$;
    $pending := \emptyset$, a list of messages to be processed
    do forever begin
        foreach message $(m_c, m_s) := ((tag, color, id), \bullet) \in pending$ do
            $(greenSet, myTag) \leftarrow (m_s, tag)$;
        if $(myColor = red) \wedge (myTag \in greenSet)$ then $myColor \leftarrow green$;
        else if $afterZone()$ then $myColor \leftarrow blue$;
        send to the server $((myTag, myColor, myID), greenSet)$
    upon arrival of message $m = pending \leftarrow pending \cup \{m\}$;
```

The pseudocode separates general types and constants from line 1 to 7, the server part from line 19 to 31 and client part from line 32 to 43.
The pseudocode in Algorithm 1 is the base that is needed to arbitrate vehicles through an intersection. This code does not consider any type of faults and simply has a server receiving requests from vehicles that want to cross.
The main part of the server is in its main loop, which continuously cycles, starts at line 25. It starts by clearing what should be sent to the clients on line 26 after which it loops through all received messages since its last loop on line 27 to 28 where it also calls the macro commitChange that can be seen on line 10.
The macro commitChange consists of five lines where firstly it assigns a tag to the client. The server checks if the client already exists in the queue and assigns that tag if it does. If it is a new vehicle, the server creates a new tag and assigns it to the vehicle as seen on line 12. The server then checks if the vehicle should be added/updated in the queue or removed from the queue depending on its color. The current state of the vehicle is then returned to the main loop and attached to a new message that will become part of the response to that client.
After this, the server assigns valid vehicles to a green set on line 29 which, in this pseudo code, assigns depending on arrival order of the vehicles. These updates are then sent to the vehicles as seen on line 30.
The client's main loop starts at 37. Clients loop through every received message from the server and updates its green set and tag as seen on line 39. This is then used to decide whether or not the vehicle is allowed to change to a green color on line 40 and thus allowed into the Critical Zone.
Once a vehicle has passed into the Departure Zone, it will change its color to blue on line 41 in order for the server to be able to remove the vehicle from the green set as seen on line 14 and in extension become able to assign new vehicles to the green set.

4.1.2 Color Transitions

We suggest a color transition model that can be seen in Figure 4.1. Each client initially assume the red color state and from there it transitions in the order that the figure shows. When a client is any other color than green, it is not allowed to change between Join Zone to Critical Zone or Departure Zone. From figure 4.1 we can define three different states a vehicle can assume in the queue system for the client. These are described in the following list.

- Red - A vehicle that is approaching the intersection and has entered the Join Zone sends a request to enter the intersection. The vehicle receives a session identifier from the server once the server has been contacted, and this session identifier represents a queue ticket. The vehicle waits as Red until it receives a set of identifiers of which itself is included. This set is what we call the green set. If the clients own identifier is a subset of this set the client is allowed to be green and will thus change from red to green. In other words, the client's queue ticket is at that moment valid to use.

- Green - Having changed to Green, the client is allowed to enter the capture zone and in extension the critical section since it is now at the beginning of the queue. This is where the ℓ-exclusion takes place. Only ℓ amount of vehicles are allowed to be Green at a single intersection at any given time. After leaving the intersection, and thus, entering the Departure Zone the client switches to Blue.

- Blue - A Blue client is a client that has left the intersection. The client is then either in the Departure Zone or outside of all of the zones. A client can safely disconnect once the server has acknowledged his blue state by responding with a message where he confirms that the client is indeed in a blue state. When this happens, the server is acknowledging that the client has left the system. Also, the clients queue ticket (session identifier) is no longer valid to enter the intersection again.

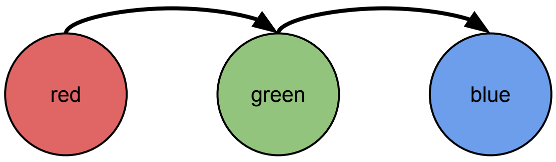


Figure 4.1: An overview of the vehicles' states their representation as a state machine. For simplicity, we display each state as a color where red is a client that has not yet been granted access to the intersection, green is a client that has been granted access and blue is when the client has passed the intersection and is telling the server to revoke its access in order to grant a new client access.

4.1.3 Network Communications

The client sends packets to the server which contain certain fields of data, including a color from the group of colors shown in figure 4.1. When the server receives packets from the clients it continuously tries to assign new clients to the green set. Initially there are only two criteria that needs to be fulfilled for a client to be placed in the green set. The first one is that the green set has to be smaller than ℓ and the second criterion is that the client has to be red with the smallest tag, since the tag is used to define the FCFS-structure. If the defined criteria are fulfilled, the client is added to the green set and the client is notified.
The server keeps track of which vehicles are allowed to become green in a separate set, which it sends to each vehicle every time it responds.
A setup of several replicated servers that would collaborate would have to make sure they hold the latest information before making a decision and make sure all servers agree before allowing a request from a client.
Assuming that the server can maintain the information of these states for each vehicle and correctly put each in a queue, specifically red vehicles, the problem of supervising the intersection would be solved even if they lose connection to the server while attempting to regain connection. By then changing states between vehicles as explained above, we would solve the supervision of crossing the intersection without collisions as well.

This algorithm also works when there is network delay since a client always needs permission from the server before continuing into the intersection. This means that even if messages get delayed, the only thing that it will affect is delaying vehicles from progressing through the Critical Zone. The requirements for the intersection will still hold.

4.2 Fault-tolerant Algorithm

Since basic functionality such as giving vehicles access to the intersection works, the focus became making it more robust for possible failures (see Section 3.3). This leads us to the recovery mode which exists to help clients and servers keep track of agreement of states between clients and servers. This section will go into detail of how we dealt with packet delay, the failure detection, what the recovery mode is used for, decreasing the time in which the system stays in recovery mode, the addition of a Registration Zone and how we optimize throughput by adding a priority module when assigning green vehicles.

4.2.1 Packet Delay

In the most basic sense, the algorithm is built around arbitrating vehicles through an intersection while keeping it safety critical. To accommodate the safety critical attribute, most of the possible failures have been simplified to behave like packet delay.

It is possible to deal other errors this way due to how the algorithm works. The clients send out a new request to the server in every execution of the main loop and the server responds to every request it receives. This execution loop occurs at a predetermined rate, and leads to quite a large number of messages being sent. However, it means that if a message is omitted, or corrupted for instance, that message can be disregarded and thus the next received message will soon arrive instead. This raises an important aspect of the proposed algorithm, namely, failure detection. A more detailed explanation of the failure detection can be found in Section 4.2.2.

Thanks to the failure detection capability most erroneous messages are essentially translated to delays since if any occur, the server does not consider the contents of message and deals with the next correct message. While the server does not consider the contents to process them it will react to the faulty message by entering recovery mode until the client stops sending faulty messages. If the client receives a faulty message, it will drop the message, perform a new request in its next execution of the algorithm loop and start dealing with the responses once they are correct again. If either the clients or the server does deliver a corrupted message, the system will enter recovery mode and eventually reach normal operations again. This in itself is also a delay as it will not break the system, but only delay the throughput of vehicles until the issue is resolved. Since vehicles always need permission from the server before continuing into the Critical Zone, the algorithm is securing the intersection from possible violations while delaying the throughput.

4.2.2 Failure Detection

To detect various failures one first needs to predict which behaviours these failures would generate in the given setting. This connects back to the failure model in Section 3.3. Due to the servers ability to know what to expect from a client it is possible to detect whenever a client does not respond with expected values. An example of an faulty message would be receiving a red color message when the client is already green. The algorithm takes this in consideration and will instead of changing the client to a red color, respond with the clients actual green color. Thus, we support a client that either rebooted and believes that it has the red color, or a message that was received in the wrong order, or a message that was corrupted.
However, there are two different cases that we consider when receiving a red color message from a client that has previously stated that it is green. If that client does not have a tag, we can directly know that this is either a very old message or perhaps more likely, that the client crashed and does not remember its old states. Either way, the server will enter the recovery mode. However, since the server always responds with its current saved state of the client, the client will follow these states and can thus receive its old tag back. Because of this technique the client can successfully reenter the system in the case of a reboot.
Dealing with failures is done by either mitigating the failure or entering the recovery mode. This way, there are never any significant combinations to consider since mitigation will not be noticed by the clients and recovery is handled the same in any case.
With the same method of failure detection as mentioned in the last paragraph, the server and the client can detect other inconsistencies. An example of such an error is a violation of the ℓ-exclusion value. This is what we regard as a physical error, i.e., a vehicle that was not granted access enters the critical section. While we support the possibility of other clients detecting such an error about other vehicles, that aspect of detecting is not part of our system. However, we allow clients that are part of the system to check their own position for violation. So if a vehicle in the system has not been granted access and has a position violation, it will then enter recovery mode, in extension, so will the server and all the other clients as well.

4.2.3 Recovery Mode

Each process in the algorithm holds one of three statuses (normal, detecting or recovery). While normal, everything in the system proceeds as intended. However, if a fault is discovered, the client or server discovering the issue will change its own status to detecting. Once the server is notified, it will in turn let all other clients know of the status change, making all other clients also change their own statuses to detecting.
To transition back into the normal status, the clients and server have to collectively agree to moving back. This is done in several steps and each uses what we call the all bit which is a Boolean. This Boolean describes whether or not that client or server has seen that all other processes agree with the same status that it itself holds. For example, when all processes have set their status to detecting the server will set its all bit to True. Once a client see that this all bit is set to true, it will in

turn set its own all bit to True. Once the server sees that every process including itself has set its all bit to True, it will change its status to recovery and set its all bit to False again, starting over the verification step with the all bit but this time from having the status recovery before changing the status back to normal.

The clients always try to follow the server unless they are more than one step away. Being more than one step away from the server is defined by the way the recovery procedure steps forward using statuses and the all bit. If we consider the statuses to each yield two points multiplied with its level (normal=0, detecting=1 and recovery=2), and by having the all bit set to True giving another point, we can define a function where if a process is more than one point away from any other, the system is faulty and everyone has to go back to set their statuses to detecting with their all bits as false. There is an exception where modulo has to be used so that the highest possible number (5 in our case) is only one step away from 0.

4.2.4 Decreasing Recovery Transition Time

To accommodate a faster transition from recovery mode back to normal operations, we make use of a fourth color, namely white. The reasoning behind this is that if a new vehicle entered while the system was in the recovery step, the entire procedure would reset since the new client would be more than one step away from the rest as described in section 4.2.3.

By adding a fourth color that vehicles assume when entering the system, they can ignore the statuses until approved to become red by the server. This means that the server can enter them into the system and give them a tag appropriate to when they arrived but keep them out of the recovery procedure. This would mean that the process of going back to normal would not be interrupted every time a new vehicle approached the intersection.

Instead of starting out as a red vehicle, the newly connected client assumes itself to be white when requesting initial access to the intersection. If the server has its status to normal, the client will receive a tag and be approved to become red after which everything works as described before.

If however, the server was in the middle of a recovery procedure, it would still give the client a tag but to remain white which would have the client ignore the recovery procedure. In the same sense, the server will ignore any white vehicles when checking to see if everyone agrees to continuing the recovery procedure. Once the server goes back to normal status, the white clients are allowed to change to red. In Figure 4.2 the white color can be seen in the FSM.

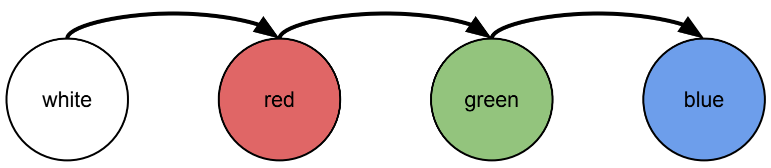


Figure 4.2: An overview of the vehicles' states and the order of their state transitions with the added white color.

4.2.5 Registration Zone

Continuous message omission on its own introduce the problem of vehicles possibly never contacting the server. If a vehicle never contacts the server it can never cross the intersection. A similar problem can occur if message omission is not 100%. If two vehicles approach the intersection from the same direction and the second vehicle contacts the server first the system would end up in a dead lock. This is because the second vehicle would receive a lower queue tag, and thus become green first. Since the green vehicle could never enter the intersection and eventually leave because of the red vehicle in front the system will not move.

By adding another zone outside of the Join Zone, we can allow message omissions. This is due to the fact that vehicles register what time they entered the Registration Zone and tries to communicate this to the server. The server will know in what order they entered for each lane. This also means that vehicles are ordered into the green set depending on their arrival time instead of the tag they got from the server. The amount of allowed omissions depends on the speed of vehicles, amount of messages sent each second and the size of the zone. The risk of a vehicle not having a single message arrive at the server before they transition from the Registration Zone to the Join Zone has to be negligible.

The calculation for this would be to calculate the amount of messages a vehicle can send between the Registration Zone and the Join Zone. This would be done by first dividing the size of the zone with the speed of a vehicle to get the time a vehicle stays in the Registration Zone. This value can then be multiplied with the amount of messages sent each second to find how many messages a vehicle sends while it is in the Registration Zone. The expected omission rate by the power of the amount of messages a vehicle has time to send gives us the risk of a vehicle having no messages received by the server before the transition between the Registration Zone and the Join Zone.

However, note that the mechanism of the registration zone does not account for large positioning system inaccuracies, which are not considered for vehicles that are traveling in the same lane, such errors could still cause deadlocks.

4.2.6 Optimizing Throughput

To show the modular property of our algorithm, we also added a substitute for assigning vehicles into the green set. This was meant to show the potential in exchanging certain parts of the algorithm to suit a specific algorithm better, although the base algorithm would suit the general case better.

Our optimization includes additions such as keeping track of where vehicles were heading to calculate which vehicles could stay inside the Critical Zone at the same time without crossing each others paths. Thus allowing multiple vehicles to be green at the same time. This greatly increased the throughput of our algorithm for the evaluation we performed.

5
Evaluation

In this chapter we present the findings of our evaluations. Some of the most notable findings are the number of trade-offs that the protocol establishes. The most notable finding is the stop margin impact which when making vehicles stop further from the intersection reduces the negative impact of positioning system inaccuracies. Another finding is that a higher message transmission frequency acts as an effective remedy against severe packet omission. Also, when experiencing failures raising the frequency of message transmissions improves the recovery time of the system. However, having a frequency that is too high causes network congestion that severely affects the performance of the system.

We define the throughput as the average time it takes for all vehicles to get through the system. An internal timer starts measuring the time when the first vehicle contacts the server and stops when the last vehicle changes its color variable to blue. At this point, the total amount of time is divided by the amount of vehicles; thus, generating a comparable metric to any number of vehicles. In other words, the throughput states how many vehicles cross the intersection per second.

A random deterministic function decides which route a vehicle should take, and where it should start from. In Appendix A, a table is included which shows the default route that was used during the evaluation.

In this chapter the results from the simulations are presented and explained. The system configuration and evaluation environment can be seen in Section 5.1. The various concepts and methodologies of the simulations will be explained in that section. This includes the simulation methodology for the injected failures, margins and throughput.

The end of this chapter contains the recovery time simulations for the failure recovery system, presented in Section 4.2.3. Here, we focus on measuring the total recovery time instead of dividing by the amount of cars that was done for the other simulations.

In Table 5.1 the default settings for the system can be seen. If nothing else is mentioned for a specific simulation, these are the settings it has been run with. In this table we can also see the vehicle spawn rate, which is the rate at which the simulator creates a new vehicle that is heading for the intersection.

Table 5.1: Default System Settings

Setting	**Value**
Transmission Frequency	2 Hz
Message Delay	0 ms
Packet Omission	0 %
Amount of Vehicles	20
Intersection Stop Margin	8 meters
Vehicle Speed	30 km/h
Join Zone Radius	200 meters
Connect Zone Radius	600 meters
ℓ-exclusion value	3
Vehicle Spawn Rate	1 Hz
Vehicle list	Appendix A

5.1 Fault Injection

The system evaluation is mainly focused on measuring the resulting throughput of different settings. We have defined throughput as how many cars that cross the intersection per second. To make a fair comparison between different cases we set a point in the simulation when a timer starts counting as explained earlier. This gives a fair metric that can be compared with any number of vehicles.

5.1.1 Fault Injection of the Message Frequency Simulations

Different message frequencies can push systems in several different directions. On one hand, having a a higher message frequency would make a system able to handle more packet omissions. However, increasing the message frequency too much would cause more omission due to the fact that it would create congestion.
To make things easier when simulating, the message frequency is done by changing the frequency of which a client would go through its main loop which is where it sends its next message. It makes sense to do this since a message would only ever be possibly updated when the client had run through the main loop. Normally, this may not be the best practice but the extra code the clients have to run is negligible considering the small amount of work the clients have to do in the current state.
It is therefore important to find a suitable frequency of which messages should be sent out. This would be affected by areas such as the intersection layout, the amount of expected vehicles and the chosen communication channel between clients and server.

5.1.2 Measuring the Impact of a Faulty Positioning System

In the developed algorithm and corresponding system the positioning system plays a big role. Whenever the client iterates through its main computation loop it compares its current position to a given x and y value. If the position is within a certain interval the client determines that it indeed is within the Join Zone, for example.

During the positioning system simulation scenarios, each of these computation loops has injected random faulty values that are individually added to both the x and y values. In the given simulations, the value is generated randomly from 0 to n where n is the chosen maximum for the given simulation.
In addition to the generated faults, different stop margins are simulated. In Figure 3.3, the lines that are perpendicular to a given lane closest to the intersection represent the stop margins. In SUMO these lines represent traffic lights, but for our use case they are simply acting as stop lines. To simulate different margins the stop lines are simply moved further back at certain intervals.
The purpose of the stop margins were to minimize the possibility that a vehicle gets an erroneous reading that would indicate that it was within the critical or the capture zone as these would trigger the failure detection and in turn, initiate the recovery procedure.

5.1.3 Packet Delay and Packet Omission Simulations

Simulating packet delay was done by having a simple static value that was added to the simulators scheduling function. Normally, when the delay is zero the scheduleAt function, which is the function that schedules message transmissions, simply scheduled an outgoing packet transmission to the current time. When running the delay simulations the transmission would essentially be postponed by x milliseconds, where x corresponds to the selected value for that simulation.
When running the simulations for the packet omission, a received packet had a certain % risk to not be considered at all, i.e., the client or server would not be aware of that packet at all.

5.1.4 Fault Injection of the Recovery Procedure

To produce the recovery procedure simulations the system was initially disabled from allowing any vehicles to cross the intersection. When all vehicles are lined up at the intersection an error is injected. This error triggers the failure detection and makes a vehicle enter the recovery mode.
Also take note that in the simulation cases for the recovery procedure we chose to instead focus on the total time taken for the recovery instead of using the throughput. This way of measuring gave a more relevant metric for comparing the performance of the recovery procedure. Take note that this means it does not measure the total time of the simulation, only how long it takes from the first detected error to the complete stabilization of the system.

5.1.5 Illustrating the Results

To present the results of the evaluation, three different types of plots are used. Firstly, the violin plot type is used, this violin plot shows the full distribution of each recorded result as well as the maximum, mean and minimum values of a given test case. It is the first choice whenever possible, but in some cases the individual violins become too small to give any meaningful illustration, this is when we swap to the box plot. The box plot is simply a standard box plot and shows slightly less information than the violin plot. Lastly, we have the contour plot, which in each use case acts as a plane section representation of a three dimensional graph. The height of this graph represents the throughput or the recovery time, depending on the test, and each area marked by a contour represent different heights that is the plane section. This means that this plot is used to depict how two different metrics change the resulting output, and is the easiest way to find trade-offs.

5.2 Message Frequency

In this section a single figure is listed. This is to take the time both to remind the reader shortly about network congestion, but also to show that a high message frequency might not be necessary in our system. In Figure 5.1 we show how the throughput is affected by message frequency. To the left we can see the default value of 2 Hz. When the system is not experiencing any failures we can see no noticeable change in throughput until we reach a point at 165 Hz where the system starts to decline rapidly. Already at the point of 200 Hz the throughput has declined to such a degree that it has reached 25% of the initial throughput.

Finally, we can say that in a system with 20 vehicles, it is not recommended to go above 160 Hz, and is not necessary to go above 2 Hz when no failures are expected, and realistically, it might be better to find some middle ground between these extremes.

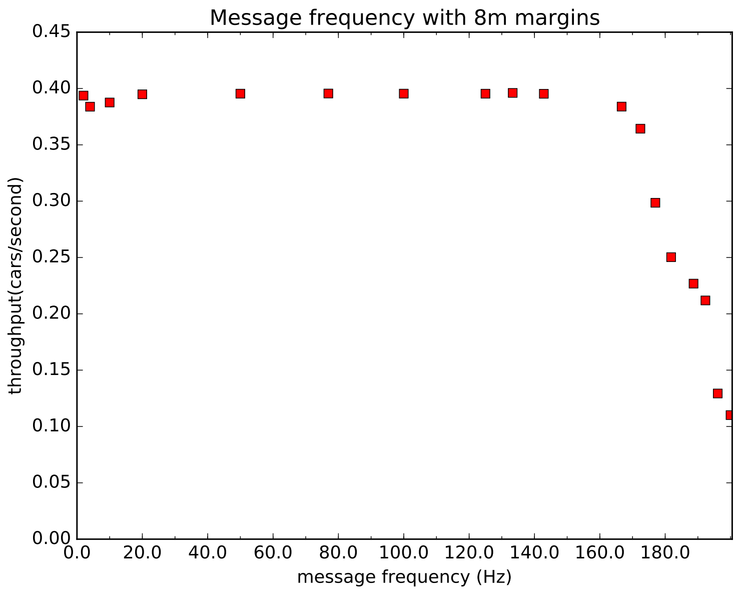


Figure 5.1: A box plot showing how higher frequency causes message congestion which affects the intersection throughput negatively.

5.3 Positioning Errors

In this section the results of the simulations that were carried out with injected failures in the positioning system will be presented. The simulations measure how throughput is affected by different positioning system errors and how margins can remedy these errors. In Figure 5.2 we can see how throughput is not affected negatively in the lower margins. At 20 meters the throughput has declined with about 10%. The throughput is steadily decreasing and is losing about 10% every 20 meters of added margin.
We can conclude from this initial simulation that if the accuracy of the used positioning system is expected to be high for a given system a margin below 10 meters should be used as higher margins will affect the throughput noticeable negatively.

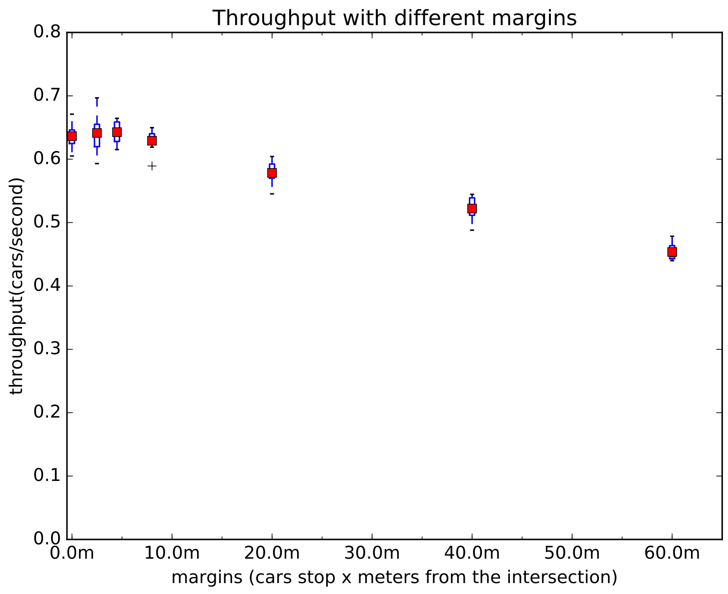


Figure 5.2: A box plot showing how higher intersection stop margins affect the throughput of cars through the intersection in the case where the cars have paths that intersect more often.

Studying figures 5.3 to 5.6 we can clearly see how severely the positioning failures can affect the throughput. We can also see how well the stop margins act to counter the faulty position readings. We remember how the previous readings from Figure 5.2 show that a too high margin will cause a lowered throughput. In this trade-off the sweet spot is definitely to have a system that is always accurate below 10 meters with a 10 meter stop margin. This is to not lose any throughput while maintaining the resistance to positioning system failures.
These erroneous readings of position makes the vehicle incorrectly report its position to be within the critical zone or the capture zone. This causes the system to enter the recovery mode. While being a correct reaction of the system to act safely it has a severe effect on the throughput. This is why it is important to use margins effectively nullify any of these issues by only letting vehicles which have been granted

access to the intersection ever be within the expected error-radius. This attribute can be seen by comparing each of the figures, where for instance Figure 5.4 shows that only when the inaccuracy is greater than 3 meters the system throughput is affected.

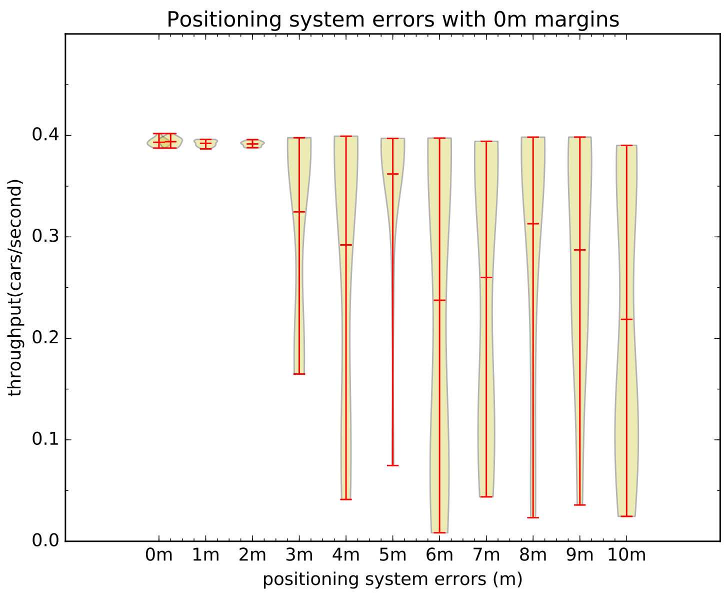


Figure 5.3: A violin plot showing how higher amounts of errors affect the throughput of cars through the intersection and how injected positioning faults affects throughput.

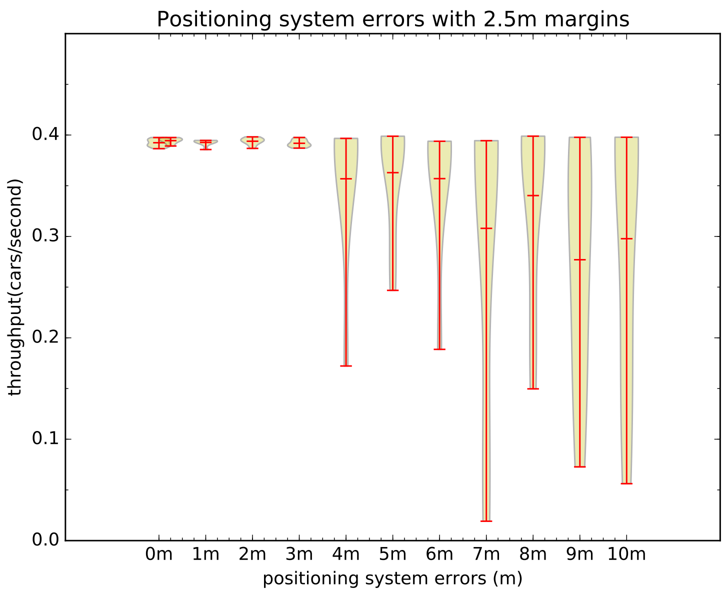


Figure 5.4: A violin plot showing how higher amounts of errors affect the throughput of cars through the intersection and how injected positioning faults affects throughput.

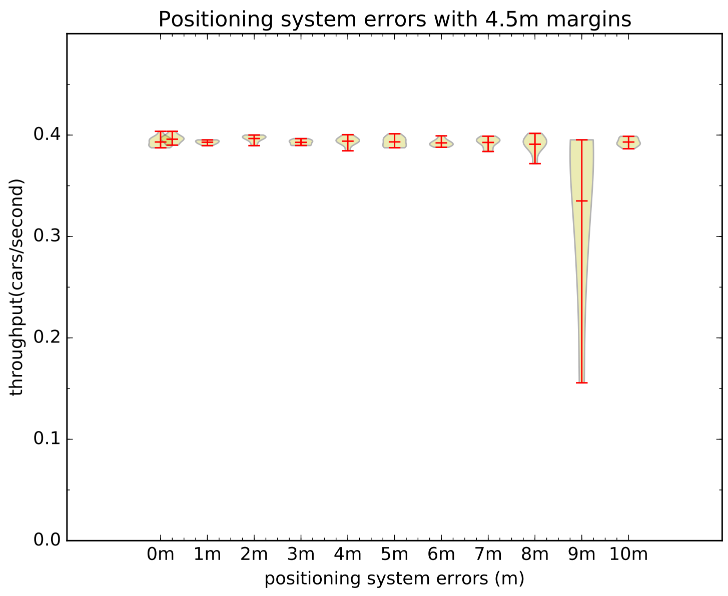


Figure 5.5: A violin plot showing how higher amounts of errors affect the throughput of cars through the intersection and how injected positioning faults affects throughput.

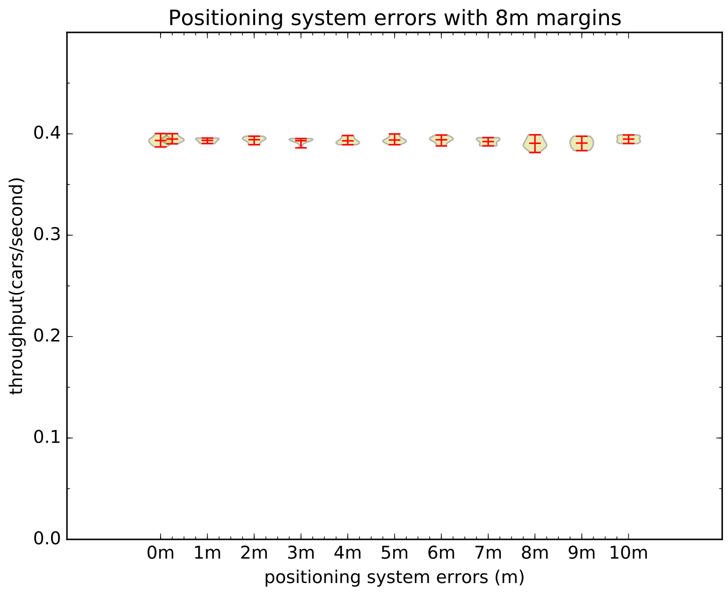


Figure 5.6: A violin plot showing higher amounts of errors affect the throughput of cars through the intersection and how injected positioning faults affects throughput.

Finally, in figures 5.7 and 5.8 we can see two contour plots to summarize the previous four plots. Here we can more clearly see how higher margins nullify the issues from the faulty positioning system. However, at 9 meters range of inaccuracy in the positioning system we notice a large impact on the throughput. This is both due to the simulation being run at 4.5 and 8 meter margins and the big impact of the 9 meter positioning system error simulation. A larger sample size might yield more even results on the 4.5 meter simulation, since as we can see in Figure 5.5, the 9 meter simulation case only encountered failure in a few cases.

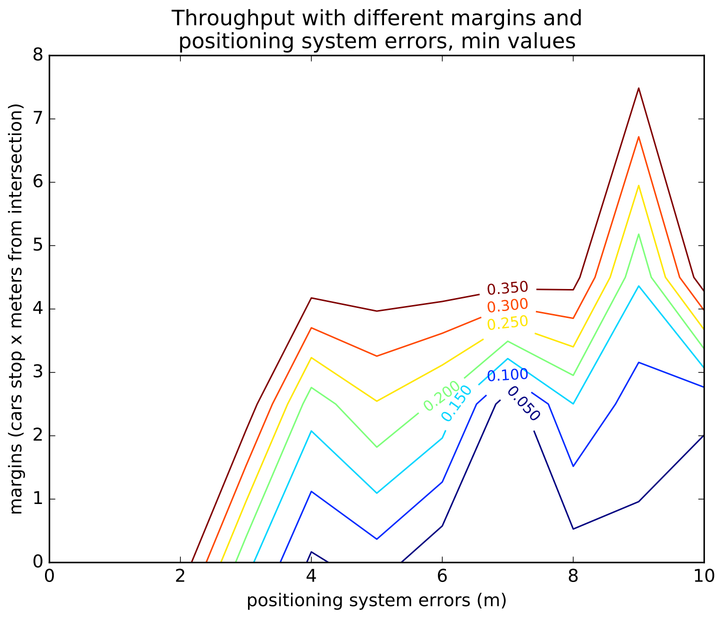


Figure 5.7: A contour plot showing how higher amounts of errors affect the throughput of cars through the intersection and how injected positioning faults affects throughput. This graph is a plot of the min values.

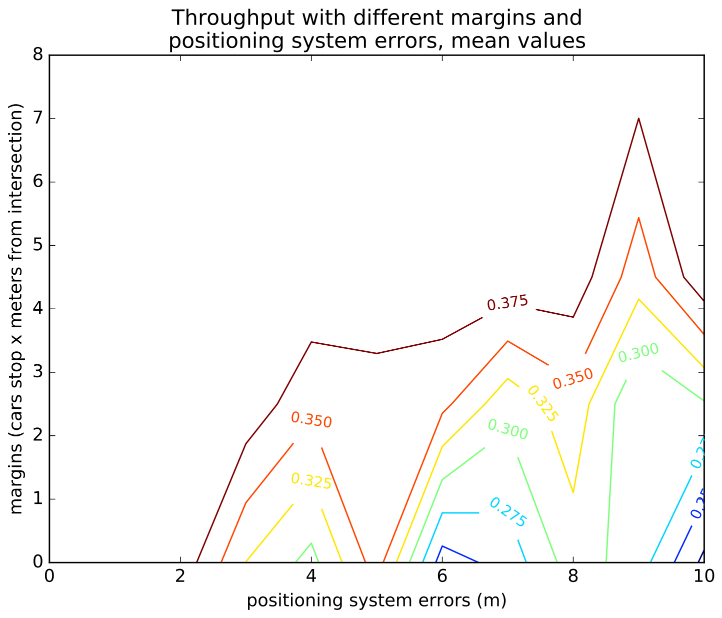


Figure 5.8: A contour plot showing how higher amounts of errors affect the throughput of cars through the intersection and how injected positioning faults affects throughput. This graph is a plot of the mean values.

5.4 Packet Omission

In this section the simulations that were run with injected packet omissions are listed and explained. Again, the methodology of these injections are explained in Section 5.1. But in short, whenever a packet or message is received the recipient has a chance to disregard the contents of that message, i.e., dropping it.

In Figure 5.9 we can see how the throughput naturally declines when the system is subjected to omissions. The system can handle up to around 30% omission without the throughput being affected at all. Then the throughput declines approximately linearly until 90%. After 90% the problems became so severe that the simulations did not provide any results as the simulations took too long to finish due to the vehicles not managing to cross the intersection.

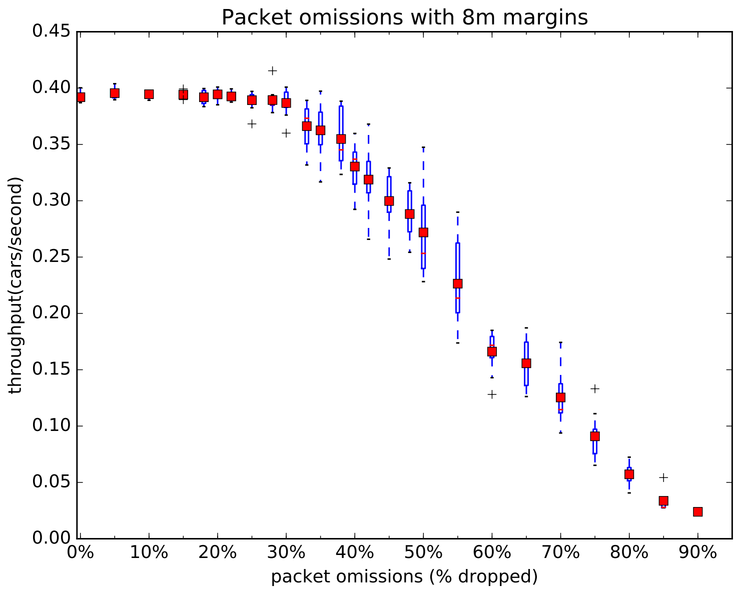


Figure 5.9: A box plot showing how higher amounts of errors affect the throughput of cars through the intersection and how message receive omission affects throughput.

In figures 5.10 and 5.11, we can see how a higher message frequency counters the packet omission rate. At the highest frequency that this simulation was run at the system was not affected at all until the omission reached around 60%. We can thus conclude that to be resilient to the injected packet omission, a higher transmission rate is a viable solution.

However, it is important to consider the cause of these omissions. In a situation where the omissions might be caused by a network congestion, a higher frequency of transmission might hurt the system performance more by adding additional congestion. While in other cases just raising the frequency to 4 or 5 Hz does not have a huge impact on the total amount of packets that the system will handle, while still not suffering at a 50% omission rate.

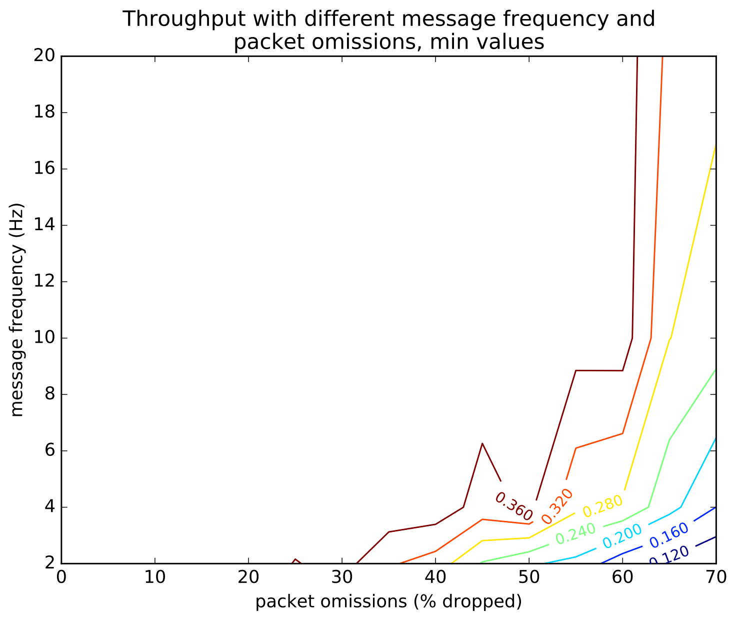


Figure 5.10: A contour plot showing how different packet transmission frequencies and injected packet receive omissions affect the throughput of the system.

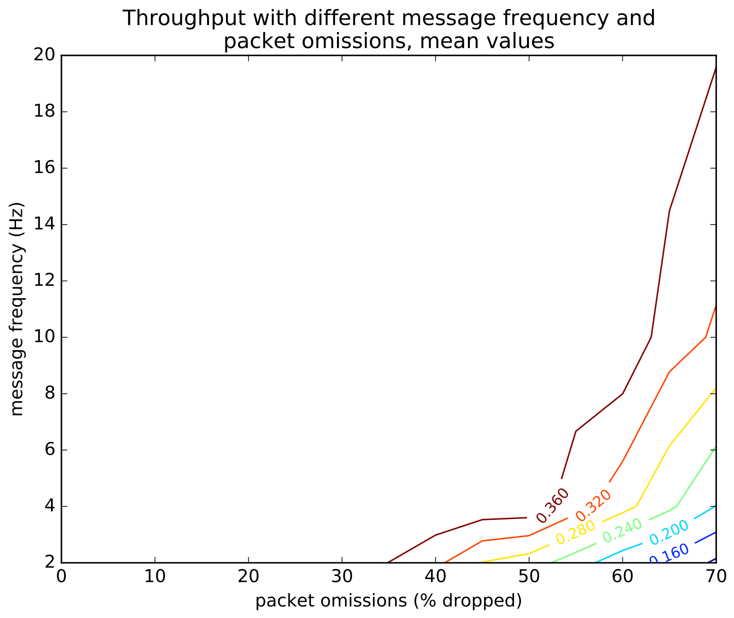


Figure 5.11: A contour plot showing how different packet transmission frequencies and injected packet receive omissions affect the throughput of the system.

The system was also simulated with margins and packet omission to check for any correlation. In figures 5.12 and 5.12 the results of these simulations can be seen. The system did not seem to be affected additionally by the margin and the packet omission. This result can especially be noticed in the graph of the mean simulation results in Figure 5.12.

Thus, there is no added drawback of having additional margin in an area where one might expect to see packet omission.

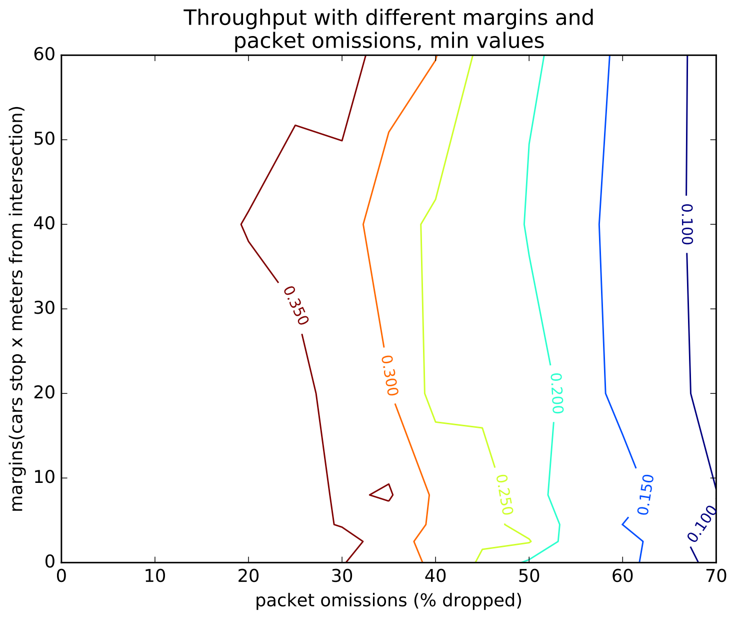


Figure 5.12: A contour plot showing how higher intersection stop margins affect the throughput of cars through the intersection and how injected packet receive omission affects throughput. This graph is a plot of the min values.

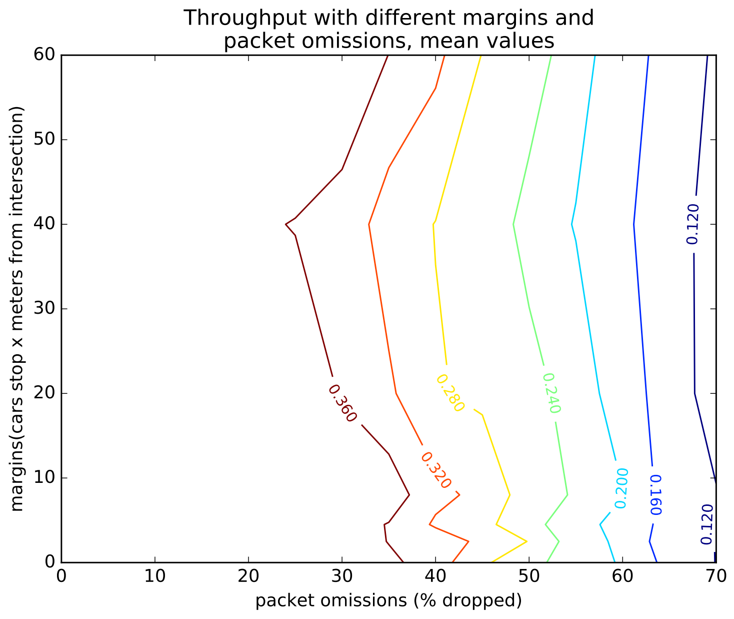


Figure 5.13: A contour plot showing how higher intersection stop margins affect the throughput of cars through the intersection and how injected packet receive omission affects throughput. This graph is a plot of the mean values.

5.5 Message Delay

This section contains the result of the simulations that measure the impact of injected message delays on the throughput of the system.

The simulation methodology and the method of injecting delay is explained more in detail in Section 5.1. The summary of the delay injection is that a static delay was added to both client and server sides before the message was transmitted. Thus, if the graph shows 1000 ms delay, the total round trip delay is actually 2000 ms. Due to the static delay that was added we see little variance in the result produced by any of the simulations in this section.

In Figure 5.14 we can see how the system throughput is affected by the message delay. The throughput starts getting affected at around 800 ms, then it starts to rapidly decline until around 2300 ms where the decline is more linear.

In conclusion it is recommended to have a delay less than 800 ms one-way or a total round trip of under 1600ms at the default message transmission rate of 2 Hz.

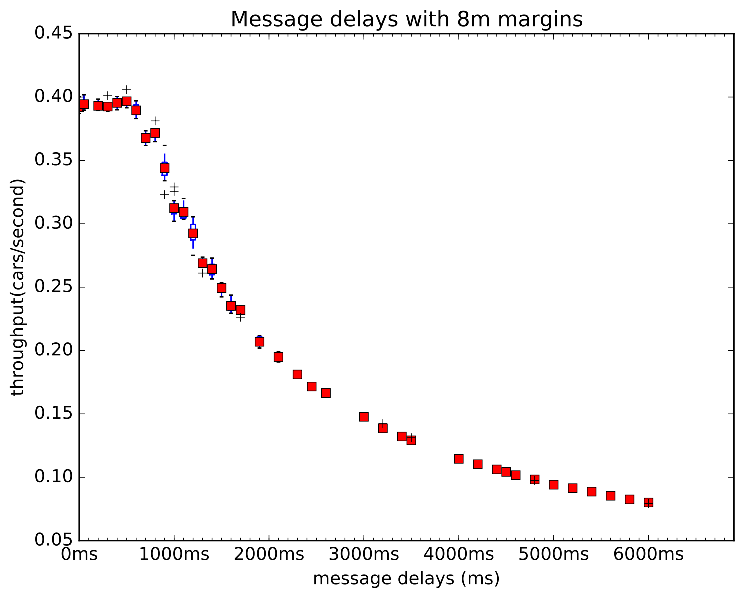


Figure 5.14: A box plot showing how higher amounts of errors affect the throughput of cars through the intersection and how message delay affects throughput.

In figures 5.15 and 5.16 we can see a pair of contour graphs. They both show how the throughput is affected by message delay and message frequency. We can see a gentle slope of increasing throughput at higher message frequencies. This means that while frequency somewhat remedies the delay, it is not by a lot.

At a 1000% increase of message frequency a 300 ms higher delay can be tolerated before the throughput starts to suffer. Increasing the message frequency by 100% yields approximately 150 ms higher tolerance of static delay which seems to be the sweet spot before the gain starts to be very meagre.

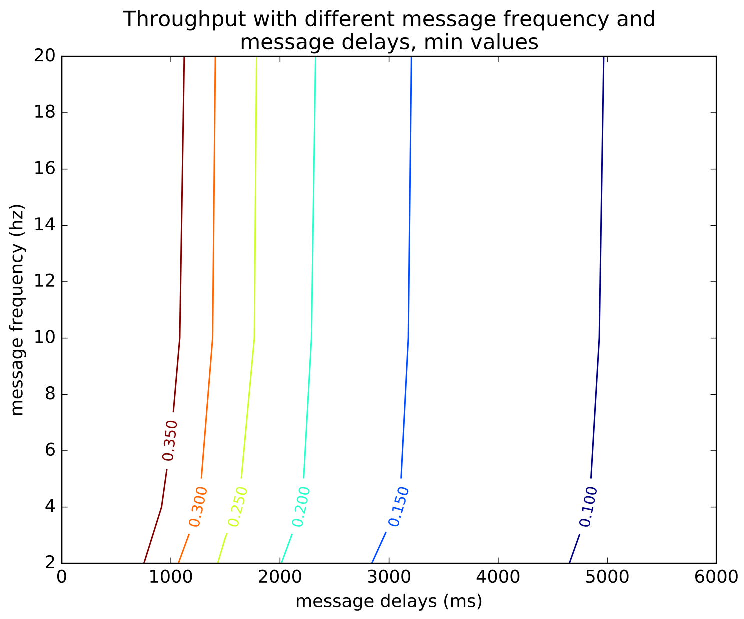


Figure 5.15: A contour plot showing how different message transmission frequencies and injected message delays affect the throughput of the system.

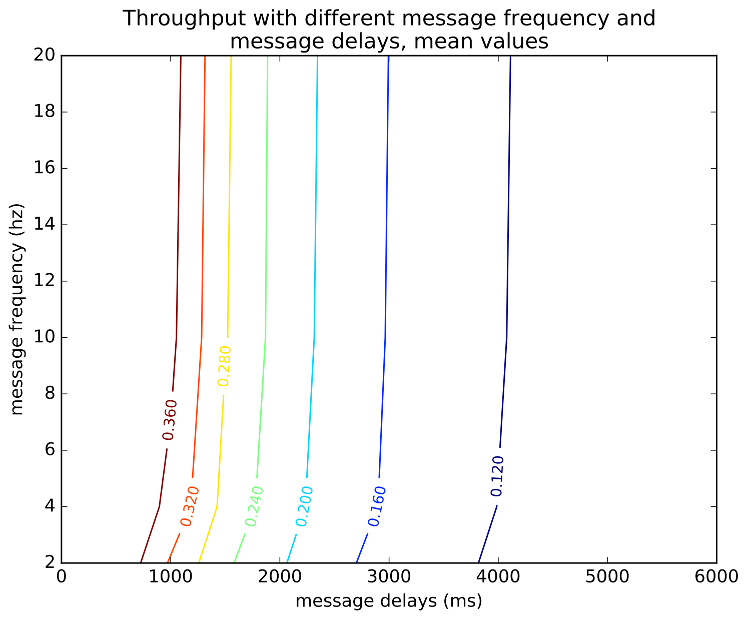


Figure 5.16: A contour plot showing how different message transmission frequencies and injected message delays affect the throughput of the system.

In figures 5.17 and 5.18 we can see contour graphs of how throughput is affected by margin and message delay. We can observe the slightly lowered throughput due to the higher margins, and the gradual decline of the throughput from the added delay. However, no correlation between the two can be seen in any of the graphs.

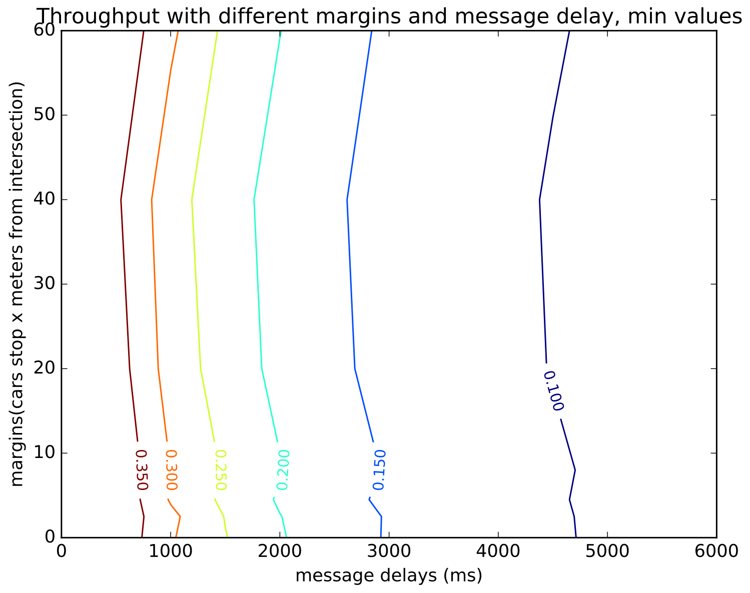


Figure 5.17: A contour plot showing how higher intersection stop margins affect the throughput of cars through the intersection and how injected message delays affects throughput. This graph is a plot of the min values.

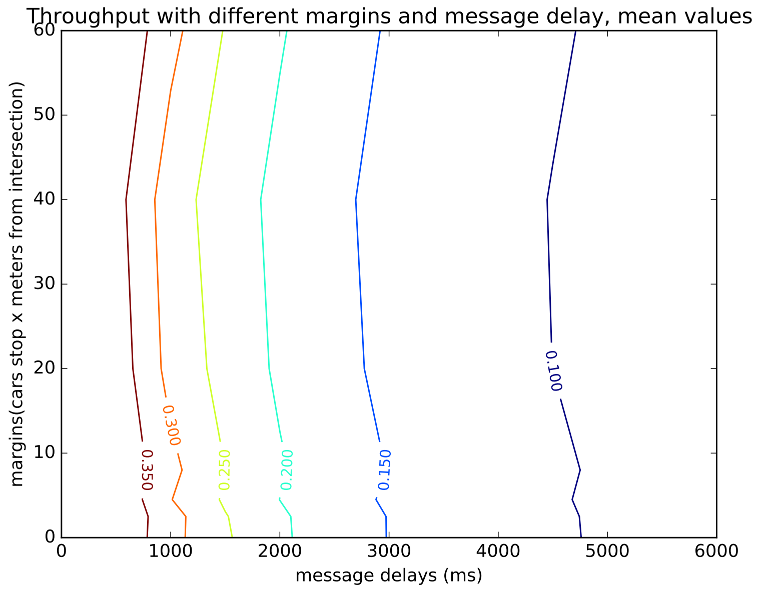


Figure 5.18: A contour plot showing how higher intersection stop margins affect the throughput of cars through the intersection and how injected message delays affects throughput. This graph is a plot of the mean values.

5.6 Recovery Simulations

In this section the results of the simulations of the recovery procedure are presented. Instead of focusing on throughput like in the other sections, this section will focus on recovery time. This means that from the moment the server detects the injected failure a timer starts, and when the complete system has reached the stable normal state the timer stops.
In Figure 5.19 we can see how the recovery time is affected by the amount of vehicles involved in the recovery procedure. For the increase from 5 to 20 vehicles we can see that without communication failures there is almost no difference, around two seconds, which is an increase of 50% in time elapsed and an increase of 300% in amount of clients. In other words, it is not free to have more vehicles in the system, but it is not very expensive either. Raising the transmission frequency might also help to reduce the time it takes for the system to recover, as we will discover later in this section. Although keep in mind, as with a higher frequency, more vehicles will at some point also cause network congestion, as more messages will be exchanged.

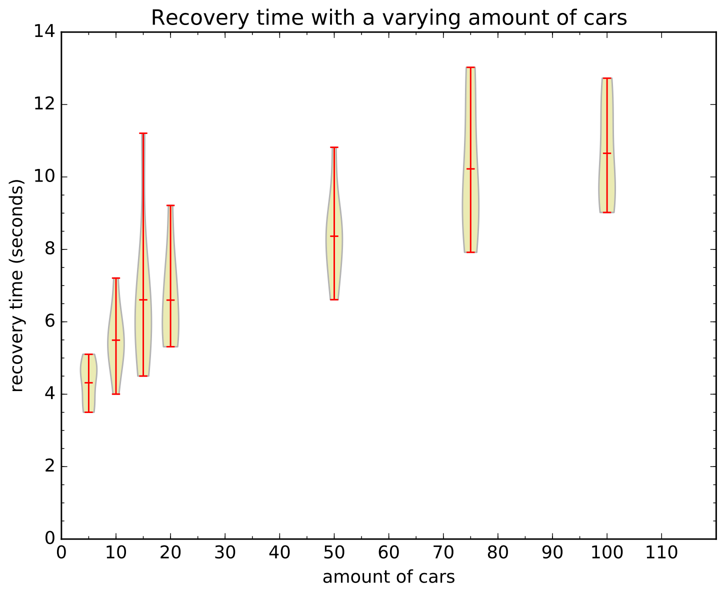


Figure 5.19: A violin plot showing how different amounts of cars affect the recovery time of cars waiting at the intersection when a fault is injected.

In figures 5.20 and 5.21 we can see how packet omission changes the recovery time of the system with different amount of vehicles. Initially, the increase of the recovery time is gradual, but as the omission passes 50% we start to see a very severe impact. From the mean time of 0% omission to 70% the time elapsed has increased by more than an entire order of magnitude, but as we have seen previously, a way to effectively combat the packet omission is by introducing a higher transmission rate (see Figure 5.10).
When looking at how more cars affect the performance of the system when experiencing packet omission we can see that even in the worst case scenario of the

simulation cases, that is, 70% packet omission, the mean only differs by around 15 seconds between 5 and 20 cars, which is an increase of roughly 20%. However, between 10, 15 and 20 cars the mean is roughly the same, this means that while the amount of cars has been increased by 100% from 10 to 20 cars, no impact can be seen on the mean recovery time in the worst case while experiencing packet omissions, while relatively minor increases can be observed in the other cases.

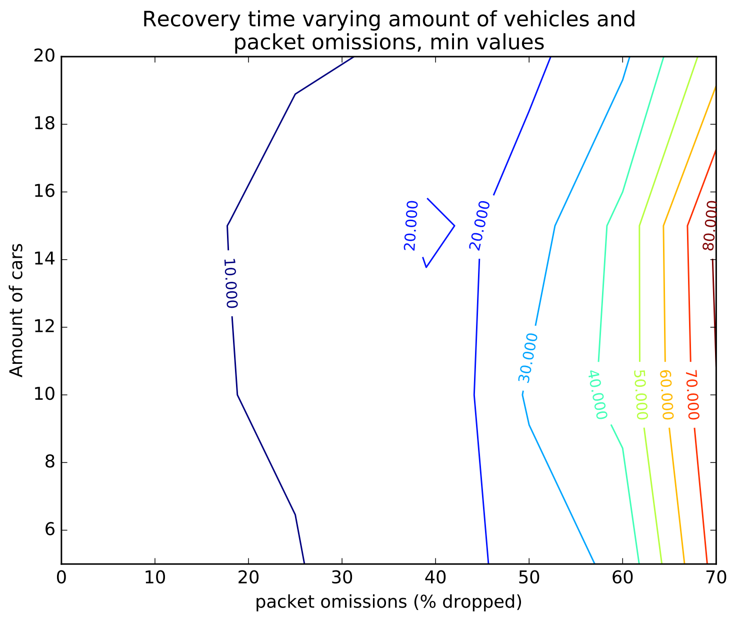


Figure 5.20: A contour plot showing how higher amounts of errors affect the recovery time of cars waiting at the intersection when a fault is injected.

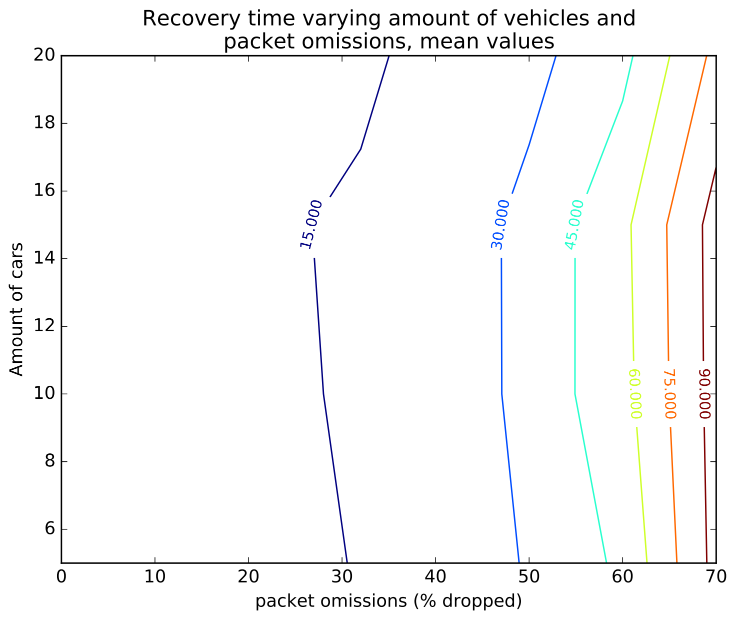


Figure 5.21: A contour plot showing how higher amounts of errors affect the recovery time of cars waiting at the intersection when a fault is injected.

In Figure 5.22 we can see the resulting recovery time from increasing delay. The delay was added statically to both client and server sides and applied to each outgoing packet, just as described in Section 5.5.

A linear increasing curve shows how the time before the system has recovered increases. We can see that the recovery time is approximately one order of magnitude greater. i.e., a one second delay results in a ten second recovery time.

In conclusion, no severe hidden issues can be observed other than the resulting natural timing delay that the injected delay causes. We can expect recovery to be quicker with a higher frequency which might be the only thing to counteract the message delay, however, only slightly, as seen in Figure 5.15.

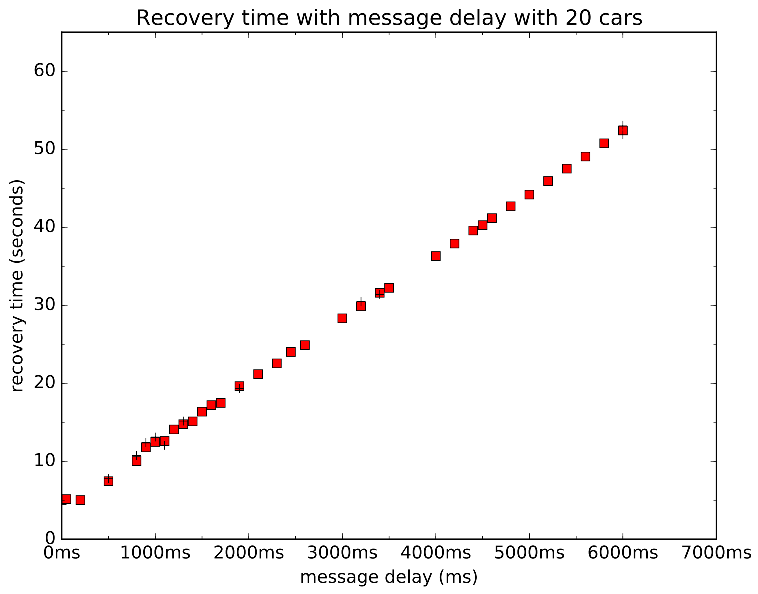


Figure 5.22: A box plot showing how higher amounts of errors affect the recovery time of cars waiting at the intersection when a fault is injected.

In Figure 5.23 we can see how the total recovery time is affected by increasing the message transmission rate. We can clearly see that it truly is worthwhile to increase the frequency in a system without failures. By increasing the rate to 15 Hz we reach the peak performance of a 0.5 second recovery time, which is an improvement of roughly 6.5 seconds when comparing the mean values. Increasing the frequency further than that does not seem to have much impact. Possibly due to the natural delay of things, such as computation time. All in all if the environment and surrounding systems supports the higher frequency it is worthwhile, especially to at least 10 Hz, to quickly recover from failures.

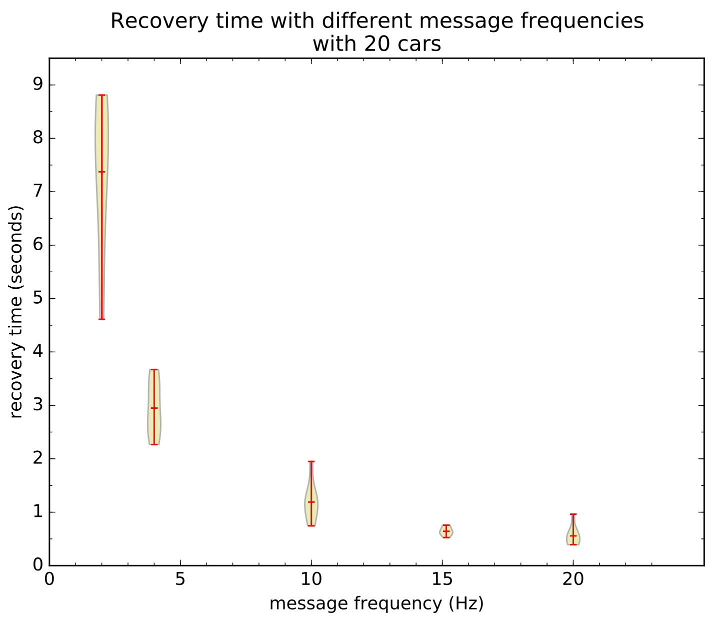


Figure 5.23: A violin plot showing how higher frequencies affect the recovery time of cars waiting at the intersection when a fault is injected.

In Figure 5.24 we can see what effect margins have on the recovery time of the system. The system and the recovery time was not affected by increasing the margin. The cause is simply that the distance of the vehicles does not in any way determine how they should behave in the recovery procedure. As such we can draw a minor conclusion that the margins does not affect the recovery procedure, neither negatively nor positively.

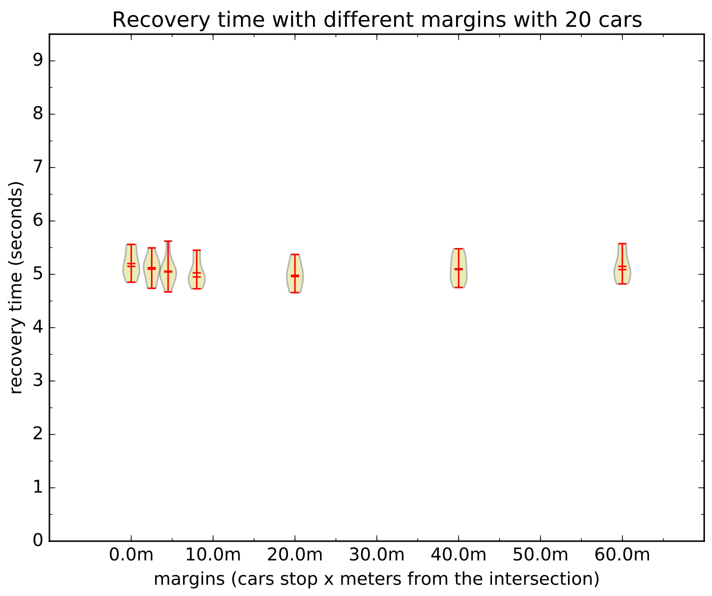


Figure 5.24: A violin plot showing how higher stop margins affect the recovery time of cars waiting at the intersection when a fault is injected.

5.7 Trade-Offs

In our evaluation we found a number of trade-offs for some metrics, as well as some that had no trade-offs, in this section we will simply give a summarized conclusion of these findings.

5.7.1 Positioning Error

Adding the stop margin allows vehicles to not accidentally trigger the failure detection when suffering from positioning system inaccuracy. The stop margin essentially make the vehicles have an estimated positioning radius that is smaller than the added margin from the intersection. This margin eliminates the drop in throughput.
We have not evaluated the effect message frequency has on positioning errors due to how we implemented the experimentation part of it. By increasing message frequency, our implementation would become much more prone to finding it was erroneous in its position, making the evaluation of it much more difficult.

5.7.2 Omission

A higher transmission frequency causes more packets to be sent, as such, when suffering packet omission, with more messages being transmitted, there is a higher chance that one of these packets arrives at the recipient. Thus, there is a trade-off that says that a higher frequency remedies the packet omission. However, take note that in a real environment the packet omission might be caused by network congestion, increasing the amount packets being transmitted might make the problem more severe than before.
There is no trade-off between packet omissions and stop margin, neither does simply not depend on the other.

5.7.3 Delay

There is a slight trade-off between transmission frequency and message delay, this means that when the frequency is higher, the impact of message delay is smaller. However, this impact is quite small, and raising the frequency from 2 Hz to 20 Hz is equivalent to having 300 ms less delay.
Just as the case for packet omission there is no trade-off between the stop margin and message delay.

5.7.4 Recovery

Recovery time is affected by having more vehicles, however, the impact is quite small. There is roughly a time increase from five to ten seconds when increasing the amount of vehicles from 10 to 100.
When experiencing packet omission recovery time takes a quite significant amount longer, especially when reaching a high amount of omission. It could be said that omission translates into delay, 50% omission translates roughly to 6000 ms delay.

Thus it can be said, while there is no real trade-off, the recovery time is quite significantly impacted by a higher percent omission.
The case of how recovery time is affected by message delay is a bit special, just as for packet omission. There is no inherent trade-off, but the system is still affected by message delay, thus, a higher message delay causes almost one entire order of magnitude per added delay in increased time to recover.
Finally, stop margins and recovery time has no trade-off. We can see that there is no impact at all on the recovery time when changing the stop margins.

6 Discussion

The finishing chapter goes into more details about potential future work and extensions as well as conclusions drawn from the project. These are extensions such as making the algorithm able to deal with server crashes and network congestion, future work that describe other possible implementations and usage of the algorithm and conclusions that go into the self-stabilizing recovery as well as throughput trade-offs that were seen in the algorithm experiments.

6.1 Summary

We started the thesis by studying the problem of creating a virtual traffic light. However, this traffic light should not only be able to arbitrate vehicles through an intersection, it should also be robust to several possible errors such as positioning errors, message delay and packet omission. This is an important building block for autonomous driving since the virtual traffic light systems benefit greatly from being able to deal with such errors. After several iterations of making the algorithm more robust to these errors, we started implementing the algorithm into a simulator for evaluation and experiments. The evaluation revealed both important trade-offs that existed such as a higher message frequency counteracting packet omission, but also trade-offs that did not appear to exist such as message delay and margins.

6.2 Comparing with Related Work

The use of a virtual traffic light is by itself not a unique project. The focus on our side have been the self-stabilizing features to make the virtual traffic light more robust to potential errors. However, it is still important to compare our algorithm on areas where we perform similar tasks as other work we have based it on.
A notable example would be comparing to Savic et al. [40], where both algorithms focus on arbitrating vehicles through an intersection. However, we support more cars in our algorithm than what is shown with the one by Savic et al. where only two vehicles are currently supported. However, the main difference is that Savic et al. uses no central coordinator, and neither does Azimi [1].
Our reasoning for using a central coordinator is that a vehicle that approaches an intersection that gets no response from any other vehicle can never be sure that there is no existing other vehicle approaching where communication has completely failed. While it might have been out of scope for the mentioned papers it is something we wanted to consider in this work. In other words, our system is capable to at least

not allow any vehicles to enter the intersection if no message has been received from the server. A vehicle should only enter the Critical Zone when explicit access has been granted. However, as is mentioned in [1] this introduces a single point of failure, in Section 6.5 we reveal a solution that could remedy this weakness. Another example of implementation with a decentralized structure can be seen in the paper by Hagenauer et al. [26].
There are also areas where we have not developed as thoroughly as the work we based it upon. An example would be the use of the Capture Zone [25], where the work it is based on explains in much further detail how to develop a capture zone while our algorithm only shows it can be used and works well with the algorithm.

6.3 Conclusions

In this thesis we developed a fault tolerant algorithm for crossing intersections. One part of the work acts as a membership service, which grant clients access to a resource. The other part of the work is a self-stabilizing recovery system which first detects certain failures, shuts down normal operation, notifies the rest of the system, and finally recovers once the issue is resolved. While work remains to tolerate more faults such as client and server crashes, many faults are currently dealt with. This includes faults such as packet omissions, delays, positioning errors and more.
The way we deal with these come with trade-offs. Dealing with packet omission can be countered by increasing the message frequency. However, increasing the message frequency can incur congestion in the network, which itself would cause omission. Message delay does not seem to have a trade-off, other than slightly with message frequency considering the update to the vehicle will have been sent out earlier. This will only decrease the delay by milliseconds though. What we did notice however, is that message delay did not appear to be affected further by other potential problems such as omission, big margins and positioning errors. That brings us to the last major fault we experimented with, which is positioning errors. Positioning errors can cause huge issues with the system, but they seem to be nullified with an appropriate margin being used. By adding a margin greater than the potential position error, the issue it would cause appear to be countered. Using a greater margin does not appear to decrease throughput by much as well, which means that cost of countering position errors is not high.
The recovery procedure is an important aspect of our algorithm. It works in a self-stabilizing manner where once an error is found, the coordinator will enter what we call detecting mode where it has to get a verification from all clients that it has entered the detecting mode. Once this is done, the coordinator asks for verification that every client is aware that every other client has verified detection. Once this is done it enters recovery mode and does the same thing with the clients again after which it enters normal mode and continues with normal operations. This is to ensure that when there is an ongoing recovery, no vehicles should attempt to cross the intersection apart from those already cleared until the issue has been resolved.
What we offer then is an asynchronous state machine which means that TCP is not necessary. While we do not require the arbitration to be FCFS, we decided to show that implementation as it makes it easier to grasp the overall functionality of the

algorithm. The idea is that many of these, such as the prioritization system at the coordinator are built as modules and can be swapped for something else depending on the need.

6.4 Proposed Future Work

While this work was developed to support vehicles in an intersection, it is certainly not limited to this usage. As a result of the flexible membership service that was developed, i.e., the queue tag system, any domain that could benefit from a similar system could use this one, where good connectivity is not always guaranteed. Consider an area in which connectivity is fleeting and message delay is long so clients and servers can not be sure if they have lost connection or if the last message is in transit between server and client. This would would mean that at certain times, many vehicles will not have heard from the server in a long time and this is when our algorithm could be useful since it guarantees that once a client has been granted access, it will not be taken back and the vehicle in this occurrence can continue towards the intersection even after having lost its connection to the server.
In its essence, the algorithm is a resource sharing algorithm and can thus be used as one. This would mean that any area that can clearly define a resource to be shared between clients, where the self-healing property our algorithm utilizes can be used, would likely be able to make an implementation based on what we have proposed. An example would be sharing access to some virtual space, specifically where connectivity to a coordinator is not always guaranteed. The fact that our algorithm can deal with failures and self-stabilize if there are issues makes it even more useful in such a scenario.

6.5 Extensions

By extending the system with virtual synchrony [6, 5], the algorithm would also be able to handle server crashes. This means that we can remove one of the weak points mentioned in [1], which is the single point of failure. This resistance can be achieved since virtual synchrony in short allows multiple servers to act as redundant systems to a single server, since they all emulate being one entity. The variable that would need to be shared between servers is the queue which contains information about all the clients with their respective tags, colors and statuses.
Another extension could be congestion control for the clients. By also adding a way for making clients aware of each other, such as receiving a list from the server of how many clients that are currently connected, they would be able to change the rate at which they send messages depending on the amount of currently connected vehicles. The specific values they could change between would have to be tested to see where the trade-offs are. This would ensure better congestion control while limiting potential packet omission from other sources.
Furthermore, by extending the algorithm with only V2V communication such as the one described by Ferreira et al. [24], one could eliminate the need for servers. This would enable to algorithm to work better with a only localized communication

as well as not having to keep a server or roadside unit running. We chose not to investigate this solution further as we decided the main concept of the algorithm would work well without such a solution and there were other areas of the algorithm that were in greater need of being reworked. However, the only shared information is, as explained earlier, the queue which contains information about clients. Once this would be set up, the clients could, with some modifications to the algorithm concerning how it is decided which vehicle would gain access next, work as it does in its current state without the use of a server. We also note the existence of techniques for reducing the occurrence of packet loss [28, 34] that can help in improving the performance of the proposed algorithm.

Bibliography

[1] R. Azimi, G. Bhatia, R. R. Rajkumar, and P. Mudalige. Stip: Spatio-temporal intersection protocols for autonomous vehicles. In *2014 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, pages 1–12, April 2014.

[2] Christian Berger, Michel Chaudron, Rogardt Heldal, Olaf Landsiedel, and Elad M Schiller. Model-based, composable simulation for the development of autonomous miniature vehicles. In *Proceedings of the Symposium on Theory of Modeling & Simulation-DEVS Integrative M&S Symposium*, page 17. Society for Computer Simulation International, 2013.

[3] Christian Berger, Erik Dahlgren, Johan Grunden, Daniel Gunnarsson, Nadia Holtryd, Anmar Khazal, Mohamed Mustafa, Marina Papatriantafilou, Elad M Schiller, Christoph Steup, et al. Bridging physical and digital traffic system simulations with the gulliver test-bed. In *International Workshop on Communication Technologies for Vehicles*, pages 169–184. Springer Berlin Heidelberg, 2013.

[4] Christian Berger, Oscar Morales Ponce, Thomas Petig, and Elad Michael Schiller. Driving with confidence: Local dynamic maps that provide los for the gulliver test-bed. In Andrea Bondavalli, Andrea Ceccarelli, and Frank Ortmeier, editors, *Computer Safety, Reliability, and Security - SAFECOMP 2014 Workshops: ASCoMS, DECSoS, DEVVARTS, ISSE, ReSA4CI, SASSUR. Florence, Italy, September 8-9, 2014. Proceedings*, volume 8696 of *Lecture Notes in Computer Science*, pages 36–45. Springer, 2014.

[5] Birman. *Vsync: Consistent Data Replication for Cloud Computing*, 2017 (accessed January, 2017).

[6] K. P. Birman. A review of experiences with reliable multicast. In *1999 Software Practice and Experience*, pages 741–774, April 1999.

[7] Olga Brukman, Shlomi Dolev, Yinnon Haviv, Limor Lahiani, Ronen Kat, Elad Michael Schiller, Nir Tzachar, and Reuven Yagel. Self-stabilization from theory to practice. *Bulletin of the EATCS*, 94:130–150, 2008.

[8] António Casimiro, Jorg Kaiser, Elad M Schiller, Pedro Costa, José Parizi, Rolf Johansson, and Renato Librino. The karyon project: Predictable and safe coordination in cooperative vehicular systems. In *Dependable Systems and Networks Workshop (DSN-W), 2013 43rd Annual IEEE/IFIP Conference on*, pages 1–12. IEEE, 2013.

[9] Antonio Casimiro, Oscar Morales Ponce, Thomas Petig, and Elad Michael Schiller. Vehicular coordination via a safety kernel in the gulliver test-bed. In *34th International Conference on Distributed Computing Systems Workshops*

(ICDCS 2014 Workshops), Madrid, Spain, June 30 - July 3, 2014, pages 167–176. IEEE, 2014.

[10] Antonio Casimiro, José Rufino, Ricardo C. Pinto, Eric Vial, Elad Michael Schiller, Oscar Morales Ponce, and Thomas Petig. A kernel-based architecture for safe cooperative vehicular functions. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014, Pisa, Italy, June 18-20, 2014*, pages 228–237. IEEE, 2014.

[11] A. Colombo and H. Wymeersch. Cooperative intersection collision avoidance in a constrained communication environment. In *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pages 375–380, Sept 2015.

[12] Shlomi Dolev. *Self-stabilization*. MIT press, 2000.

[13] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad M Schiller. Self-stabilizing virtual synchrony. In *SSS 2015 and DISC 2015*. Springer-Verlag Berlin Heidelberg, 2015.

[14] Shlomi Dolev, Chryssis Georgiou, Ioannis Marcoullis, and Elad Michael Schiller. Self-stabilizing reconfiguration. *CoRR*, abs/1606.00195, 2016.

[15] Shlomi Dolev, Seth Gilbert, Nancy A Lynch, Elad Schiller, Alex A Shvartsman, and Jennifer Welch. Brief announcement: virtual mobile nodes for mobile ad hoc networks. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 385–385. ACM, 2004.

[16] Shlomi Dolev, Seth Gilbert, Nancy A. Lynch, Elad Schiller, Alex A. Shvartsman, and Jennifer L. Welch. *Virtual Mobile Nodes for Mobile Ad Hoc Networks*, pages 230–244. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[17] Shlomi Dolev, Seth Gilbert, Nancy A Lynch, Elad Schiller, Alex A Shvartsman, and Jennifer L Welch. Virtual mobile nodes for mobile ad hoc networks. In *International Symposium on Distributed Computing*, pages 230–244. Springer, 2004.

[18] Shlomi Dolev, Seth Gilbert, Elad Schiller, Alex A Shvartsman, and Jennifer Welch. Autonomous virtual mobile nodes. In *Proceedings of the 2005 joint workshop on Foundations of mobile computing*, pages 62–69. ACM, 2005.

[19] Shlomi Dolev, Ariel Hanemann, Elad Michael Schiller, and Shantanu Sharma. Self-stabilizing end-to-end communication in (bounded capacity, omitting, duplicating and non-fifo) dynamic networks. In *Symposium on Self-Stabilizing Systems*, pages 133–147. Springer Berlin Heidelberg, 2012.

[20] Shlomi Dolev, Ronen I Kat, and Elad M Schiller. When consensus meets self-stabilization. *Journal of Computer and System Sciences*, 76(8):884–900, 2010.

[21] Shlomi Dolev and Elad Schiller. Communication adaptive self-stabilizing group membership service. *Parallel and Distributed Systems, IEEE Transactions on*, 14(7):709–720, 2003.

[22] Shlomi Dolev and Elad Schiller. Self-stabilizing group communication in directed networks. *Acta Informatica*, 40(9):609–636, 2004.

[23] Shlomi Dolev, Elad Schiller, and Jennifer L Welch. Random walk for self-stabilizing group communication in ad hoc networks. *Mobile Computing, IEEE Transactions on*, 5(7):893–905, 2006.

[24] Michel Ferreira, Ricardo Fernandes, Hugo Conceição, Wantanee Viriyasitavat, and Ozan K Tonguz. Self-organized traffic control. In *Proceedings of the sev-*

enth ACM international workshop on VehiculAr InterNETworking, pages 85–90. ACM, 2010.

[25] M. R. Hafner, D. Cunningham, L. Caminiti, and D. Del Vecchio. Cooperative collision avoidance at intersections: Algorithms and experiments. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1162–1175, Sept 2013.

[26] Florian Hagenauer, Patrick Baldemaier, Falko Dressler, and Christoph Sommer. Advanced Leader Election for Virtual Traffic Lights. *ZTE Communications, Special Issue on VANET*, 12(1):11–16, March 2014.

[27] Jaap-Henk Hoepman, Andreas Larsson, Elad M Schiller, and Philippas Tsigas. Secure and self-stabilizing clock synchronization in sensor networks. *Theoretical Computer Science*, 412(40):5631–5647, 2011.

[28] Olaf Landsiedel, Thomas Petig, and Elad M Schiller. Dectdma: A decentralized-tdma. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 231–247. Springer International Publishing, 2016.

[29] Pierre Leone and Elad M Schiller. Self-stabilizing tdma algorithms for dynamic wireless ad hoc networks. *International Journal of Distributed Sensor Networks*, 2013, 2013.

[30] Oscar Morales-Ponce, Elad Michael Schiller, and Paolo Falcone. Cooperation with disagreement correction in the presence of communication failures. In *2014 17th IEEE International Conference on Intelligent Transportation Systems, ITSC 2014*, pages 1105–1110, 2014.

[31] Mohamed Mustafa, Marina Papatriantafilou, Elad Michael Schiller, Amir Tohidi, and Philippas Tsigas. Autonomous tdma alignment for vanets. In *Vehicular Technology Conference (VTC Fall), 2012 IEEE*, pages 1–5. IEEE, 2012.

[32] OpenSim Ltd. Omnet++ discreet event simulator, 2017.

[33] Mitra Pahlavan, Marina Papatriantafilou, and Elad M Schiller. Gulliver: a test-bed for developing, demonstrating and prototyping vehicular systems. In *Proceedings of the 9th ACM international symposium on Mobility management and wireless access*, pages 1–8. ACM, 2011.

[34] Thomas Petig, Elad M Schiller, and Philippas Tsigas. Self-stabilizing tdma algorithms for wireless ad-hoc networks without external reference. In *Stabilization, Safety, and Security of Distributed Systems*, pages 354–356. Springer International Publishing, 2013.

[35] Oscar Morales Ponce, Elad Michael Schiller, and Paolo Falcone. Cooperation with disagreement correction in the presence of communication failures. *CoRR*, abs/1408.7035, 2014.

[36] Oscar Morales Ponce, Elad Michael Schiller, and Paolo Falcone. Cooperation with disagreement correction in the presence of communication failures. In *17th International IEEE Conference on Intelligent Transportation Systems, ITSC 2014, Qingdao, China, October 8-11, 2014*, pages 1105–1110. IEEE, 2014.

[37] M. Raynal. *Fault-tolerant Agreement in Synchronous Message-passing Systems.* Synthesis lectures on distributed computing theory. Morgan & Claypool, 2010.

[38] Vladimir Savic, Elad M Schiller, and Marina Papatriantafilou. Aiding autonomous vehicles with fault-tolerant v2v communication. *arXiv preprint arXiv:1710.00692*, 2017.

[39] Vladimir Savic, Elad M Schiller, and Marina Papatriantafilou. Distributed algorithm for collision avoidance at road intersections in the presence of communication failures. *arXiv preprint arXiv:1701.02641*, 2017.

[40] Vladimir Savic, Elad Michael Schiller, and Marina Papatriantafilou. Distributed algorithm for collision avoidance at road intersections in the presence of communication failures. *CoRR*, abs/1701.02641, 2017.

[41] Elad Michael Schiller. Towards predictable vehicular networks. In *Intelligent Transportation Systems*, pages 153–167. Springer International Publishing, 2016.

[42] Shlomi, Chryssis Georgiou, Ioannis Marcoullis, and Elad M Dolev Schiller. Self-stabilizing reconfiguration. In *Networked Systems: 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings*, volume 10299, page 51. Springer, 2017.

[43] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, January 2011.

[44] Cristoph Sommer. Veins the open source vehicular network simulation framework, 2017.

[45] Sumo. Sumo simulation of urban mobility, 2017.

[46] Axel Wegener, Elad M Schiller, Horst Hellbrück, Sándor P Fekete, and Stefan Fischer. Hovering data clouds: A decentralized and self-organizing information system. In *Self-Organizing Systems*, pages 243–247. Springer Berlin Heidelberg, 2006.

A

Default Vehicle List

Spawn Time	**Start Position**	**End Position**
0	East	West
1	East	South
2	West	North
3	North	South
4	South	North
5	South	East
6	South	North
7	East	West
8	South	West
9	South	North
10	South	West
11	South	East
12	North	West
13	East	North
14	West	North
15	West	North
16	South	West
17	West	North
18	South	East
19	South	East