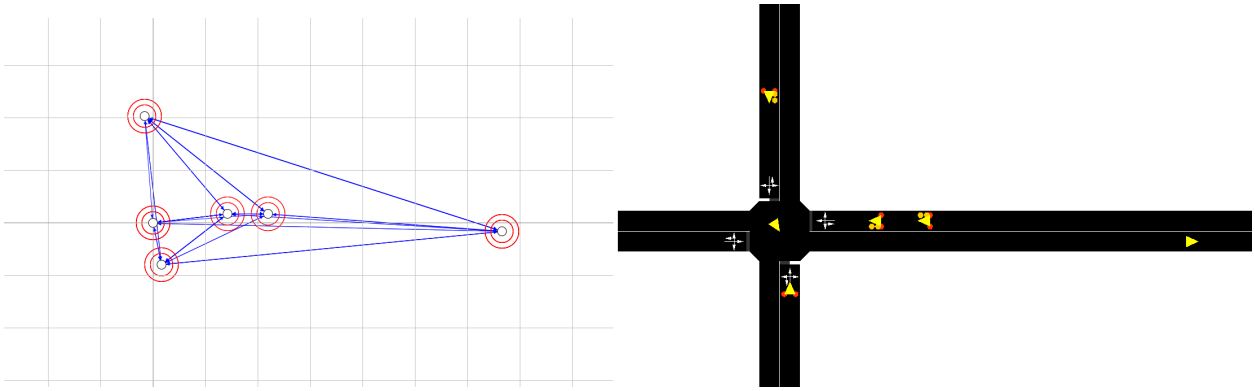
CHALMERS
UNIVERSITY OF TECHNOLOGY



Implementing a Cooperative Driving Scenario using Chaos

Evaluating Chaos in the Context of Inter-vehicle Communication

Master's thesis in Computer Science: Algorithms, Languages and Logic

ARON MANUCHERI

Department of Computer Science & Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016

Implementing a Cooperative Driving Scenario using Chaos

Evaluating Chaos in the Context of Inter-vehicle Communication

ARON MANUCHERI

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

Implementing a Cooperative Driving Scenario using Chaos
Evaluating Chaos in the Context of Inter-vehicle Communication
ARON MANUCHERI

Supervisor: Olaf Landsiedel, Division of Networks and Systems
Examiner: Patrik Jansson, Division of Software Technology

Master's Thesis 2016
Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An image showing a simulated scenario in both SUMO and Cooja side by side. The simulators are kept in sync, simulating the same scenario.

Typeset in LaTeX
Gothenburg, Sweden 2016

Implementing a Cooperative Driving Scenario using Chaos
Evaluating Chaos in the Context of Inter-vehicle Communication
ARON MANUCHERI
Department of Computer Science & Engineering
Chalmers University of Technology

Abstract

This thesis investigates the use of the data-sharing primitive Chaos in the domain of cooperative driving. Chaos is a primitive which natively supports all-to-all data sharing in low-power wireless networks. Chaos enables programmable in-networking processing in conjunction with synchronous transmissions, and this enables efficient sharing of data, in contrast to traditional approaches.
The thesis investigates if Chaos can be used for a cooperative driving scenario consisting of vehicles driving through an intersection. It evaluates if Chaos' unique properties, with regard to all-to-all data sharing, can be used successfully for the scenario. Doing this the thesis evaluates Chaos in the context of inter-vehicle communication, seeing if the efficient data sharing capabilities of Chaos can lay ground for more efficient cooperative driving solutions.
By building a complete simulation system, in which the simulation representing the cooperative-driving scenario is simulated, and which uses Chaos to handle all the data sharing, we can obtain results of how Chaos performs for the scenario, and the feasibility of Chaos in such contexts.
We found in this thesis that Chaos can be successfully used for the scenario of vehicles driving through an intersection, in which the vehicles decide their actions solely by inter-vehicle communication, using Chaos.
Chaos provides high reliability, with successful sharing of data with around 30 participating vehicles. It also provides good scalability, with data sharing delays in the area up to a few hundred milliseconds even for scenarios with around 30 vehicles. Much of this delay was found to be implementation dependent, based on the maximum clock speed of the underlying hardware in which Chaos runs on, and thus not a constraint of Chaos. While not tested experimentally, nothing speaks against that this scalability and reliability should hold for even more vehicles, with a linear increase in data sharing latency. From the evaluation Chaos is found to be a good fit for the domain.

Keywords: Chaos, cooperative driving, inter-vehicle communication, wireless communication, data sharing, vehicular ad hoc network, autonomous vehicles, all-to-all data sharing, network primitive, vehicle communication

Acknowledgements

Special thanks to Olaf Landsiedel, for his help and guidance throughout the thesis as my supervisor; Beshr Al Nahas, for giving valuable advice and help; and Patrik Jansson, for agreeing to be my examiner.

Aron Manucheri, Gothenburg, February 2016

Contents

1

Introduction

This thesis investigates if the data sharing primitive Chaos [15] can be used for inter-vehicle communication. In particular it investigates the use of Chaos in the context of cooperative driving.

This chapter gives an introduction to the thesis. It starts by giving the required background and motivation for the subject, followed by the main problem statement and goal of the thesis, and finally the contributions it brings to the field.

1.1 Context & Motivation

The field of self-driving vehicles is an area of high interest, with research going on on multiple fronts. The activity on the topic is high, and technology which was not too long ago thought of almost as science-fiction has now started to be available for people all over the world [28].

The technological advances are quick. Autonomous trucks have been cleared to drive on US roads for the first time [27]. Modern personal cars have many autonomous aspects, such as being able to park by themselves, drive on freeways, and change lanes whenever needed.

This technology is still new, and for most scenarios the vehicles still need human interaction to help them whenever they face an obstacle, where they don't know what to do. However, with the pace the technology is currently improving, it is not far-fetched to see a future in which we have fully autonomous vehicles for many situations, from transport of goods to transportation of people.

The autonomization of vehicles brings great opportunities to improve on various aspects of today's vehicle- and traffic systems, with improvements to security, efficiency, the environment, and more. With the vehicles being autonomous, they can not only do what humans once did, such as driving themselves to the correct destinations. Self-driving vehicles can see the surroundings in a whole different way,

and process vast amounts of data very quickly. Together with the right infrastructure, these aspects can be used to enable cooperation between vehicles, using inter-vehicle communication to a far greater degree than what is possible with human drivers.

Most self-driving vehicles use the vehicle itself as the center of information, reading its surroundings using different types of sensors, together with various databases (for example high-quality street maps) [12]. However, there are many things the vehicles can't get from these sources alone, but that might be very relevant for the vehicle to help it make better driving decisions. Not only is it hard to know the exact speed of other vehicles, their weight, and so on, but more importantly the vehicle can not know the other vehicles' intentions. It can't, for example, beforehand know which way another vehicle wants to take at an intersection.

By having vehicles communicate with each other, we give the vehicles the ability to cooperate. This opens up many application scenarios, where the vehicles together solve various traffic scenarios. This type of data sharing, and cooperation between vehicles, is used in the notion of cooperative driving. This field can also be extended into creating cooperative Intelligent Transport Systems (ITSs), in which vehicles not only communicate with each other, but also with the traffic control infrastructure, enabling improvements to traffic control and traffic management in general.

Cooperative driving is a very big opportunity in order to enable better traffic- and transportation systems in the future. To give room to the best solutions for cooperative driving, we need good ways to handle the communication between vehicles. Scenarios which have been studied are, for example, vehicles driving in intersection crossings [3], cooperative driving for freeway traffic [13], and cooperative driving at blind crossings [17].

In order to enable cooperative driving, the vehicles not only need smart algorithms and designs in how to handle different scenarios, such as some of the previous mentioned work describes. They also need ways to communicate with other vehicles, and process their data in a fast, efficient and reliable way. This communication needs to work both at low-traffic roads in rural areas with few vehicles, as well as at very crowded intersections. It needs to work with vehicles moving at high speeds, as well as with stationary vehicles. All nearby vehicles need to have information about all surrounding vehicles, and this information needs to be up-to-date and reliable, in order for the vehicles to make good use of it.

There is hardware today to enable wireless communication, different techniques to distribute data so that all participating entities get the same information, and techniques to process this data. However, in the context of cooperative driving, there are no primitives that natively support all-to-all data sharing and processing. This might be because such primitives are relatively new, and thus haven't been tested in that setting yet. Using most other techniques, collection, processing, and dissemination of data are separate steps. Landsiedel, Ferrari, and Zimmerling [15] show

that using the network primitive Chaos these steps are performed inside the network itself, handling all-to-all data sharing efficiently. This can give many benefits in the context of cooperative driving, as many scenarios require that all vehicles have information about all other vehicles around them. Vehicles driving, and thus moving, need this data in a fast and efficient way, as it can get outdated very quickly.

By implementing and evaluating Chaos in the domain of inter-vehicle communication, we can find out if it is a viable alternative for communication between vehicles, and using its unique properties open up possibilities which this data-sharing might bring, such as enabling complex scenarios that requires fast data-sharing or more intricate algorithms. The benefits of performing data-sharing using Chaos can thus be of benefit to the whole field of cooperative driving, and inter-vehicle communication. This thesis evaluates Chaos in this context, evaluating if it works in the setting of cooperative driving, and thus provides these opportunities.

1.2 Problem Statement

The main goal of the thesis is to evaluate Chaos for use in the context of inter-vehicle communication and cooperative driving. This is done by implementing a simulation of a scenario consisting of a number of self-driving vehicles driving at an intersection, and using Chaos as the underlying communication primitive. By doing this the thesis aims to see if Chaos can be used in the field of cooperative driving.

Cooperative-driving requires that the vehicles share data with each other, in a way that can propagate their data to nearby vehicles, as the communication is needed to make relevant decisions. Standard communication solutions today are mostly on the form that the vehicles communicate their data to a single location, which might be either a specific vehicle or a centralized server. This centralized location then processes the data, and then disseminates it back to all vehicles. That approach does propagate and process the data, but has a number of drawbacks. The communication steps are dependent on each other, which essentially divides the communication into three, sequential, steps. This can increase latency, in that each step can't work in parallel with the others. Having a centralized location for processing also undermines the distributed nature of the communication, providing a single point of failure. Also this might require more data than necessary to be sent, depending on what kind of processing that needs to be done on the data. For example, for some scenarios the processing might use data that is stored locally at each vehicle, but which does not need to be sent to the other vehicles.

Vehicles cooperating with each other need data fast, and reliably, in order to work efficiently; we want to avoid a system where a vehicle decides to drive into an intersection, and another vehicle also thinks it can drive on the same lane because it has outdated data. By cooperating with each other, the vehicles communicate

their position and intent to the other vehicles in the Vehicular ad hoc networks (VANETs), avoiding collisions.

We implement the cooperative-driving scenario simulation, using Chaos as the underlying communication primitive. We evaluate the implementation in the context of the cooperative-driving scenario and its constraints, and from that we draw conclusions as to how well Chaos works in the domain, and what benefits Chaos can give if used in real-life vehicles.

This thesis can be said to have fulfilled its goal if the scenario is successfully implemented, and data can be extracted from it, and evaluated. Thus the implementation does not need to give the best current solutions, as that in itself is not the goal. However, it should be noted what properties in Chaos, the implementation, or design, that are missing or need to be improved, in order to achieve better results.

The goal can be formulated as a main research question, with the focus on *can* Chaos be used:

- **Can Chaos be used in the domain of cooperative-driving, to the point that it successfully can be used for a scenario of vehicles driving in an intersection?**

With underlying sub-research questions, which basically consists in *how* can Chaos be implemented in this setting.

- **What is a way to implement a simulated cooperative-driving scenario for autonomous vehicles using Chaos?**
  - How to implement a simulation of the scenario in order to enable the evaluation of Chaos?
  - What needs to be communicated?
  - What kind of processing is needed by Chaos?
  - What are the constraints for the communication?
  - How to implement the communication with regard to the scenario's model, i.e. how get the vehicles' data?

1.3 Contributions

We make the following contributions in this thesis:

- We demonstrate a working simulation of a cooperative-driving scenario. Local data on each vehicle data is shared with and processed by all other vehicles, using Chaos - showing that Chaos can be used for inter-vehicle communication.

- We evaluate Chaos for cooperative-driving. By implementing a working scenario, we provide knowledge for possible improvements to inter-vehicle communication using a new way to share the data.

- We implement a way to test cooperative-driving scenarios in a simulated environment. This gives greater flexibility to test new algorithms and techniques without real hardware and vehicles.

  - This implementation enables other scenarios to be implemented, and evaluated, on supported platforms.

- We show through simulations the scalability of Chaos when increasing the number of vehicles.

These contributions give better insight into the capabilities of Chaos, and show that it is feasible to see Chaos being used as a viable alternative for use in VANETs, and cooperative-driving use cases.

2

Background

This chapter gives an extended background to the thesis. It brings up relevant research and solutions which this thesis is built, and dependent, on. It does not provide comprehensive information about the whole field, but focuses on parts needed to understand the rest of the thesis. It brings up background to the communication and data-sharing, with the focus on Chaos. Following that it takes up relevant information about traffic and vehicle simulators. Finally, it ends with information about the operating system Chaos is ported to, as well as the network simulator used.

2.1 Communication & Data-sharing

In order to enable cooperative driving for different scenarios, the vehicles need to share data between each other in a reliable, and efficient, way. The vehicles involved in a cooperative driving scenario can be said to be in a wireless connected network, where each vehicle is a node in that network. Exactly how this network is formed can vary. A natural way would be an ad-hoc network formed of nearby vehicles, and where new vehicles can join and leave at will, a VANET. Nodes in this network share their data with the other nodes. There are many possible problems related to data sharing that can arise in VANETs. Problems that need to be taken into account are interference, mobility of the nodes, bandwidth, latency, signal strength, and so on.

These are all problems that also show up in other fields. Mobility of nodes can be of importance in other situations except vehicles. The same can be said for interference and latency. There is ongoing research to enable better communication related to these kinds of problems. Ferrari et al. [9] provides the architecture *Glossy*, which focuses on the problem of interference with identical packets, by using constructive interference to enable its communication. This allows efficient flooding of packets in a network. Glossy enables one-to-all delivery in a very efficient way, with high reliability and low latency. Doddavenkatappa, Chan, and Leong [6] provide a dissemination protocol named *Splash*, which handles the problem of content res-

olution, building on Glossy's use of constructive interference together with a way to handle wireless channel diversity. This enables high bandwidth dissemination between nodes.

This thesis builds on, and in all essence focuses most on other communication research, albeit in the same domain as the ones mentioned above. However, instead of solving the problem of all-to-one communication, or fast dissemination between two nodes, the communication primitive in this thesis focuses on all-to-all communication. The focus, and research, is on the network primitive *Chaos* [15].

2.1.1 Chaos

Chaos is an all-to-all data sharing primitive, aimed at wireless networks. It is highly versatile and efficient, and it works well even in multi-hop, lower-power networks [15]. It implements a new way to communicate data among all nodes in a network, and builds on some of the work of Ferrari et al. [9].

Many applications need to share and process data among all units in a network, not least the application in mind for this thesis. Chaos enables this in an efficient way. Before Chaos, the usual way all-to-all data sharing worked was to divide the process into three sequential steps.

- All-to-one data collection
  - Where all nodes send their data to a server, which collects the data from the nodes in the network. This server does not even need to be a participating node in the network, but can be external.
- Centralized processing
  - Where the server processes the data. The type of processing depends on the use case of the communication and data sharing. For example, it might consist of calculating the highest number received, if the incoming data consist of numbers.
- One-to-all dissemination
  - Where the server sends the results to all participating nodes. The data-sharing communication is now finished.

Chaos essentially merges these steps to happen in-network, without an external server, and without a single node which does all processing, instead using in-network processing. This provides a more decentralized solution, with greater efficiency and performance. It does this by parallelizing the three steps described: collection,

processing, and dissemination. This is possible because of two different phenomena. The capture effect, which allows the stronger of two signals at the same frequency to be demodulated, which is used in the first stages of a Chaos round. As well as constructive interference, which enables nodes to receive packets even if multiple senders are transmitting, as long as the packets being transmitted are identical, which is used for the completion part of Chaos.

Chaos is built on two main mechanisms:

- Synchronous transmissions
- User defined merge operators

The nodes synchronously transmit the data they want to share. This means that there might be many different packets transmitted at the same time. Chaos takes advantage of the capture effect to enable nodes to receive the packets, as they can receive the strongest signal. After a node successfully receives a packet, it performs the processing of the payload with its local data, according to the user-defined merge operator, and then sends the merged packet out again.

All sending and receiving of data are highly timed, to make the transmissions synchronous. The round is divided into slots, where the slot length is set to a specific length of time. This is essential to make the transmission and reception of packets timed correctly, in order to take advantage of the capture effect.

Before this a starting node is chosen, which starts the process by transmitting its packet. Quickly all nodes will both be transmitting and receiving data. This creates a distributed system, where all nodes have a part in the data sharing.

When the nodes perform the merging and processing of data, they do it according to a merge operator defined by the user of Chaos. This merge operator is the same for all nodes and defined at compile-time. For example, an operator which takes the 'max' of the data shared. This allows considerable flexibility in what Chaos can be used for. Use cases might be all from simple aggregation of data to more complex computations.

These two main mechanisms both give Chaos a purpose, i.e they enable use cases of Chaos, as well as providing the ability for Chaos to work at all, as the mechanisms are a part of how Chaos works.

Typical Flow

A typical flow for how data gets shared between the nodes using Chaos is shown below. This builds on the information from Landsiedel, Ferrari, and Zimmerling [15], but is simplified for brevity.

At the start of a Chaos round, when the nodes want to start sharing data, all packets will have a bit (a flag) for each participating node in the network. At the start of the round all these flags are set to 0. The user-defined merge operator is provided, which will be used for processing of the data. An initiator node is set, either by having it predefined in the nodes, or using any standard election algorithm. The initiator node needs to be picked before the round starts. All nodes will then be ready, and the following steps happen for each round:

- The nodes set their own bit-flags to 1.
- The chosen initiator node transmits the packet with its data while all other nodes listen for packets.
- All nodes that are close enough manage to receive the packet.
- They merge the payload of the packet with their local data, according to the merge operator.
- They merge their local flags, with the received packet's flags. This guarantees that their flags are set to 1 in the merged packet.
- They send out the new, merged, packets. This happens concurrently among the nodes. In many cases more than one packet will by this point be "in air". Due to the capture effect packets can be received.
- After transmitting, the nodes wait a set amount of time and then start to listen for packets again.
- Repeat until all flags are set to 1 (all nodes have set their own flags, and thus sent and received their data).
- When all flags are set to 1 on a specific node, we have the finalized packet with data from all nodes, and the node enters the completion phase. It floods the network with the completed packet. Constructive interference here comes to play, allowing nodes to receive the finalized packet. Thanks to using techniques from Glossy [9] in this phase the interference will be used constructively, and the packets will be able to be demodulated. Eventually, all nodes will receive the finalized packet.
- If all flags never get set to 1, for example because of node failure, Chaos eventually times out, with partial results. A new round can be initiated to try again.
- The data has been shared and processed by all nodes - we are done.

As mentioned, the above flow is simplified. Chaos relies on the synchronization of

the communication made possible by using a predefined slot length, which sets the times to transmit and listen for packets. There are also special rules, and exceptions, to the flow above to increase performance. For example, a node receiving a packet with nothing new does not send it out again, as it already has merged its own data.

Chaos, with its properties regarding data sharing, and with its main use case being all-to-all data sharing, is interesting for use in cooperative driving scenarios that require vehicles to have information about nearby vehicles. It is used as the main foundation for the communication in our system.

2.2 Vehicle & Traffic Simulators

In order to implement and evaluate our solution, without having access to real self-driving vehicles, a system that can simulate realistic cooperative driving scenarios is needed. This system needs a way to simulate the network communication, as well as the vehicles and traffic. For vehicles the system needs as realistic, and detailed, data of the vehicles and their properties as possible.

When it comes to traffic and vehicle simulators, there exists research which has spun out ready-to-use simulators. With advancements in mathematics and computing related to traffic simulation, these software simulators are increasingly faster, more detailed, and realistic [7].

One type of simulation is that on the microscopic level, which is what is of interest to this thesis. On this level, all individual vehicles are simulated separately, as opposed to the type of simulation looking into collective behavior at a macro-level. By having a microscopic simulation level, the vehicles can be set to have predefined routes, characteristics, and other properties [7]. This enables us to treat a simulated vehicle much in the same way as we would treat a self-driving vehicle. The vehicle will effectively, within eventual constraints of the simulation, behave as a self-driving vehicle trying to get to its destination. For these reasons we use a microscopic vehicle simulator in this thesis.

There are various available microscopic traffic simulators. We use one named Simulation of Urban MObility (SUMO) for our system.

2.2.1 Simulation of Urban MObility

In order to get a realistic simulation of traffic and vehicles, we need an underlying model of the traffic and vehicle data. The model needs to be detailed, with information on a low level of every separate vehicle.

We use the traffic simulation package SUMO [2] as the base of our underlying model. SUMO is an open source traffic simulation package, which simulates traffic and vehicles on a microscopic level.

SUMO is a good choice for use in simulations of VANETs and ITSs for several reasons. It is a stable and well-known package, which also is open source and free to use. The first open source release was in 2002, and it has been continuously updated since then. This gives us a stable and proven platform, which can be used and extended regardless of what happens to the current maintainers, as the source is available. SUMO is written in standard C++, using only portable libraries, which gives us the ability to choose what platform to run it on more freely.

SUMO allows us to create scenarios for use in the simulation. This includes road-layouts, road properties, as well as vehicles and their properties. It supports complex road layouts, with multiple lanes for roads, traffic signs, speed limits, and so on.

SUMO is a purely microscopic simulator, and each vehicle is given explicit identifiers and properties. Details of each vehicle's speed, lane, position, and so on, are available. This is crucial information, and a requirement for efficient simulation of cooperative driving scenarios, as we need specific data about every vehicle to provide realistic cooperation between the vehicles.

Traffic Control Interface

A crucial aspect for using SUMO for the underlying model, is the ability it has for online external interfacing with the simulated environment. The SUMO package includes an interface for online communication with the simulation, named Traffic Control Interface (TraCI). It was originally built specifically for communication with network simulators and VANETs, and is thus a good match for the use-case to simulate a cooperative driving scenario [29]. TraCI allows communication over standard TCP and provides a documented interface for the communication. It also provides a Python API for easy use of the control interface.

By using TraCI we can link a vehicle simulator with a network simulator. The vehicle simulator handles all aspects of the vehicles, and the network simulator handles all aspects of the communication. By linking them we create a bridge between the two different simulations, making the context of vehicles known in the network simulator. In the case of this thesis we use SUMO as the vehicle simulator, and a network simulator which simulates the network environment for Chaos. Chaos is ported to the operating system Contiki, which runs on the emulated hardware provided by the network simulator.

2.3 Contiki

Contiki is a lightweight operating system made specifically for networked sensor systems and the Internet of Things (IoT) [8]. It is portable and designed for resource limited devices. Contiki is open source and provides an extensive set of features for use in low-power Internet communication applications. This includes fully standard IPv6 and IPv4, as well as many low-power wireless standards. Contiki provides a full-featured development environment, where applications are written in the C programming language. The operating system is built around an event-driven structure, but also supports multithreading at a per-process level [8].

Contiki runs on a range of hardware and platforms. Chaos itself is ported to run on Contiki, which has support for the MSP430 platform, running on an emulated TelosB system [19]. This is the main reason as to why specifically Contiki, as opposed to other operating systems, is of interest to the thesis, as it is a requirement to be able to run Chaos.

The Contiki project also provides, as a core feature of the project, a network simulator called Cooja. Cooja allows a Contiki application to be emulated in a network environment without a requirement to run on real hardware [8], which is precisely the needed use case for this thesis.

2.3.1 Cooja Network Simulator

The Cooja network simulator is a sensor network simulator made for the Contiki OS. It is written in Java, and part of the Contiki project [22]. It simulates Contiki motes, where a mote is a single emulated hardware node running Contiki OS. A mote runs an actual compiled and fully executing Contiki system, just like if it was running on physical hardware. Cooja can fully control and analyze these motes, by using Java Native Interface (JNI), getting access to the running native code executing on the motes. Cooja can simulate many different kinds of sensor nodes in the same simulation, and can control and communicate with all of them directly. It has the ability to send events to the motes, and it can also access the entire memory of the motes if needed [22].

Building on the Java platform, together with JNI, Cooja is built in a modular fashion, supporting great extensibility, using plugins written in Java. By default it provides a number of plugins, including visualization of motes and analysis of network activity. It also supports writing plugins which can, for example, modify mote positions and control simulation execution. This is of great use, and one can extend the features and use cases of Cooja by writing plugins for it. This gives full control and access to all simulated motes, as well as control over the simulation environment.

Cooja can be used with a graphical interface, but also be run headless, allowing it to be integrated into automated workflows. It allows fine-grained control over simulation properties, such as step-by-step execution of simulation time. All these properties make Cooja essential for building the simulation system for this thesis.

3

Related Work

This thesis uses a system that consists of a vehicle simulator bridged with a network simulator. There exist similar systems, made for other scenarios and use cases, like the ones discussed in section 3.1.1 and 3.1.2. The contribution of this thesis is a vehicle simulator bridged with the Cooja network simulator, and thus Contiki OS, with the specific purpose to evaluate vehicle-to-vehicle communication. We will discuss other simulators in this chapter to point out similar and distinct features.

Wireless communication techniques for VANETs already exist. This thesis investigates if Chaos can be viable in this field. The techniques today build mostly on standard communication primitives, and we discuss how the protocols and architectures usually work with VANETs today, and how Chaos relates to those architectures.

Finally, there are studies on how to solve the problem of cooperative driving decisions, related to intersection driving. While the focus of this thesis is not on the algorithms to solve cooperative driving scenarios, we need to know how these work, to identify constraints they might put on the communication, and how that relates to Chaos.

3.1 Traffic & Network Simulation Environments

SUMO is often used as an underlying model to evaluate vehicular communication, using vehicle-to-vehicle or vehicle-to-infrastructure communication. This is what this thesis uses SUMO for, and it is used for similar use cases on other systems [2]. In this section we bring up other solutions that bridge vehicle and network simulators, and how they relate to our system.

3.1.1 TraNS

The Traffic and Network Simulation Environment (TraNS) is similar to the vehicle and network simulation environment implemented in this thesis. It integrates SUMO with a network simulator, in its case the network simulator ns2 [18], to provide a simulation of a VANET [24]. It provides graphical representations of the simulation in real life locations, using Google Earth imagery.

The implementation of our solution also bridges SUMO to a network simulator, but in our case Cooja. Both our solution and TraNS use TraCI for the communication with SUMO, and the communication happens in 'real-time' with the network simulator running at the same time as the vehicle simulator. TraNS works so that behavior that happens in the model can be influenced by information exchanged in the network simulation, like slowing down or speeding up vehicles [24]. Our implementation also influences the mobility model, as we want the implementation of our scenario to decide actions for the vehicles to take.

In contrast to TraNS, our solution focuses on providing a feasible model for evaluating the communication, whereas TraNS is aimed more at simulating the traffic behavior which occurs when vehicles communicate. The focus of TraNS is not on the communication, but how it affects traffic.

More specifically there are differences between our solution and TraNS. Our implementation uses the Cooja network simulator, and not the ns2 simulator. The aim of this thesis is to specifically evaluate Chaos in the context of cooperative driving, and our implementation of the network and vehicle simulation reflect this. While TraNS studies different ways VANETs work, and how they affect traffic, our solution communicates information to full-featured Contiki OS motes. These motes have capability to execute C-code for the platform they run on. This allows each single vehicle to make decisions on its own, and vehicles might have totally different use cases for the data. Nothing in our design prohibits different motes, with different code, from receiving and using the vehicle data, and communicating with each other, as long as they communicate in a standard way after processing the data.

So while TraNS is aimed more on a high-level overview of how different VANET scenarios can affect traffic, our solution is aimed to enable Contiki applications to be run in a context of a vehicle in traffic. Our implementation aims to position the vehicles at the correct position in the network simulation, as well as to update them with the needed data from the vehicle model. The Chaos application can then act as if it was running on real vehicles, where the simulated application is unaware of whether it is simulated or not. It sends actions calculated with the help of the data sharing with other simulated vehicles. These actions get sent to, and change, the underlying vehicle model, effectively making the simulated scenario progress according to the implementation.

That is not to say that TraNS and our implementation does not have similarities,

as in the way the communication between the network and traffic simulator works. They differ in how to implement applications and what they simulate, as well as working for different network simulators. To enable testing of a real-life implementation of an application running on a vehicle system, as opposed to simulating how an application affects the traffic and vehicle model, then our design is better suited to the task.

3.1.2 iTETRIS

The traffic and wireless simulation and management platform iTETRIS has similarities to our simulation environment. iTETRIS is an integrated wireless traffic platform for real-time road traffic management solutions [11]. The main objective of the software is to achieve simulation of communication and behavior of Intelligent Transport Systems, making use of cooperative communication systems.

The justification for iTETRIS lies in testing systems before they are being deployed in the real world, and it allows simulations between vehicles and infrastructure on a large scale, even over whole cities if needed [26].

The simulation aspects of iTETRIS work in a similar way as TraNS, by using SUMO as the vehicle and traffic model, and the ns3 network simulator for the wireless communication simulation. Both solutions, iTETRIS and TraNS, use SUMO for the underlying traffic model. iTETRIS builds on ns3 for the network simulation, as opposed to TraNS which uses ns2. ns3 is the current, and more feature rich version [4] of the ns network simulator. The simulation aspects of iTETRIS are a more modern alternative to TraNS in many ways. It is a system on a larger scale than TraNS, with broader use-cases, and the simulation part is mostly a building block required for its main goals of traffic management.

Many of the differences of the simulation environment in our work compared to iTETRIS are similar to the differences our solution has to TraNS. Our simulation environment uses Cooja for the network simulation and is tailored to evaluate Chaos in a cooperative driving context. iTETRIS focuses more on evaluating the benefits and impact of different ITS systems than about evaluating a specific technology. It is aimed at large scale evaluations, with several thousands of vehicles, and how ITS systems affect these simulations. Our solution, by contrast, focuses on simulating a specific scenario with a much smaller number of participating vehicles, and see if Chaos can be used for that particular scenario. Our goal is, in contrast to iTETRIS, not to evaluate the possible benefits of ITS and cooperative driving. Nor is the goal to see if such an implemented scenario could be of use to improve traffic, but to evaluate the properties of Chaos, and its potential use in the context of cooperative driving.

3.2 Communication in VANETs

VANETs traditionally work by integrating an ad hoc network, Wireless Local Area Network (WLAN), and cellular technology to achieve inter-vehicle communication.

What makes these vehicular networks special, and thus having different constraints compared to more conventional networks, is the movement of the nodes, new and unevaluated application scenarios, as well as the communication architecture.

Research regarding communication often goes into different models for the communication. Li and Wang [16] bring up three possible models aimed at VANETs:

- Pure cellular communication
  - Where the vehicles communicate over cellular networks, for example using 3G.
- Pure ad hoc communication
  - Where the vehicles communicate directly with each other in a local network.
- Hybrid communication
  - Where the vehicles use a combination of ad hoc communication and cellular communication.

Chaos, and our solution, is mostly related to the aspect of ad hoc communication, where the vehicles are on the same, physical, local network. Communication using cellular technology, for example to join and leave the network, are not in the scope of this thesis. Instead it focuses solely on the local communication between nearby vehicles. We thus focus on the research regarding communication in ad hoc networks in this section.

3.2.1 Ad Hoc Routing

VANETs can be seen as a special form of Mobile ad hoc networks (MANETs). Thus, protocols used in MANETs can be adapted to VANETs. However, the distinguishing features of vehicle networks, such as movement, results in that these protocols usually perform badly [16]. Ad hoc On-Demand Distance Vector Routing (AODV) [23] is an example used in MANETs which performs badly in common vehicle scenarios [16]. There is ongoing research in how to solve these issues, by modifying and improving the protocols to better fit vehicle and traffic scenarios. In [21] Nambood-

iri, Agarwal, and Gao present research using gateway vehicles together with speed and location information to predict and improve links and routes in the wireless network.

The constraints on routing and communication between vehicles show that the network connections and its characteristics can be different when it comes to inter-vehicle communication as compared to more standard network types. This work is relevant to see where the faults might be related to how Chaos works, and what is a result of the general area of communication in VANETs, regardless of which technique is used for the communication.

3.3 Cooperative Driving Scenarios

The main purpose of the communication between the vehicles is to enable different applications and scenarios. By studying scenarios and applications, one can obtain information in how to develop the communication, in order to meet the applications' needs. While the focus of this thesis is not in these applications by themselves, knowing what possible constraints they might put on the communication is of use when evaluating the feasibility of Chaos being a part of that communication.

The same holds for the algorithms used by the applications. With the aim of evaluating Chaos in the context of cooperative driving, we want the evaluation to be as realistic as possible. So while our goal is not to design and develop new algorithms for cooperative driving, we need information about how such algorithms work, and what kind of data they need. What is important, from the point of view of Chaos, is if the data can be shared efficiently, and if the needed processing can be performed by the merge operator.

Li and Wang [17] study cooperative driving at blind crossings, using inter-vehicle communication. This scenario relates well to our work, as it is the scenario used to evaluate Chaos. They bring up a concept of driving patterns, with regard to safety, in order to represent the movements of the vehicles at crossings. They found a solution in which all allowable movements can be found, and then formulated algorithms to determine which of these movements give the least execution time. This gives both security, in the form of no collisions, and efficiency to the solution.

In common among all findings, is that the scenarios share what kind of data they require, with the main parts being speed, position, and direction. These data are utilized differently, but that does not affect the communication needed to share the data, which is our focus.

4

Design & Architecture

This chapter brings up the architecture of our system, which includes the implemented simulation platform, and our use of Chaos. It also brings up the design goals and choices. Additionally, this chapter takes up our design of the Chaos application. The design, and parts of the architecture mentioned here are explained in more detail in the implementation chapter, at chapter 5.

4.1 Architecture of Simulation System & Application

To evaluate Chaos, we designed a simulation system, which can represent a scenario of moving vehicles sharing data with each other.

This section takes up the architecture and design of this system and the application running on the simulated motes. After explaining the design, we analyse the it, with commentary on design goals and choices. We end with design challenges and alternatives.

The communication between the different parts of the system is coordinated and handled by a controller program, which ties the different parts together, acting as a middle-layer glue between them. We have one part of the system which consists of the SUMO vehicle simulator which handles its associated vehicle model. All communication with SUMO goes through the communication interface TraCI, which enables access to the simulation and its data. We also have one part which consists of the Cooja network simulator, which handles the simulation of the network and its motes. This simulator includes two sub-systems, which can directly communicate with the Cooja simulator, as plugins. The first of these is the time-handling plugin. This plugin controls the execution of the simulation, starting and stopping it at request. The second plugin is the mobility plugin. This plugin handles the mobility of all the simulated network motes. It can change the positions of all motes in the simulated space whenever needed.

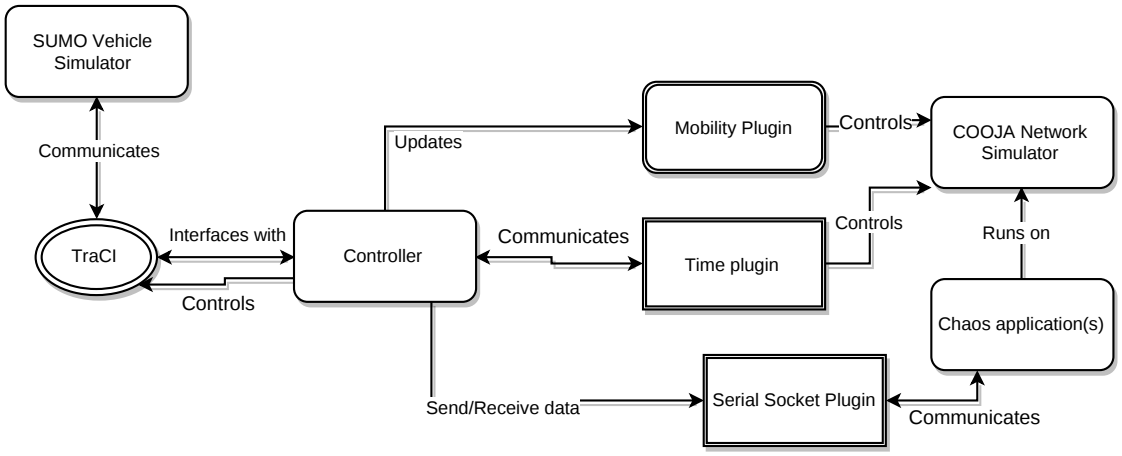


Figure 4.1: A high level overview of the architecture of the system. The controller has access to the SUMO simulator, using TraCI. It also has access to the Cooja simulator, and its related parts. All communication between Cooja and SUMO thus pass the controller.

The last part is the Chaos application. The Chaos application is running inside the Cooja simulated environment, on each of the simulated motes as a fully executing Contiki OS process. Each mote simulated corresponds to a simulated vehicle in SUMO. Thus the number of motes are the same as the number of simulated vehicles. The Chaos application is the application performing the actual sharing of data, and it acts like if it would have been on a real vehicle. The application has two communication channels: the serial port and Chaos. The serial port is used to get external data from the vehicle simulator, corresponding to data that would be available in a real vehicle. Chaos is used to share data with the other motes in the network.

The application processes the data it has with data it obtains from the data sharing, and from that calculates possible actions to be applied to the vehicle model. The serial line port gets its data from a bundled plugin for Cooja, which enables sending messages through a socket which gets forwarded to the serial interface of the mote. It is also on this socket the mote sends actions to the controller, informing the mote's corresponding SUMO simulated vehicle that it wants to perform an action.

A high level overview of the whole system can be seen in Figure 4.1. We will go through each of the subsystems, and bring up how they are designed and how they communicate.

4.1.1 SUMO

The SUMO system simulates all the vehicles and traffic, and acts as the vehicle model on which we build our scenario. It simulates all vehicles on a microscopic

level, where each simulated vehicle has its own properties. It is from this system that we get continuous information about the different vehicles and their data. It is also to this system all actions taken by our Chaos application gets propagated, affecting the SUMO vehicle model.

When running a scenario, we first provide SUMO with the needed scenario files for that particular scenario. This includes the number of vehicles that participate in the simulation, their routes, and their special properties (is it a car or a bus?). We also provide the road traffic network, which in this case is an intersection. This includes how many lanes the roads should have, if it is a three-way or four-way intersection, possible speed limits, and so on. SUMO uses these files to build up a simulation of the road network, and all the vehicles, which will represent self-driving vehicles, their routes, and such.

When the simulation is started, the vehicles will behave according to SUMO's simulation model, driving the best they can toward their destination. At every step in the simulation, the vehicles will progress according to their set properties and routes, and the simulation rules, until they reach their destination. Each simulation step corresponds to a set number of seconds in the simulation, so all values are updated at every discrete step according to the changes since the last step.

However, the vehicles are also affected by external input. The data about each simulated vehicle is continuously polled for by the controller program, which sends it on to the corresponding mote. The mote, running the Chaos application process, parses the data and saves it as its own active data. The Chaos applications run in conjunction to the SUMO simulation, and they might decide on an action based on the data retrieved from the simulation.

Whenever any of the mote processes decides on an action, it gets sent to the controller. The controller parses the action and immediately sends it to the SUMO simulation. This is the external input which can affect the simulation and vehicle model, which otherwise is deterministic for every run with the same scenario files. An action might be to change the route for a vehicle, or change a vehicle's speed. These changes can propagate and give different outcomes for the simulation.

In this way SUMO provides a model for our system, which we can adapt to many possible scenarios, retrieve data from, as well as make changes to while it is running, seeing the results of actions taken and how they affect the simulation.

TraCI

All communication with SUMO happens through the interface TraCI. The controller sends, using TraCI, specific packets, which are interpreted by SUMO as actions on the simulation model. These actions can be to slow down a specific vehicle, change a vehicle's route, remove a vehicle from the model, and so on. It is through this

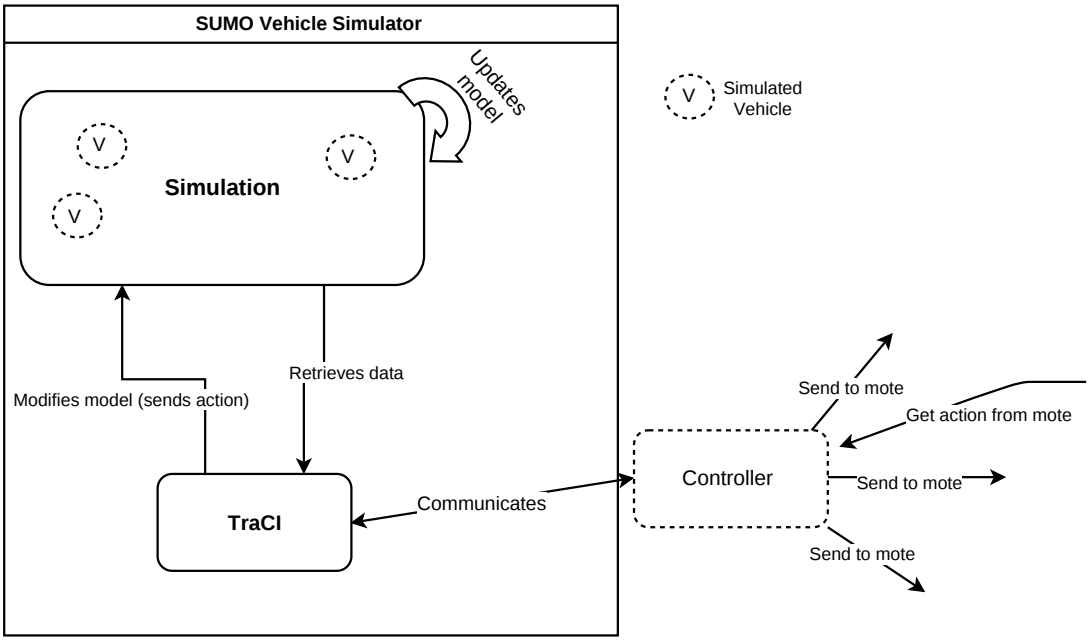


Figure 4.2: An overview of the design of the vehicle simulation system, built on SUMO, and its communication. The controller inferfaces with the simulation by sending data using TraCI. These data contain actions which change the underlying vehicle model. In the same way the controller polls for the underlying model's state, sending the data to the simulated Cooja motes.

interface all external changes to the model goes. Likewise, whenever the controller wants to extract data from the vehicle model, it does this by querying TraCI, which obtains and sends the data to the controller. TraCI works on-line, on a currently running simulation, and can thus alter the simulation as it progresses depending on what is sent to it. Figure 4.2 shows the different parts involved in the SUMO system, and how they communicate.

4.1.2 Cooja

Cooja is the base of the network simulation, and simulation of the motes. It simulates the network, allowing us to run the Chaos applications without real hardware. Cooja simulates all aspects of the network, and handles things such as radio communication, number of motes, signal strength, positions of motes, and so on. It provides the most essential things related to the network environment, needed to evaluate Chaos realistically. This while still being able to use the same codebase for our application as if we had run the application on physical hardware, as the motes emulate full-featured Contiki OS systems.

Our setup of Cooja consists of loading it with a configuration file that corresponds to the scenario-file loaded into SUMO. This includes having a mote for each vehicle in

the SUMO simulation, where each mote is loaded with the Chaos application as its firmware. These motes can receive data from the serial-line interface, by opening up a socket through the serial socket plugin, bundled with Cooja. This allows external processes to send messages, which gets transferred to the serial-line interfaces of the motes.

The positions of the motes get set and updated by a plugin to Cooja. This mobility plugin gets information about the simulated vehicles' positions, and updates the motes in the Cooja simulation accordingly.

The execution of the motes are controlled by Cooja. In order to keep the execution of the motes not drifting apart from the executed time in the SUMO simulation, which would make it impossible to take any relevant actions, we have a time handler plugin that controls the execution.

Figure 4.3 gives a detailed overview of the different parts, related to the network simulation, discussed in this section, and how they communicate with each other.

Mobility Plugin

We want the network simulation environment to correspond as closely as possible to the actual model environment, which is the simulated vehicles in the SUMO simulation. This includes having the correct spatial positions of the motes in both the vehicle simulation and the network simulation, so that we get correct results with regard to for example signal strength. This includes having the motes' positions being updated when needed, to simulate movement.

The mobility plugin handles this. As Figure 4.3 shows, it has a constant connection to the controller process, which feeds it data about the vehicle simulation every step of the simulation. The plugin receives the data, and immediately, through a direct interface to the simulation in Cooja, updates the positions of all motes to their received position values. This ensures that all motes always have the same positions as their corresponding vehicles do in the vehicle model, keeping the two simulations in sync with each other. Section 5.1.4 shows, in more detail, where in the flow of the system the mobility plugin is.

Time Handler Plugin

One crucial aspect of our design is to keep the SUMO simulation of the vehicles and the Cooja simulation of the motes in sync. That is, if a specific set of simulated time passes in the SUMO simulation, that same amount of time need to pass in the Cooja simulation before the execution of the SUMO simulation can continue. Note here that we do not talk about real-life time, as computers vary their scheduling and

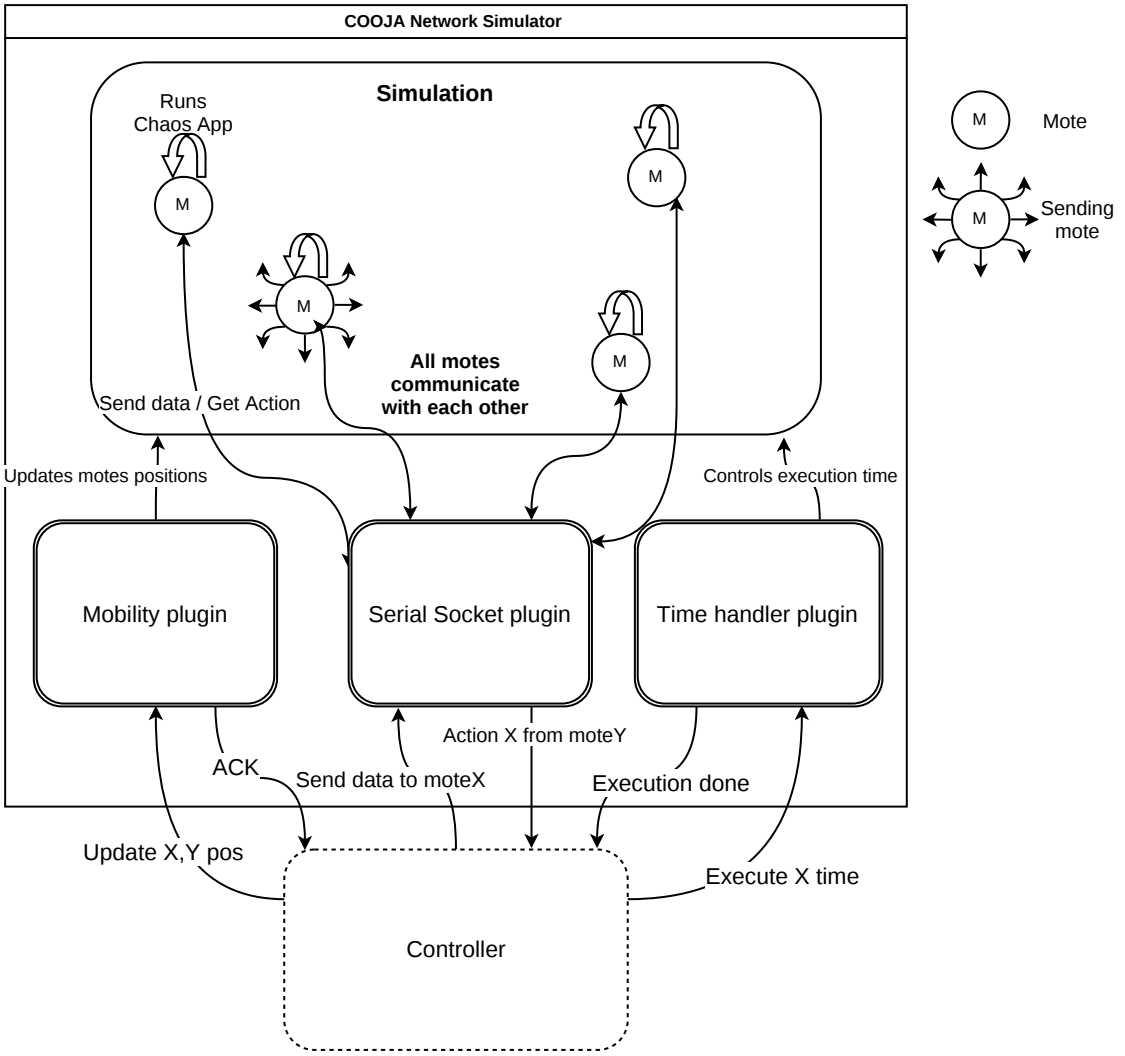


Figure 4.3: An overview of the design of the network simulation, and the parts it communicates with. The Cooja simulation handles multiple motes, each running an instance of the Chaos application. The controller sends updated positions, retrieved from the SUMO simulator, to the mobility plugin, which updates the motes' positions in the Cooja simulation. The controller also sends data related to each simulated vehicle's properties to that vehicle's related mote using the serial socket plugin. It is also using this plugin that the motes send actions to the controller, which the controller processes and sends to the SUMO simulation. The controller also interfaces with the time handler plugin, notifying it when to execute a new simulation step. All in all this enables the controller to keep the Cooja simulation in sync with the vehicle model, and both send and retrieve all needed data.

processing power, but the executed simulation time in both environments.

In order to do this we have a time handler plugin, which interfaces with the Cooja simulator. As Figure 4.3 shows, it gets information from the controller whether there has been progress in the vehicle model simulation. It then proceeds to start

the simulation of the motes in Cooja, letting them execute, in simulation time, for exactly the same time as the SUMO simulation has executed. It then immediately stops the execution. It tells the controller that the execution is done for this step, and proceeds to wait for the next step. In this way the network simulation is always in sync with the vehicle simulation. Section 5.1.4 shows more in-depth all steps required to achieve this synchronization.

Simulated Motes

The parts discussed in this system, the vehicle simulation, network simulation, and all the supporting parts, were created in order to reach this thesis's goals, namely to evaluate how Chaos performs in the context of a cooperative driving scenario. They provide the system that enables us to implement the Chaos application in a relevant way.

The Chaos application runs on each of the motes running in the network simulation. It has no knowledge of the system around it. It does not know that its position changes in the network simulation by the mobility plugin, nor does it know that its execution gets paused and resumed at steady intervals by the time handler plugin. It acts as if it was running a Contiki-OS process, executing an application and using Chaos for the data-sharing. All connections to the external systems are only there if it is feasible to presume that a similar connection would exist in a real life vehicle. The application reads from its serial line interface, where it can obtain data about its corresponding vehicle speed, position, and properties. Just like if it was attached to a real life vehicle, which has this data available. Figure 4.3 shows the place of the motes in the simulated system, and which parts they communicate with.

A mote only knows about its corresponding vehicle from the serial line interface. It does not get any information about neighboring vehicles from the serial line interface. The only external communication it sends over the serial line interface is when it decides on an action. This is sent to the controller, which handles the action by propagating it to SUMO. This can be seen as, for example, if the running application decided that the vehicle should slow down, which is a possible feature a computer can tell a vehicle it is attached to in the real world to do. Otherwise it works as any other Contiki process. It communicates with the other motes in the simulated network, using Cooja as the network simulator, and Chaos as the data-sharing primitive.

4.1.3 Controller

The controller works like a spider-in-the-web, having some connection to most of the different parts of the system. It is the controller that launches and controls the simulation. It controls, and has access to information about, both the vehicle

and the network simulation, bridging them together. As seen in Figure 4.2 and 4.3, TraCI is used for 2-way communication between SUMO and the controller. The data exchange is always initiated from the controller's side, which either reads or writes to the vehicle model. Finally the controller fully controls the execution of the vehicle simulation, controlling whether it can progress or not.

The controller is tied to the network simulation with connections to the mobility plugin, the time handler plugin, and the serial socket plugin, and thus all Cooja motes. By utilizing these connections it can fully control the network simulation environment, updating the positions of the motes, controlling execution time, and sending and receiving data to and from motes.

It is also the controller which the user will interact with. The user will start the controller program, and supply it with needed input parameters, such as what scenario to run. The controller will then launch this simulation, and execute it, before closing down.

These parts, controlled by the controller, gives us a powerful solution, which uses already available simulators without direct modification of their source, instead extending them with the needed functionality and enabling communication between them. This approach, with simulating the scenarios step by step, decoupling simulation time for the actual processing time, also allows slower computers to execute scenarios, as the simulations do not get affected by the overhead of the communication, as they wait until everything is in sync before they execute.

4.2 Design of Chaos Application

With the simulation system and communication between the network simulator and vehicle simulator in place, we have the prerequisites needed for the design of the cooperative driving scenario, using Chaos. Here we take up the design of the scenario, and the Chaos application, which we aim to evaluate. Figure 4.4 shows the main design flow of the application.

4.2.1 Communication

The application consists of three main tasks. The first task is that of communication. The application, running on each simulated mote, gets data from the serial port, and communicates data back over the same interface, based on the outcome of the scenario and the data-sharing.

To achieve this, data need to be fed into the serial port, which is already done based on the design of our simulation system, which provides data from the controller to the

serial port for each mote. In the same way the mote can easily send back data, which is interpreted by the vehicle simulation. More details of how this is implemented can be found in the implementation chapter, at chapter 5. The implementation simply reads data from the serial port between every Chaos round, and saves it to memory. In the same way it also sends data back over the port after each finished Chaos round, based on the scenario outcome. Figure 4.4 shows this communication as part the main application flow.

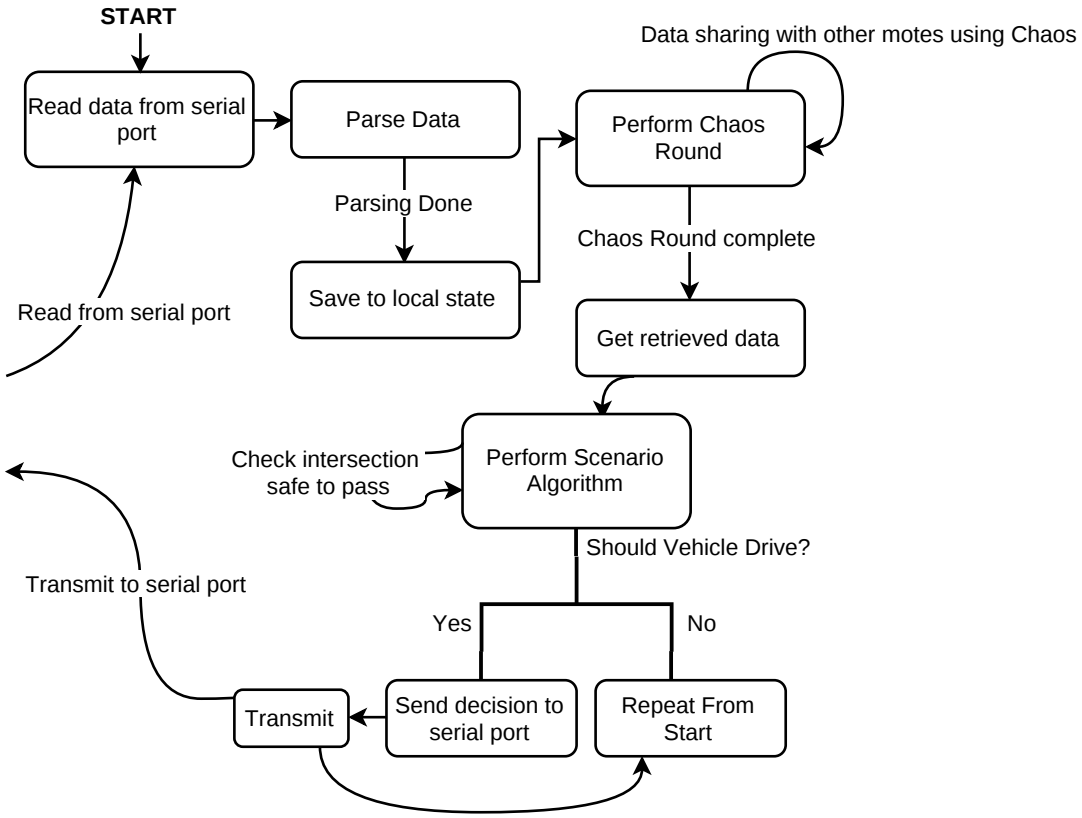


Figure 4.4: An overview of the design of the whole execution flow of the Chaos application. The application continuously reads data from the serial line port, performs a Chaos round to share that data, and based on the shared data performs the scenario algorithm. If a decision is taken, it is transmitted to the controller over the serial port, and then the flow repeats.

4.2.2 Merge Operator

With communication between the various parts of the system and the application set up, so that we can get external data to the application, the second task is that of the configuration of Chaos. The main design decision here is related to defining and designing a merge operator, suitable for the task.

The overall design of the merge operator is based on a rather straight-forward bitwise OR operation of all motes' data into a bigger structure holding the data. More details about this is to be found in the implementation chapter, chapter 5. The design works so that we set up a structure of data, with a specific length based on the number of participating nodes in the Chaos round. This in turn corresponds to the number of vehicles participating in the scenario. Each node manipulates this structure, but only at a specific index of the structure, assigned to that particular node. One can see it as an array of boxes, where each node gets its own box, where they can put their data. The nodes then add the data they want to share at their specific indexes. This data includes vehicle properties. For more details of exactly what data, and how it is added, see the implementation chapter, at chapter 5.

In this way, when we perform the all-to-all data sharing, each node adds information to the shared data, without affecting any of the other nodes' data, nor having to take into account the order of the data, as long as it has its own node index.

In the same way data can be extracted easily, by simply reading the data at the nodes' specific index. We have an operator consisting of an elementwise application of bitwise OR to the individual elements of the array. This enables all nodes (and thus vehicles) to share their data using Chaos, in a single round.

4.2.3 Scenario

Having the communication between the different parts done, as well as the merge operator for Chaos set up, the final part is that of the design of the cooperative driving scenario itself.

The scenario designed is that of an intersection, with vehicles driving in all possible directions (north/south/east/west). The vehicles communicate their data, including speed and destination. Each vehicle also have a 'Passed' property, indicating if they have passed the intersection, which is part of the shared data. By utilizing the data the scenario then decides which vehicles should go through the intersection, and when.

As the focus we have here is not to find the smartest and most efficient algorithm to solve the problem, the design is rather straight forward. We focus just on sending all relevant data, and testing if simple decisions can be made to make the scenario progress. Our design of the scenario is thus quite simple.

The algorithm design starts with all vehicles driving towards the intersection. When a vehicle is close to the intersection, it stops to a halt. Eventually all vehicles are halted. Using Chaos, this information will soon be propagated to all vehicles. When the vehicles see that all vehicles are currently stationary, one of the vehicles gets an action to drive through the intersection. When it has passed, it sets Passsed

to True. Eventually this information gets propagated to all vehicles using Chaos, and now the next vehicle can drive through the intersection. This continues until all vehicles have passed the intersection. More details about the algorithm and its implementation are found in the implementation chapter 5.

4.3 Design Goals & Choices

In designing our system, we have a main goal. The system should provide an environment to make the simulated application appear to run on real-life self driving vehicles. If we deviate too far from that, the evaluated data would not be as usable, and whilst not necessarily useless, would not allow us to draw conclusions with as high confidence.

We set out to try not to focus too much work implementing all needed parts for the simulation environment. Our goal is not to provide a vehicle simulation environment, that is more of a necessity in order to be able to evaluate Chaos, without spending a lot of resources on equipment and vehicles. Thus we made the design choice to go with already available software as much as possible. This also gives us more features to work with, and more proven and battle-tested systems to use, which give confidence to our simulation environment. Thus we did not create our own vehicle simulation, but instead used SUMO, and with Chaos already being ported to a supported platform in Cooja it was a straight forward choice to choose that as our network simulator.

Another benefit of this is that we can more easily create new scenarios to test. The scenario itself should not be hard-coded into our system, but the user of the system can provide different parameters, such as number of vehicles, and their directions. The road traffic network should be able to be replaced easily, using the standard tools provided by SUMO. In this way we do not have to redo big parts of the system in order to test possible future scenarios or different variations of the intersection scenario.

We wanted our system to be runnable on all platforms which SUMO and Cooja runs on. SUMO is portable by design, and Cooja also works on multiple platforms, so being able to take advantage of that let users of our system test Chaos (or their own Contiki OS programs) using their platform of choice.

The design is modular, with messages being sent between the different parts. This is also by design. A modularized design has obvious benefits, with separation of concern, easier extendability, and easier debugging. Cooja does not know, and does not need to know, how the controller communicates with SUMO, it just wants updated data. Likewise SUMO can run without knowing how the data extracted from it is used. This also allows users to change one of the parts, perhaps replacing SUMO with something else, or choosing another network simulator, without having

to rewrite the whole system, merely just changing some aspects of the controller program.

We want our system to be as performant as possible, but also reliable. All communication takes place offline, with messages being sent only locally. We want the performance to be good enough even for scenarios which involve more than a few vehicles. While performance is important, it does not trump reliability of the results. We choose more correct results over faster results whenever possible. Performance should never affect the simulation of our implemented scenario, but merely the time it takes for the system to perform the simulation. This is because we can always automate the simulations and let them run over night, but only if we ensure that they run correctly. Performance does help with evaluating, as it gives a faster feedback loop, so it is still a goal to keep our system as lean, and efficient as possible.

Finally we want to be able to evaluate our solutions in a good way. That means we need to be able to extract relevant data from the system. Both Cooja and SUMO have methods to log their execution, and shared data passed between the simulators first goes through the controller, which can use that information as well. This allows us many ways to get data, which we can aggregate and evaluate.

4.4 Challenges

Building such a system, with aforementioned goals, posed some challenges. One of the biggest was that of synchronizing the different parts of the system, which are completely independent systems that can even run on different computer. Starting them at the same time, and hoping they would align correctly is not reliable enough, and also does not allow us to do any processing, perhaps on data to evaluate, without skewing the execution further. Our solution to completely control the execution of both programs, and require each system to perform one simulation step at a time solves this. It induces overhead of having to pause and resume the simulation very often, but gives a reliable and consistent system. Implementing this required us to create plugins for Cooja, using Cooja's simulation API to get fine grained control over the simulation. The communication between the controller and the Cooja simulation then need to be consistent.

One challenge with us having to stop execution of Cooja is that this effectively also halts all execution of the simulated motes. However, we still want to send information to these motes, which they should have when started. This is solved by queuing messages on the serial line socket interface, and then parsing the most recent message when execution starts. The implementation chapter 5 further explains this.

Another challenge is that of performance and scalability. Cooja itself requires some overhead to run, as it presents a lot of tools to handle and analyse simulated systems.

SUMO itself can handle a high amount of vehicles efficiently, but for each vehicle, we will have a fully executing simulated mote in our network simulator. This can, when the number of motes get high, slow down the simulation. This is not such a big issue thanks to our design not focusing on the speed of the simulation, but instead only taking into account simulated time executed in the simulation. One second of simulated time can be much more, or much less, than one second of real time, depending on the hardware the system is executed on, as well as what kind of program is executing on the motes. In all cases, the results should be consistent across systems and hardware, as the simulated time will always be allowed to execute completely before next step in our simulation environment.

4.5 Alternative Solutions

Our solution, using an available vehicle simulator and bridging it with a network simulator, is not the first of its type. This thesis has brought up related work with similar systems, which allows coupling of network simulators and traffic simulators. One alternative solution to our design could have been to modify these, in the case they are open-source, to our use case, and build on already implemented work. However, as this would still require a reimplementation of the network simulation aspect, as well as current solutions not aligning specifically to our use case, we felt that a home grown solution would be of better use, and more aligned with our needs.

Another alternative solution would have been to make Chaos run on a different kind of network simulator. This would require porting Chaos to a supported platform other than Contiki OS, which is out of the scope for this thesis. Also even here we would not have a custom-made system made for our specific purposes.

We choose a message based approach, where the controller is an external program. Another alternative might be to integrate the controller into to network environment, perhaps as a plugin. This would give the effect that one can integrate some parts related to controlling the network environment to the controller. This might increase performance, but it also would increase complexity, and decrease portability. Our solution sends messages locally, which is very fast. However there is no inherent design decision which disallows the network simulation to run on a completely different computer, perhaps on the other side of the world, than the controller and the vehicle simulation, which is a feature possible in our solution.

5

Implementation

This chapter brings up the implementation of the main parts of our solution. This includes the simulation environment, and its implementation details, as well as the Chaos application running on the simulated Cooja motes. Many of the main concepts of the architecture and its functionality can be found in the previous design chapter. However, this chapter goes more in-depth in many parts, and talks more about the actual implementation, making the chapter needed to fully understand how to implement the system.

5.1 Simulation Environment & Controller

The simulation environment gives us a foundation to perform simulations of Chaos in the context of cooperative driving. The main architecture and its related design decisions can be found in the design chapter. Below follows notable implementation details of the different parts of the system.

5.1.1 Controller

The main 'brains' of our simulation system is the already complex and intricate vehicle and network simulators. The controller utilizes the functionality these systems provide, to enable the implementation of a cooperative driving scenario using the system. The main design of the controller system can be found in the previous design chapter.

The controller is implemented as a Python program, run from the command line, and started directly by the user of the system.

The controller program requires that the SUMO simulator is installed on the system and available on the user's path. It also requires that the Cooja simulator is set up, loaded with the scenario configuration that the user wishes to simulate.

This in turn requires the user to have compiled and loaded up the corresponding application which is to be simulated in Cooja. More on this, and this application's implementation can be found in section 5.2.

When started, the controller can be passed a few arguments. The most important one is the "–scenario" argument. To this argument the user supplies a scenario file. This file specifies the needed information about the scenario the user wants to simulate, such as the number of vehicles, their starting positions, and their directions.

The controller also takes in optional arguments '–nogui' and '–nodebug', which provides the flexibility to run the simulation without the corresponding SUMO-GUI, as well as disabling debug prints that are used for development and troubleshooting.

When started, the controller parses the scenario file and sets the needed variables based on the information parsed. It then proceeds to open a new subprocess. This subprocess starts the installed SUMO simulator, providing it with the required parameters based on the loaded scenario. This loads the SUMO binary, either headless or with a provided GUI, depending on the supplied arguments. Following that, the controller sets up all needed connections. It connects to the newly created SUMO process, using the TraCI interface. It also connects to the Cooja simulator, opening connections to the time-tracker plugin, the mobility plugin, as well as to all serial socket connections on all Cooja motes. All of these are standard TCP-connections, running on localhost, but can also be run on remote systems if required.

After that the controller starts executing the main program loop, whose execution flow can be found in subsection 5.1.4. In this loop the logic related to the scenario, but not running on the Chaos application, can be found. This logic includes what to do when receiving an action from a mote, and what to do when a car finds itself at a specific position related to the scenario. However, there is never any data shared, related to scenario decisions made, in the controller. This because that would defeat the purpose of the evaluation of the Chaos application, where we want the only way of sharing data between the vehicles to be using Chaos.

5.1.2 Time Handler

The time handler plugin, which keeps the different parts of the simulation synchronized, is implemented as a Cooja plugin, written in standard Java 7, using the Cooja Java API [5]. This API allows us to tie into the network simulation and control it.

The plugin is installed in Cooja, and loaded up whenever we start a new simulation session. It is implemented by constantly listening on a network socket for incoming

connections, on a predefined port, defined in both the plugin and the controller. The plugin wants only one active connection, so when it has established a connection to the controller, it stops listening for new connections.

Directly after a connection is set up, it creates a new thread, continuously listening for messages from the established network connection. The first message it looks for from the controller is a setup message. This message consists of a string starting with "SETUP". This message also contains two numbers, containing the step length in milliseconds and the maximum number of steps before simulation end.

The plugin saves these numbers, and responds to the controller that it has been "set up". It is now ready to control the network simulation running in Cooja. It listens for messages from the controller, more specifically messages saying "CONTROLLER_READY".

Whenever it obtains this message, it immediately calls the Cooja Java API, to schedule that the network simulation should be executed for "step length" number of milliseconds. This is done by creating a TimeEvent, which is an event that is executed at a specific time by the Cooja simulator. This event takes a closure which stops the simulation, also this handled by the Simulation API provided by Cooja, and then sends a message to the controller with "STEP_DONE". This message indicates to the controller that the Cooja motes are done executing. This TimeEvent is then scheduled at the time: "current simulation time + step size". When the Cooja simulation reaches that specific simulation time, it will immediately execute the code in the TimeEvent closure.

After scheduling this event, the plugin promptly starts the execution of the simulation, again using the Simulation API. By using the APIs of the Cooja simulation, we can thus make sure that the time executed on the simulation is the same as that of the vehicle simulation, regardless of the processing power of the hardware the simulation runs on, as we work exclusively in simulated time.

Finally this, the plugin starts to listen for connections from the controller, and performs the same procedure again at every step. In the execution flow in subsection 5.1.4, one can see where in the application flow the time handler plugin fits in.

5.1.3 Mobility

The mobility plugin updates the Cooja simulation's positions of all motes, to get correct values for their relative positions which affects, for example, the signal strength.

The plugin is, like the time handler plugin, implemented as a Java plugin for Cooja, and uses the Cooja public Java APIs to control the network simulation environ-

ment.

The plugin is installed in Cooja, and has to be enabled whenever loading a new simulation. It immediately starts to listen for a connection from the controller. When it gets a connection, it immediately stops to listen for incoming connections, and starts a new thread to listen for further messages.

The controller will send the mobility plugin messages over the socket at every new simulation step. This will result in many messages for many scenario simulations, as each step is set to 10 ms in simulation time length. These messages contain the positions of all the simulated vehicles, as two double values, as well as the identifiers of the vehicles, sent as strings. The identifiers of the vehicles are set up in such a way so that the IDs of the vehicles in SUMO correspond to the same IDs of the motes in Cooja.

Many messages will be sent during the simulation of a full scenario, and as we have access to a full Java environment, we implement the data interchange between the controller and the mobility plugin using Protocol Buffers [10], allowing efficient communication, as well as robustness in terms of extensibility. This communication method is not done in the simulated motes, as there we have much stricter constraints due to the emulated platform.

Whenever the mobility plugin receives a message from the controller, it parses it, but it does not save the values. Instead, it immediately updates the motes' positions to the values it received. This is done using the Simulation API available in Cooja, more specifically using the 'invokeSimulationThread' method, where we obtain the 'Mote' we wish to update and change its 'Position' to the provided value. As we do not want to create too many threads, slowing down the simulator, we bundle all updates of all motes for a specific time in this single thread, only creating one thread per update message.

As soon as the positions have been updated, the mobility plugin again starts to listen for new messages from the controller. In the execution flow subsection at 5.1.4, one can see where in the application flow the mobility plugin fits in.

5.1.4 Complete Execution Flow of Simulation Environment

The control flow of the simulation system starts with SUMO executing one simulation step while the Cooja simulation is currently paused. This corresponds to 10 ms of simulated time in the vehicle model. The controller then immediately stops the execution of the SUMO simulation from progressing any further. Now both the vehicle simulator and the network simulator are paused, not executing any application code. We want to keep both simulations in sync, with updated data, letting each execute one step at a time. This is done by continuously executing one step each,

and letting the controller keep track of the progress, notifying the different parts when it is their turn to perform actions. The whole application flow is described below.

1. Cooja simulator is started and loaded up with a configuration for a scenario.

2. Controller is started, provided with a scenario file that corresponds to the Cooja configuration.

3. Controller sets up connections to the mobility plugin, the time handler plugin, and to all serial socket connections tied to the motes in Cooja.

4. Controller starts SUMO, providing it with a valid configuration, generated based on the provided scenario file.

5. Controller connects to the running SUMO instance using TraCI.

6. Controller proceeds to let SUMO execute one simulation step, then pauses the execution of the simulation.

7. Controller retrieves data about the vehicle model from SUMO using TraCI.

8. Controller sends the retrieved positional data to the mobility plugin.

9. Mobility plugin parses the data, and updates the positions of the corresponding Cooja motes in the network simulation to that of the received data.

10. Controller sends out data about all vehicles' properties to all Cooja motes.

    - The data sent to each mote is only data related to the specific vehicle associated with that particular mote.

11. Controller tells time handler plugin that it can execute as many seconds as the SUMO simulation executed in the previous step.

12. Controller waits for time handler to send a response.

13. Time handler plugin receives how long time to execute from the controller. It starts execution on the Cooja motes for the same amount of simulation time. When done, the it stops execution.

    - The motes execute during this time. The Chaos application picks up from where it left off.

    - It reads from the serial line interface for updated data, sent from the controller.

- It executes, or continues to execute, the Chaos data sharing.
- If the Chaos round finishes during execution and the application calculates an action to be taken, it sends that action over the serial line interface to the controller.
- It continues execution, possibly sending many actions until paused by the time handler plugin.

14. Time handler responds to the controller that execution has completed.
15. Controller receives information that execution has been done on motes.
16. Controller checks for possible actions from all the motes.
17. If an action is found, it parses the action, and sends the corresponding message to SUMO using TraCI. For example, if the action is: 'slow down vehicle 5 to speed 0'. This action will be executed the subsequent SUMO simulation step.
18. Controller now proceeds to let SUMO execute one simulation step. The control flow repeats from step 6.
    - This continues until the simulation ends, which results in the controller shutting down. The simulation ends either by timing out, or if the scenario completion conditions are fulfilled.

5.1.5 Communication

The communication has already been described for each related part. The controller communicates over TCP with the SUMO simulation, using the TraCI interface. It polls for data from the simulated motes, to retrieve actions, and sends needed data to the Cooja simulation, and its related plugins.

We communicate new positions to the Cooja simulator every 10 ms, and communicate new vehicle data every 100 ms. The difference between the times are due to performance reasons for the simulation system. We update the positions every 10 ms to get the correct values for the signal strength.

All connections are made locally, using TCP sockets, but the design also allows either the Cooja simulator or the SUMO simulator to also be used on external locations. The protocol used for communication between the controller and SUMO is string-based, and defined by the TraCI interface. We also use a simple, string-based, protocol for communication between the motes and the controller, as we need the protocol to be as simple as possible, due to computational constraints of

the motes.

Communication between all other parts of the Cooja simulation and controller either use simple string-messages for doing "ACKs" or Protocol Buffers for sending more, structured, data. All messages goes through the controller. All other parts are decoupled from each other, and can thus be replaced by only changing code in the controller.

Implementation of Communication on Motes

In the standard case, the serial socket interface in Cooja is implemented so that whenever it gets a whole line of text sent to it, it broadcasts an event to all running motes, with the data of the whole line. The motes can then read and act on this data, without having to do any polling.

This implementation is efficient, but does not work for our use case. Whenever a Chaos round is occurring, it needs complete control over the execution in order to work correctly, due to the timing constraints. The process running the Chaos application is yielded (by calling the PROCESS_YIELD function available in Contiki) until woken up at the end of the Chaos round. However, whenever the serial socket interface broadcasts a global event, this wakes up yielded processes, causing the process to wake up prematurely. This affects the execution of the Chaos round, giving incomplete rounds, and timing errors.

In order to solve this, we extend the serial socket interface to be able to disable all broadcasts. We implement so that the serial socket plugin instead saves the last received message from the controller in a buffer, which we make available for reading using the plugin's public API.

By doing this, instead of waiting for an event to get information sent to the mote by the controller, we read the data from the buffer between every Chaos round. This data will always be the most recent data sent from the controller, containing all needed information for the mote. Hence, all communication and processing with the application and the controller happens strictly between the executions of the Chaos rounds, and never interferes with a Chaos round.

5.2 Chaos Application

This is the main application which is to be evaluated. It is implemented and executed as a Contiki OS process. It uses Chaos for all data sharing between motes. The application also contains the scenario implementation logic, including the algorithm which decides which vehicle's turn it is to drive.

5.2.1 Merge Operator

To share data among all motes, we need to set up Chaos for this scenario accordingly. A vital part of Chaos is the merge operator defined for the processing of the shared data.

For the use case of sharing required data among all motes, what the merge operator needs to do is to merge the data of all motes into a single payload, which then can be interpreted at the end of the round, consisting of information of all vehicles.

We implement a "data-merging" function, where each mote appends the information about its own vehicle to the payload, without interfering with the other motes' data.

This is done by setting up a data structure, containing a set number of bytes, and where each mote have a specific index, and a size of the data it can transmit. It is in this structure that their data will be stored at position $index * size$. The size of the data the motes can transmit is the same for all motes, and defined in the implementation. This index is based on the motes' IDs, which are provided by the simulation. Mote 1 gets assigned index 0, mote n gets assigned index $(n - 1)$, and so on. This way the motes can add information about their specific vehicles, without interfering with the other motes' data. In addition, at the end of the round, a mote knows exactly which mote has provided what corresponding data, based on the index that data can be found at.

We set up this data structure at the start of each Chaos round, when we provide the data to be shared from each mote. The motes all set up an array of *vehicle_data*. Each element in the array is a struct consisting of all needed information that is to be transmitted for each mote.

For our scenario this is:

speed - Unsigned Int8 - The speed of the vehicle as an integer value.

x pos - Unsigned Int8 - The x position of the vehicle in 2D space.

y pos - Unsigned Int8 - The y position of the vehicle in 2D space.

direction - 2 bit - The direction the vehicle has in the intersection. A value between 0 and 3 where 0 is north, 1 is south, 2 is west, and 3 is east.

passed - 1 bit - A bit indicating if the vehicle has passed the intersection or not. Value 0 if the vehicle has not passed and 1 if it has passed.

See figure 5.1 for a graphical representation of the mote data, with the size of each data field. The total size of the struct is 27 bit, so the data, containing useful

information needed by the intersection scenario, which needs to be shared by each mote is 27 bit for each mote.

Speed	X Position	Y Position	Destination	Has Passed
8 bit	8 bit	8 bit	2 bit	1 bit

Figure 5.1: Mote Data - Header showing the mote data added to the shared data structure for each mote. The total size is 27 bit.

When creating the array, it is empty, and the mote only adds the data at its own index, leaving the other parts of the array without any set data. The array has a length corresponding to the number of motes participating in the Chaos round. Thus, if we have set up a number of 6 participating motes for the scenario simulation, we have an array of *vehicle_data*, with a length of 6, and thus with size $6 * 27 = 162bit$.

Before each new Chaos round, all motes set all bits of the array to 0. They then create a vehicle_data struct, and add the struct at index (moteID-1) in the array. Now we have an array of all bits set to 0, except at the index of each mote, which now holds the data that mote will share. This array is the actual payload we wish to share, and what will be processed by the merge operator. The array will be populated with the other vehicles' data during the Chaos round.

This array, together with the needed flags forms the complete payload part of the packet that is transmitted using Chaos. See figure 5.2 for the complete payload structure.

Node 1 Data	Node 2 Data	Node 3 Data	...	Node 25 Data	Node 26 Data	Flags
27 bit	27 bit	27 bit	...	27 bit	27 bit	26 bit

Figure 5.2: Payload structure, including flags, part of each packet transmitted between the motes during a Chaos round. The payload includes all needed data that the vehicles want to share. The maximum number of motes are 26, which is because of frame size limits on the transmitted packet.

When a mote receives a packet, it performs the merge operation. This operation consists of looping through the array of mote data, index by index, and performing a bitwise OR operation at every index with the data the mote already might have in its own copy of the array. As the motes only add information about their own vehicle at the start of the round, we will not get the issue of conflicting data. This is because the data at a specific index should always correspond to a specific vehicle's data for that particular Chaos round, and thus must either be 0 or a specific value. This specific value will always be the same in the local data and the incoming packet's data. By doing a bitwise OR we never alter that data in any way, as a bitwise OR of two identical elements will result in the same element. This will result in that all information the mote does not have will be present in the resulting array, as well as all the information the mote already had also being present in the resulting array. This without ever changing already present data, only adding or preserving already

available data. The mote sets its own version of the array to be the resulting array, and continues the Chaos round, using that as the new payload.

The payload data is assumed to not have been tampered with by an external party, and is assumed to be sent without any data corruption. Usage in real-life vehicles would require securing the protocol and the communication channels.

By doing this, when the Chaos data sharing is complete (i.e. all flags are set to 1), the resulting array in the payload will consist of all motes' data. This data contain the information about all motes' corresponding vehicles. The data is also easily associated with the correct mote, based on the index in the array, and can thus easily be used for the scenario implementation.

5.2.2 Scenario Implementation

Having a way to share all the data between the motes, and having the vehicle simulator communicate with the motes, feeding them data, we have the parts needed to set up our scenario.

The scenario's main design and idea is discussed in the design chapter. The chapter brings up the simple algorithm where one vehicle at a time passes the intersection, until all vehicles have passed.

This is implemented in two parts, with logic both in the controller, as well as the mote. As the mote is severely constrained when it comes to computation power, some aspects are done on the controller. However, this does not in any way affect the results, as the processing power of the motes can be much faster with different hardware. Additionally, the algorithm efficiency is not part of the inherent properties of Chaos, which are what is of main interest, but on the algorithm implementation. All computation related to the processing of shared data, which is tied to Chaos, is done on the mote.

The controller, at each step, checks the completion condition of the scenario. That is, it checks if the scenario is done (all vehicles have passed) or not. If done, it ends the execution, if not, it proceeds. It also checks the positions of the vehicles related the intersection center. If some vehicles are close to the center, the controller sends a message to SUMO to stop the related vehicles to a halt. Additionally, it calculates, based on the vehicle position, if a vehicle has passed the intersection or not. This information is part of the information sent to the motes. This information could just as well be calculated on the motes, as it is not related to the other vehicles, but can be seen as local information, but due to processing power constraints its easier for some of the implementation to be done in the controller.

The controller then waits for actions from the motes, which decide when the vehicle

can pass the intersection or not. When it gets an action from a mote regarding a specific vehicle, it let that vehicle accelerate and drive through the intersection. This then repeats until all vehicles have passed, and execution stops.

On the mote side is where all data sharing happens, and where the logic for deciding if a vehicle, and if so which vehicle, can drive through the intersection.

Using Chaos, at every round all motes communicate their properties. This include, speed, position, direction, and if the vehicle has passed the intersection or not. This is done using the previously defined merge operator. The scenario then aims to, using this data, make sure that all vehicles pass the intersection, without crashing into each other.

The motes continuously initiate Chaos rounds, sharing their data with other motes. This is done as often as possible, but with room to allow needed logging and processing between rounds.

When a Chaos round is ongoing, the mote is not executing any other computations not related to Chaos. Between each round the mote extracts the data it might have got from the previously executed Chaos round. It extracts the data, not just for its own vehicle, but for all the other vehicles participating in the round.

It checks this data for two conditions, and it does this for every vehicle:

- Is the vehicle completely stationary?
- Has the vehicle already passed the intersection?

It checks if at least either of these conditions are true. If either are found true for all other vehicles currently in the simulation, the mote now has information that there currently is no vehicle in, or driving into, the intersection.

The mote then knows that it is safe for a vehicle to enter, but so does all other motes, as they are executing the same logic. To decide which vehicle can drive the mote first checks if it already has passed the intersection or not. If it has not it checks what mote_id it has. It compares this mote_id with the other motes, and checks if all other motes with a lower mote_id have already passed the intersection. If they have, the mote sends an action to the vehicle to drive through the intersection. That is, it only sends an action for its vehicle to drive if it currently has the lowest mote_id which has not already passed the intersection.

We ensure in our configuration of the scenario that the vehicles get added to the vehicle simulation with increasing id numbers, and that a vehicle does not overtake another vehicle in the same lane. By doing this, we ensure that the vehicles with the lowest id numbers will always be able to drive, without a stationary vehicle 'blocking' them from driving.

This action is sent to the controller, which in turn updates the SUMO simulation, making the vehicle drive through the intersection at the next simulation step.

This is then repeated, until all vehicles have passed the intersection, and then scenario ends, and we have successfully managed to drive vehicles safely through an intersection using Chaos for the data sharing.

5.2.3 Flow of Chaos Application

The complete flow of the application running on the motes, including the scenario algorithm and the communication with the controller, is presented below:

1. Application starts, setup with n motes.
2. Initialize array of vehicles_data[n] to all bits set to 0.
3. Disable all broadcasting from serial socket interface.
4. Read and parse message sent from the controller on the serial port.
    - Save data to memory.
5. Initialize Chaos round (blocking).
    - Add data parsed from controller to (mote_id - 1) index in vehicles_data.
    - Array is sent to be used as payload for the Chaos round.
6. Chaos performs a new round
    - Merge operator performs bitwise OR operations on the data.
7. Round done - Application receives a pointer to the array with merged data from all other motes.
8. Check if round completed successfully.
9. If it did, perform scenario algorithm, else go to next step.
    - Send possible actions to controller.
    - Update variables.
10. Go to step 4. Repeat.

5.3 Challenges

Some specific challenges with the implementation mostly has to do with the constraints set up by the underlying platform the Chaos application is implemented on. This due to the implementation of the Chaos related applications is done on the MSP430 platform, on emulated TelosB hardware. This hardware is low-power, low-performance, with a low amount of RAM and processing power[19]. This, in conjunction with Chaos being dependent on the speed of execution between its rounds in order to allow a lower interval between the rounds, made us have to take into account processing time and complexity in many ways.

One example of things needed to be taken into consideration when implementing our application was that only a subset of C is available for the MSP430, using the standard port of the GCC compiler to the MSP platform [20]. For example, handling floating point values is not supported well, where we only have access to functions such as atoi() and atol(), but not their floating point counterparts such as atof(). Overall, the support for floating point numbers is bad, which also put constraints on possible solutions in our implementation, where we have to use integers exclusively.

Chaos, on the implemented platform, also has a limit on the packet size, based on the maximum frame size. This affects how generous we can be with what data to send, and how many motes can participate in a single round.

The number of motes increases both processing time, as well as package size. This gives us strict limits on the number we can have for our implementation. These limits, however, have nothing to do with the core Chaos primitive. If we have better hardware, and use other ways to transmit the data (perhaps using 802.11 for the wireless communication), these constraints can be loosened considerably, which is something that will be discussed further in the evaluation chapter.

6

Evaluation

This chapter presents the evaluation of Chaos in cooperative driving. The evaluation confirms that Chaos successfully enables the scenario discussed in this thesis, and evaluates its performance.

The evaluation presents the viability of Chaos for the scenario. This is done by parametrizing Chaos in the context of the scenario, analyzing in which ways Chaos affects the scenario completion time. Additionally, we analyze what aspects of our results are related to the implementation or other domain-specific factors, and what aspects are because of the properties of Chaos.

The chapter starts with the setup for the evaluation. Following that is the evaluation of the scenario at a macro-level, where we evaluate its performance with regard to the number of vehicles successfully completing the scenario. After that the scenario is broken down into different parts, including vehicle driving time, algorithm execution time, and the overhead of Chaos. Each part is then evaluated.

The evaluation shows us that the scenario can be completed using Chaos. Additionally, we see what constraints Chaos puts on the scenario in relation to constraints related to the implementation or the domain, such as the required time for the vehicles to drive.

For each evaluated aspect, we discuss how the evaluation was performed, the rationale, and the results of the evaluation. Finally, the impact of Chaos is evaluated, where we present the overhead and constraints Chaos puts on the scenario. This includes tying all previously evaluated parts together to form a bigger picture of Chaos in the context of the scenario. The results are then interpreted and evaluated.

Finally, based on the results we discuss the viability of Chaos, and how improvements to the design affect the real-life applicability of Chaos, outside of our simulation environment. We analyze the results and present our thoughts on the limitations related to the thesis.

6.1 Experimental Setup

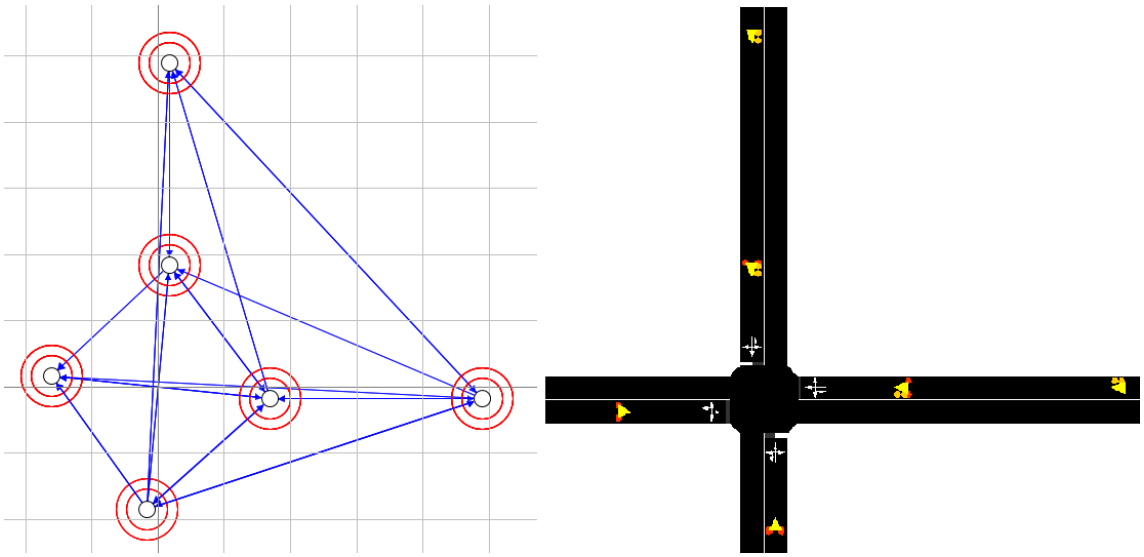
The setup for the evaluation consists of the simulation environment, which we use to evaluate the intersection driving scenario. We set up the scenario, consisting of a 4-way intersection, in SUMO, with each road having two lanes. The participating vehicles can drive in any direction of the intersection.

Figure 6.1 shows a graphical representation of the scenario. It shows six participating vehicles and their positions in the SUMO and Cooja simulators.

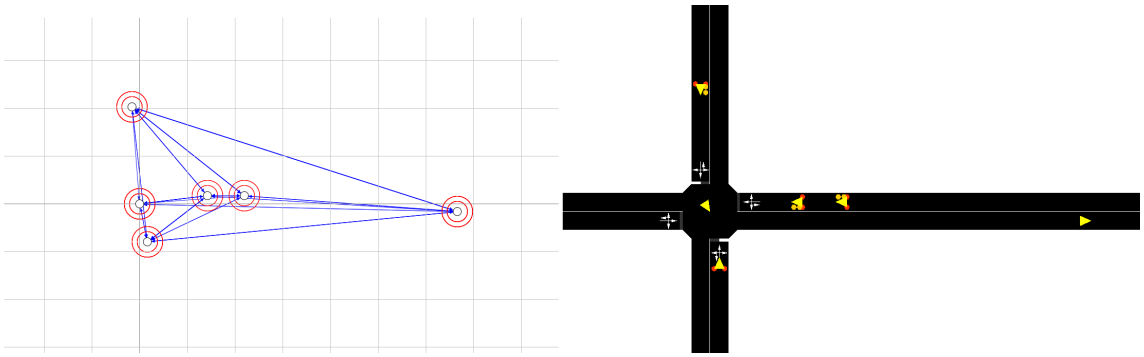
We set up the scenario for a predefined number of vehicles, known at the start of the scenario. All vehicles are always in communication range of each other, and no new vehicles enter the scenario. We thus do not evaluate the notion of leaving and entering vehicles. Instead we focus on the communication in an already set-up network, as the other aspects are out of scope for this thesis.

We evaluate the scenario with respect to if it can be completed safely, and its total completion time. We identify the different aspects which affect the time and evaluate them separately. This includes the time for the vehicles to physically move, which affects the scenario time regardless of the efficiency of the data sharing.

Another aspect is the time needed for a Chaos round to complete, which essentially sets the lower bound on how fast the data at each vehicle can be updated. This aspect has different underlying parts. These include the time to perform the merge operation, the number of slots required to successfully share the data for all vehicles, and the total slot length required. Additionally, we evaluate the aspect of the required algorithm processing time.



(a) The early parts of a scenario with six participating vehicles. All vehicles are currently driving towards the intersection.



(b) When the scenario has progressed so that one of the six vehicles has passed the intersection, and another vehicle is currently in the middle of passing the intersection. The rest of the vehicles are still waiting their turn. The algorithm allows one vehicle to pass at a time, rendering the situation in the image.

Figure 6.1: Two images showing the scenario with six vehicles at two different times of progression. Figure 6.1a shows the early stages of the scenario, where the vehicles are yet to reach the intersection. Figure 6.1b shows the later stages, where the vehicles are currently driving through the intersection.

6.1.1 Evaluation Parameters

We test the implementation of the intersection scenario. The implementation is evaluated with different numbers of vehicles between 2 and 26 vehicles.

These numbers are chosen as 2 is the minimum number of vehicles that make sense to test, in order to have any meaningful communication. 26 is the maximum number of vehicles we can fit in a given packet for our platform, defined by the maximum frame size [14]. We choose a subset of numbers between that, to provide a view of different trends in the data.

Our implementation can be extended to work with more than 26 vehicles. However, that is out of scope for this thesis. Doing so would require using another standard for the wireless communication, such as 802.11, unavailable in our implementation. Another method would be to implement the data sharing over multiple Chaos rounds, and doing a merge of the results. This would increase the time needed to share the data.

The Chaos round interval is set to 1 second. This value fits well with our scenario and is required in our implementation to enable the necessary logging and bookkeeping between the rounds. With more performant hardware this value can be lowered.

6.2 Scenario Evaluation

This section brings up the evaluation of the scenario. Figure 6.1 shows a graphical representation of the scenario inside the simulation system. We evaluate the scenario with regard to completion time. The following sections evaluate the different parts contributing to the completion time of the scenario.

First we justify the evaluation of the scenario, following that we describe the method and parameters used, followed by the measured results.

6.2.1 Rationale

To conclude if Chaos can be used for cooperative driving, we need to simulate an actual scenario and see if we can achieve the desired results. Otherwise, it is hard to draw conclusions and get a clear picture of the impact Chaos might have.

The algorithm for cooperative driving in an intersection in itself is implemented in a relatively naive way, and there are likely more efficient algorithms which can

be developed. However, evaluating a more efficient algorithm is not needed, as we are mainly focused on the communication and data-sharing aspects which Chaos provides. We can evaluate the viability of Chaos using this simple algorithm, as long as it gives the desired result of all vehicles passing the intersection without crashing.

Additionally, by showing a successful execution of the scenario using this algorithm, together with Chaos for the data sharing, we show that the possibilities with better hardware and algorithms can only improve the situation. However, in order to show the viability of Chaos, this requires us to be able to identify which shortcomings of the scenario are dependent on our implementation, as opposed to Chaos itself, which is done in this chapter.

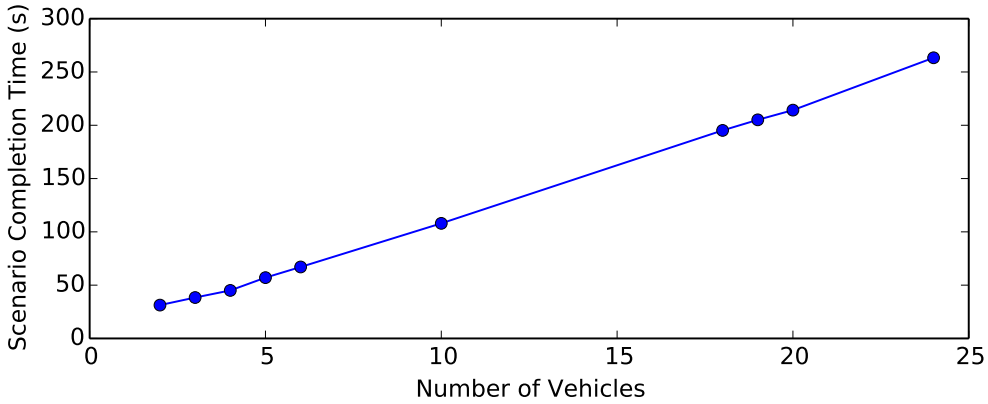
6.2.2 Method

We perform our evaluation on the implemented simulation system. The motes are simulated using Cooja, running our application on an emulated TelosB platform.

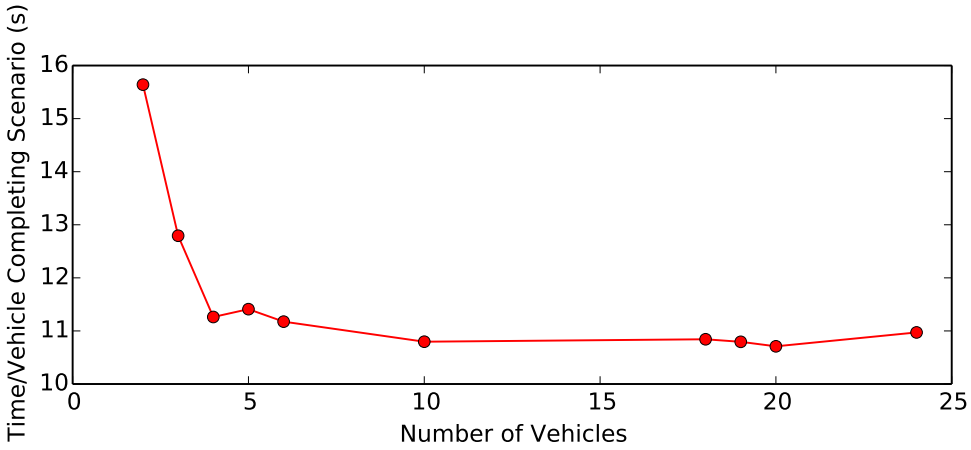
We benchmark the scenario using the running time for the completion of the scenario. That is the time when all vehicles have successfully passed the intersection. This time includes the time for when the first vehicles start to drive towards the intersection, to when the last one leaves it. We test this for different numbers of vehicles. We measure the running time and the flow of vehicles that complete the scenario.

6.2.3 Results

Figure 6.2a shows that the completion time of the scenario increases in a linear fashion, related to the number of vehicles. This is quite natural as the time needed to make every vehicle pass the intersection increases with the number of vehicles.



(a) The total time for running the whole scenario, for different numbers of vehicles. The running time increases linearly with the number of vehicles.



(b) The time per vehicle needed to complete the scenario, for different numbers of vehicles. Approximately one vehicle per eleven seconds successfully completes the scenario except when the number of vehicles are less than four.

Figure 6.2: Two plots related to the running time of the scenario. Figure 6.2a shows the total time to complete the scenario, and Figure 6.2b shows the flow of the vehicles.

The flow of the vehicles, which is the time it takes per vehicle to complete the scenario, is also in line with these results, which is to be expected as the flow is directly related to the total completion time. We see in 6.2b that around one vehicle successfully completes the scenario every eleven seconds, calculated from scenario start.

The data shows an anomaly when there are very few vehicles, but then it stabilizes, indicating no big difference with the increase of vehicles. This is due to the vehicles having to drive to the intersection at the start of the scenario. This affects the vehicle flow greatly when the number of vehicles is low, but is marginalized when the number increases. This is due to the ratio of the vehicles having to wait in line, compared to driving in the intersection, increases with the number of vehicles.

6.3 Vehicle Driving Time

The scenario completion time is broken down into different parts, starting with the time needed for the vehicles to move inside the SUMO simulator. Figure 6.2 shows the total time for the completion of the whole scenario. However, this completion time depends on a lot of different factors, the time for the movement of the vehicles being one.

6.3.1 Rationale

We have shown that the scenario can be completed using Chaos. However, to better evaluate the impact of Chaos in this setting we need to analyze in what ways Chaos affects the scenario completion time.

Many factors of the completion time do not depend on Chaos, such as the speed of the vehicles to physically get to their positions. We evaluate in this section how these properties impact the total completion time. We present the time needed for all vehicles to reach the intersection, as our algorithm waits for all vehicles to be stationary close to the intersection before vehicles can drive through it.

The vehicles physically need to transport themselves through the intersection until another vehicle can drive, in order to avoid a crash. We measure this time, as our algorithm will not proceed during that time, no matter the efficiency of Chaos.

6.3.2 Method

We perform the evaluation on the simulation system. We run a complete scenario and calculate the time needed for all the vehicles in the scenario to reach the intersection and come to a complete halt. This is a single value for the whole scenario, reached once per scenario run.

In the same way we calculate, for every vehicle in the scenario, how long it takes for it to drive through the intersection. It needs to get the action to drive, accelerate,

and drive through the intersection. We take the mean and standard deviation of these values.

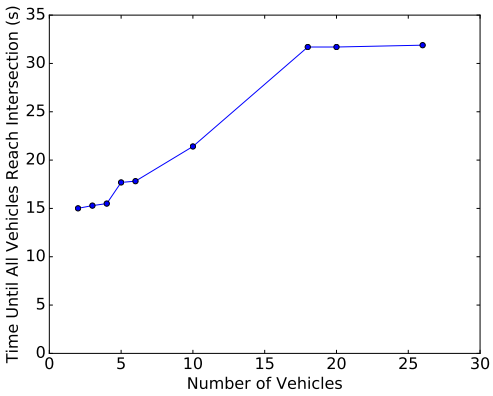
The benchmark uses the elapsed simulation time, provided by the SUMO simulator. We repeat the simulation for different numbers of vehicles.

6.3.3 Results

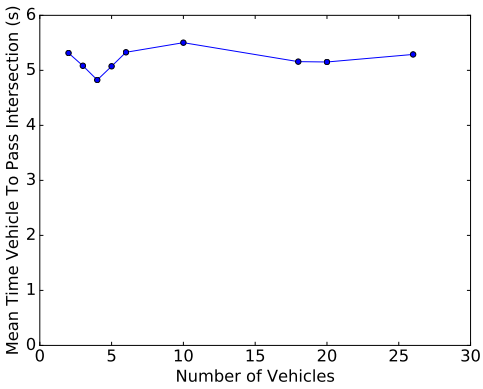
Figure 6.3a shows that a sizable amount of time is taken for the vehicles to reach the intersection, which is when Chaos begins to affect their efficiency, as they now require external data to continue. This time increases with the number of vehicles, as the vehicles are placed at the edge of the simulated roads in SUMO, and with a high number of vehicles they are not added at the same time. This is because there is not enough room before the already added vehicles have started to drive further towards the intersection.

However, the time stabilizes with a higher number of vehicles, in our scenario 18 vehicles. This is to be expected as we already have the impact of vehicles being queued to wait, not giving a substantial difference to the scenarios with fewer vehicles, where most vehicles get added instantly.

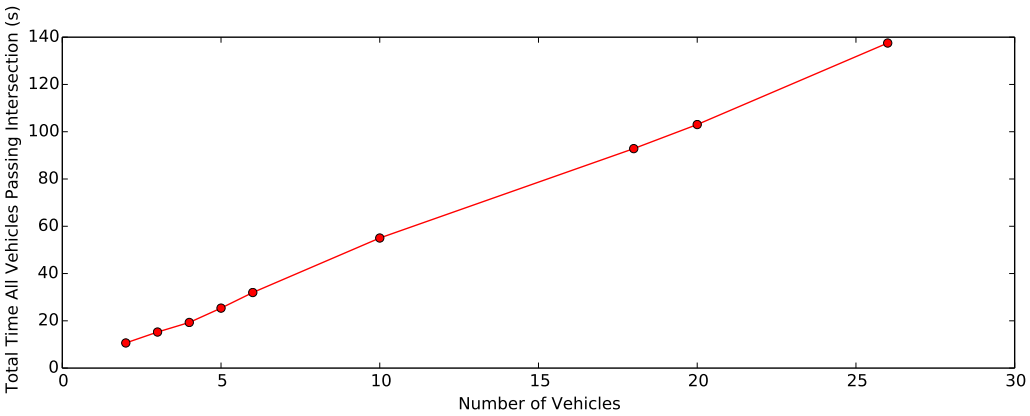
The results show, as seen in figures 6.3b and 6.3c, that a quite substantial amount of time is taken for a vehicle to pass the intersection. This follows quite naturally, as the vehicle needs to accelerate and physically pass the intersection. We see that the time for a vehicle to pass does not depend on the number of vehicles, but is rather constant at different scenario settings with varying numbers of vehicles. This is to be expected, as a vehicle allowed to drive through the intersection will be able to do this without competing vehicles, and it should already be present at the near edge of the intersection when it starts to drive. Thus, the total time for all vehicles to pass the intersection is linear, increasing with the number of vehicles, as the vehicles pass one by one.



(a) The total time, in seconds, for all vehicles in the scenario to reach the intersection and come to a complete halt.



(b) The mean time it takes for a vehicle to pass the intersection, for different numbers of vehicles. This value is relatively constant no matter the number of vehicles, with only smaller deviations.



(c) The total time needed for all vehicles to drive through the intersection, not including the time before or after the actual driving inside the intersection, for the whole scenario. This increases linearly.

Figure 6.3: Three plots showing the time associated with the vehicles to reach and pass the intersection.

6.4 Algorithm

Another contributing factor not related to Chaos itself, as it is performed between rounds, is the implemented algorithm. The running time of the algorithm depends both on its implementation, as well as the available processing power.

To parametrize Chaos, we need to see what impact our algorithm has on the scenario, running on the relatively slow TelosB hardware with a clock speed of 8 MHz [19].

6.4.1 Method

We benchmark the algorithm by running the scenario and measuring the execution time of the algorithm between every Chaos round.

We then measure the mean and standard deviation of the values, to see how much the algorithm affects the vehicles from deciding on an action.

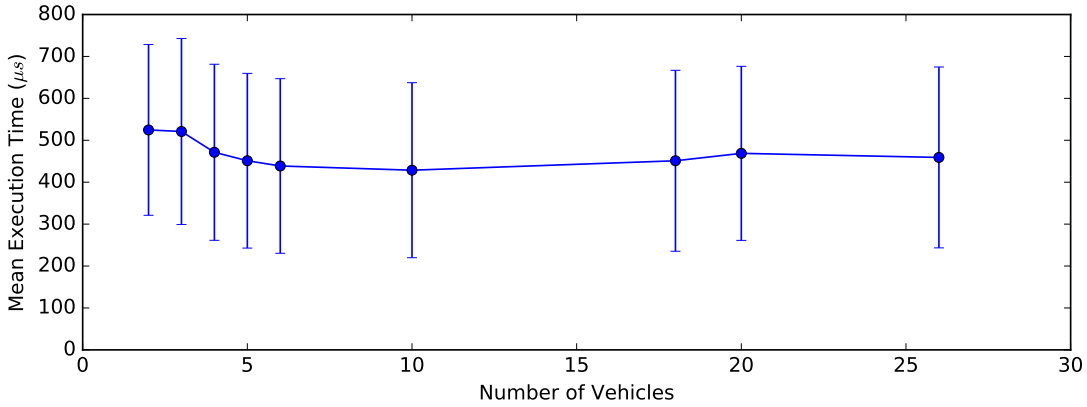
6.4.2 Results

Figure 6.4 shows that the mean running time for the algorithm to execute is around 500 microseconds. This is done between every Chaos round, and the number is in relation to the overall scenario completion time small.

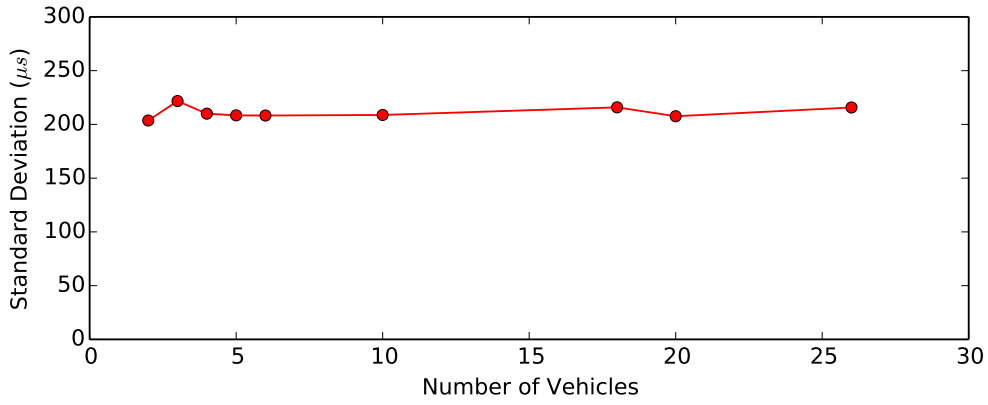
Figure 6.4 shows that the mean running time does not increase with the number of vehicles, even being a little higher at the scenarios with just a few participating vehicles. However, one would expect an increase in running time of the algorithm with an increase of vehicles, as there is more data to loop over.

To explain this, we look at the implementation of the algorithm. The implementation, as described in section 5.2.2, checks if the vehicles are stationary. Most of the time all vehicles are not stationary, but at least one vehicle is either driving towards, or inside, the intersection. This is a useful property, as it contributes to the progression of the scenario. However, when this is the case, the algorithm exits almost immediately. This is because the algorithm only makes decisions when the intersection is clear, and all vehicles which have not passed the intersection are stationary. This makes the running time higher for scenarios with a larger number of vehicles, as it might take a few more loops to find this vehicle.

However, this is shadowed by when the condition does hold true, and a vehicle performs an action. The implemented algorithm is very efficient when seen to the number of instructions it requires for each loop, except when performing an action, which involves printing to the serial port. This I/O operation is orders of magnitude slower, which can be seen in Figure 6.4b, with a quite substantial standard deviation. This deviation is related to the times I/O is needed. Most of the time the algorithm executes rather quickly, but with spikes in execution time whenever the algorithm finds an action. As the complexity of doing an action is constant, the mean execution time should increase with the number of vehicles, but a larger sample size is needed to show that in practice.



(a) The mean running time for the algorithm for each number of vehicles. It is rather constant over the different number of vehicles. The vertical lines show the standard deviation, also shown in 6.4b



(b) The standard deviation of the algorithm running time for each number of vehicles. This deviation is relatively constant over the different numbers of vehicles.

Figure 6.4: Two plots showing the characteristics of the benchmarked algorithm. Figure 6.4a shows the mean running time for the algorithm for each number of vehicles. Figure 6.4b shows the standard deviation for each number of vehicles.

The algorithm needs to perform less loops with a low number of vehicles compared to a higher number of vehicles. However, an action will be sent for a vehicle to drive at least once for each participating vehicle in the scenario. This makes the ratio of times the algorithm performs an action, compared to where it prematurely exits, higher with a lower number of vehicles, increasing the mean execution time.

In essence: the running time should increase linearly with the number of vehicles, but this increase in negligible in our implementation, as most of the time the algorithm will exit without looping over all vehicles.

6.5 Chaos Evaluation

Having shown how the physical movement of the vehicles and the algorithm affect the scenario, we focus on Chaos and its different properties. We identify the main aspect that affects the performance of the scenario, namely the **Chaos round completion time**. A Chaos round consists of different parts. This includes our implemented merge operator, as well as the number of slots needed for each round and the total length of these slots. Combined we can find the limit on how long a single round takes for our scenario. This will be the "bottleneck" Chaos puts on the scenario. Studying this time, which parts that affect it, and how they relate to our implementation, is important to see how Chaos affects the scenario.

6.5.1 Chaos Merge Operator

This section presents the evaluation of the Chaos merge operator for the communicating vehicles. We justify the evaluation, and after that the method and parameters used are presented, followed by the measured results.

Rationale

The merge operator is an essential part of Chaos. Our implementation defines a merge operator to share all necessary data between vehicles. This operator is called not just at every round, but at every slot inside the round, affecting the lower bound of the slot length. Thus the performance of the merge operator affects the performance of our implementation, and we are interested in how much.

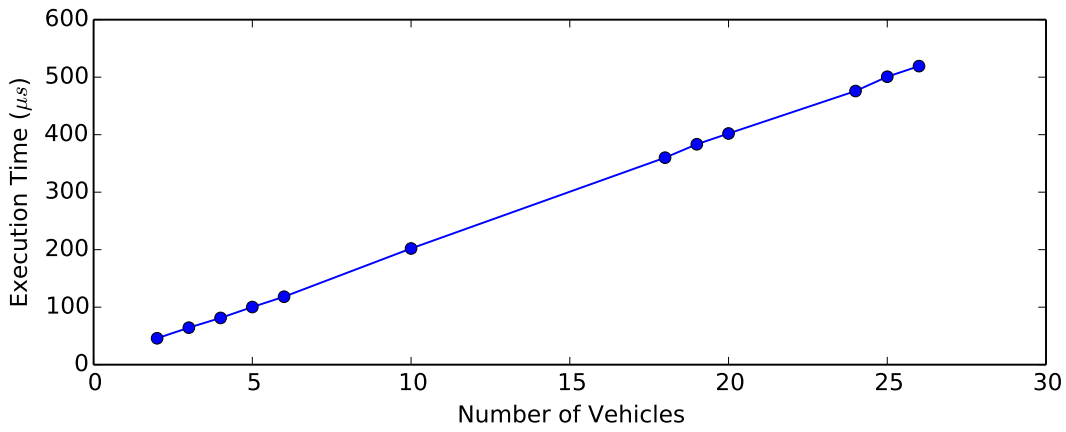


Figure 6.5: The running time of the merge operator for different number of vehicles. We see a linear increase in execution time related to the number of vehicles.

Method

We perform the evaluation on the simulation system. The motes are simulated using Cooja, emulating a TelosB system [19]. The merge operator is benchmarked using the running time as the metric. We measure the time needed for performing the merge operation at each mote, and at each slot. We do not measure other parts done inside the slot execution time, and focus on our implementation of the merge operator, as that is what is specific to the vehicle communication use case. We measure the merge operator running on our intersection scenario, with different numbers of vehicles.

Results

Figure 6.5 shows that the execution time of the merge operator increases in a linear fashion, with the increasing number of vehicles. This makes it a simple operator (computation-wise) for a lower number of vehicles, but with a higher number of vehicles the execution time increases, albeit at a linear complexity of $O(n)$.

The results from the merge operator are the same for every round in the scenario with the same amount of vehicles. This is due to the computation always being the same, with no changing parts between rounds, thus making it easy to see the linear increase with the number of vehicles.

6.5.2 Chaos Round Time

The total time needed for a complete Chaos round is of interest. The interval in which the vehicles can update their data depends on how fast a Chaos round can complete, as well as the required time to handle the algorithm and other processing between rounds.

We evaluate the Chaos round time for different numbers of vehicles in our scenario.

Number of Slots

The time for a Chaos round depends on the number of slots in each round. If the round have to execute a large number of slots, this will also increase the total time of the Chaos round. By running the scenario, and logging the number of slots required for each round until scenario completion, we calculate the mean number of slots required for a single round in our scenario.

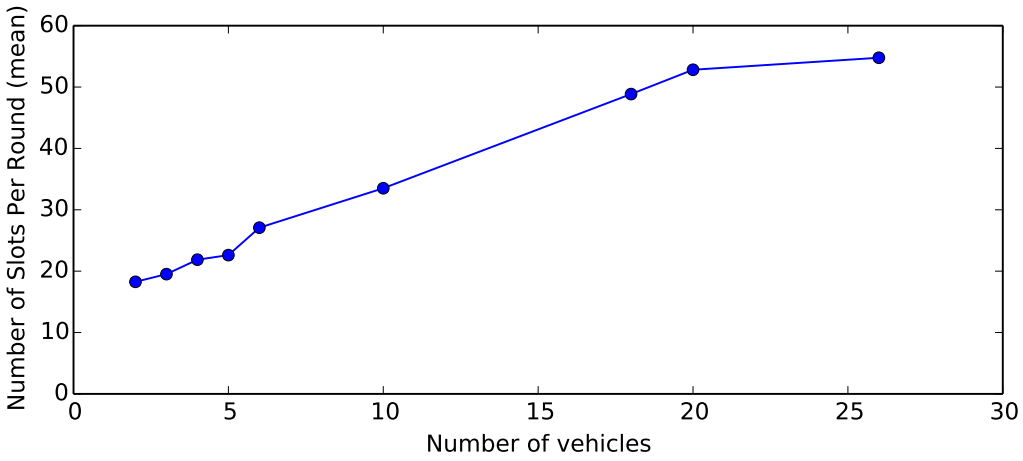


Figure 6.6: The mean the number of slots needed for a round to complete for different numbers of vehicles.

Figure 6.6 shows that the number of slots needed increases with the number of vehicles. This is understandable as with a larger number to share data between, more slots are required in order to successfully make all motes share the data. This depends both on the number of motes, and the positions of the motes in relation to each other. For our particular scenario, the number of slots needed increases with the number of vehicles, in a relatively linear fashion.

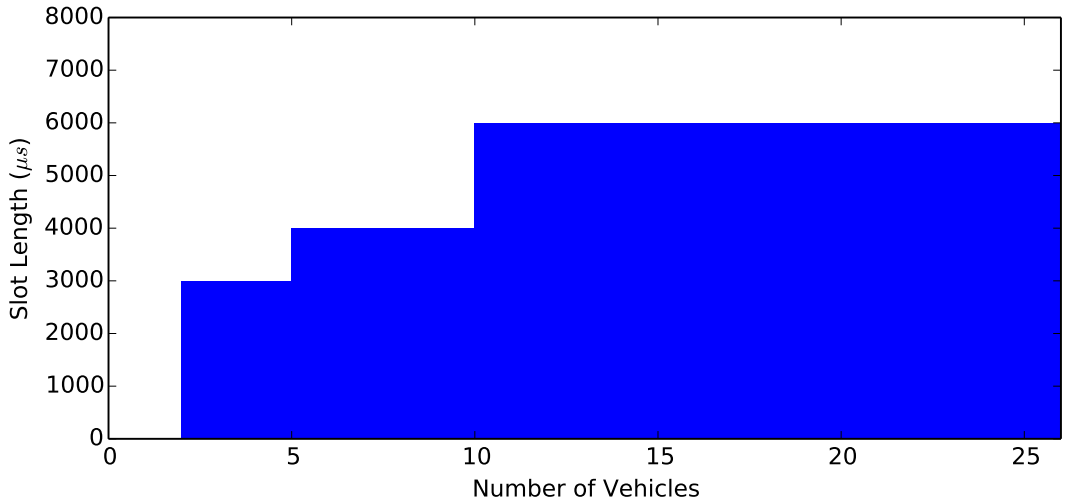


Figure 6.7: The needed slot length for different number of vehicles. We see here that the required length increases with more vehicles.

Slot Length

In addition to the number of slots executed in a Chaos round, the length of each slot is of interest. We have already evaluated parts of the execution time needed inside a slot with the merge operator, effectively already having a lower bound. This section presents a more detailed evaluation of the slot length, and how it scales with the number of vehicles.

The slot length is defined at compile time, and the numbers for the slot lengths were found by testing the lowest possible number for a scenario and increasing the number whenever a reliability drop occurred.

We see in Figure 6.7 that the slot length required increases with the number of vehicles. This is because more processing power is required to process the data (similar to the merge operator in Figure 6.5). Additionally, more time is needed to transmit and receive the data. With more processing power this length can be decreased. However, there is still a requirement on the time it takes to transmit the data which depends on the radio medium.

Final Result

Combining the values from the slot length and the number of slots gives the average time needed for a round to complete in our scenario.

Figure 6.8 shows that the round length increases with the number of vehicles, which

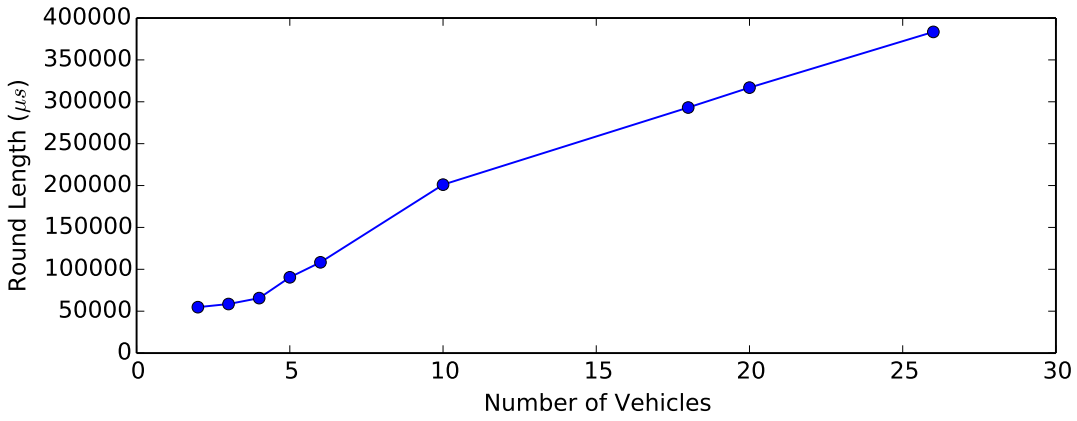


Figure 6.8: The mean round length for a single round for different number of vehicles. We see here an increase in round length with the number of vehicles.

is to be expected as both the number of slots and the slot length increase with the number of vehicles. It shows that with 26 vehicles we have a total slot length of around 0.4 seconds. Thus, if all processing outside of the Chaos round were negligible, a Chaos interval of 0.5 seconds instead of one second could be feasible for our scenario. This is even more evident with a lower number of vehicles, where we have a round length of around 0.05 seconds.

All in all, we see that we can obtain, and process, data from nearby vehicles in a matter of a few hundred milliseconds even for a larger number of vehicles. This is good enough for our scenario and may be enough for many other scenarios as well.

6.6 Impact of Chaos Results

This section presents the results related to Chaos viewed against the scenario.

We have evaluated the scenario with regard to its completion time and the performance of Chaos. To find the impact Chaos has on the scenario's performance, we first analyze how our solution performs, and then compare it to the best theoretically possible solution, as well as to the solution of using Chaos without any processing constraints.

6.6.1 Method

The evaluation is done by taking the elapsed time needed for all vehicles to reach the intersection added to the time needed for all vehicles to pass through the intersection. By doing so, we get the optimal scenario; where no vehicle ever waits for data. They drive to the intersection, and then drive through it, one by one, without waiting for any data. This is the best possible outcome for our algorithm, where it has access to the data instantaneously, and processes it immediately.

From this data we add the minimum required overhead caused by Chaos, where we assume a minimum slot length, unlimited processing power, and no overhead of neither our simulation system, implementation, nor hardware.

We assume a Chaos interval set to the same as the Chaos round length, as that is the lowest possible interval. We assume the same number of slots for each round in the scenario, using the data in Figure 6.6. This is done because the number of slots is not bound by processing time. However, we assume a much lower slot length.

When assuming a value for the slot length, we need to look into other factors than processing speed, as the slot length is not entirely bound by processing speed. We assume instant communication with the Microcontroller Unit (MCU) and the radio. For the TelosB platform we have a 250 kbit/s data rate radio [19]. The slot length will thus, when assuming optimal processing time, be dependent on the packet size related to this bit-rate. The packet and payload size are discussed in chapter 5. We can thus calculate the slot length by taking the time to transmit the whole packet, assuming no delay to get the packet to and from the radio.

From this data we can calculate our new Chaos round length, by having the number of slots and the slot length.

The minimum overhead added by Chaos to the scenario completion time is the Chaos round length, as well as the mean waiting time between rounds, which is half of the Chaos interval. However, the optimal Chaos interval is equal to the round length, so the overhead is 1.5 times the round length. This time is then added once for each vehicle, as only one vehicle can pass the intersection at a time. We add the resulting overhead to the scenario with no delay at all. So if we have n vehicles, we add the overhead of

$$n * (x + y/2) == n * (x + (x/2))$$

Where x = Chaos total round length, y = Chaos interval between rounds

We do this once for every vehicle, as that is the minimum amount the vehicles need to make the decision if they can drive or not. This way we only depend on how long a round takes for our scenario, and not the processing speed, nor the communication between our simulation systems or eventual overhead of SUMO.

6.6.2 Results

Figure 6.9 shows the relative overhead of the data sharing. An overhead of zero means instant data sharing and no implementation overhead at all.

Figure 6.9 shows the overhead of the best possible implementation using Chaos for the data sharing. I.e., an implementation using Chaos that is only bound by network transmission speed, not processing speed, nor any other implementation constraints imposed by the system. We get a minimum bound of the best possible way Chaos performs for our scenario. This includes both the least amount of time needed to complete a Chaos round, but also properties concerning how long a vehicle waits before it initiates a round. We assume only the overhead of network transmission, so we get a lower Chaos interval, slot length, and no processing delays. Additionally, no overhead induced by the simulation system is induced, and the motes always have their own updated local data at all times. This is the best possible lower bound using Chaos.

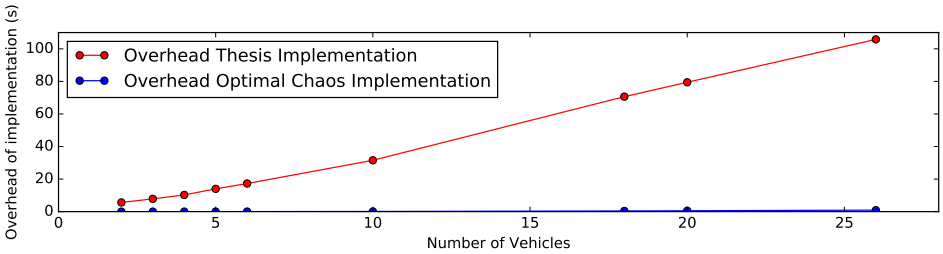


Figure 6.9: The relative overhead of the data sharing for our implementation compared to the best possible implementation using Chaos. The implementation with no processing overhead, or other implementation constraints, adds a very low overhead compared to not performing any data sharing at all. To see the difference between no overhead at all see Figure 6.10 which shows the differences in time in more detail. The thesis implementation adds a large overhead, due to processing and implementation constraints.

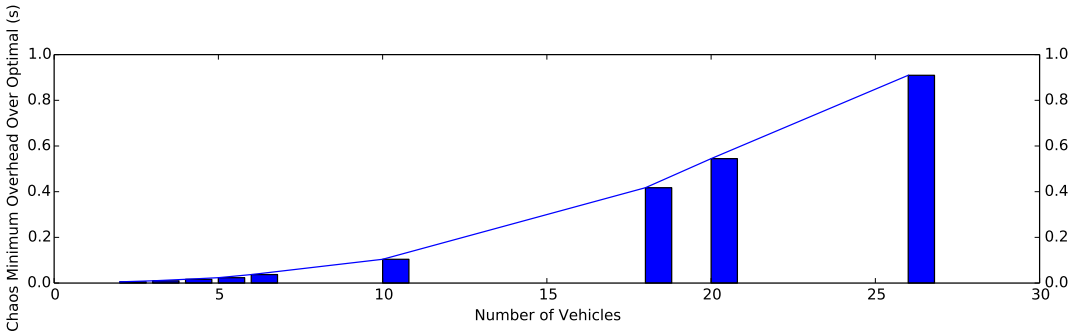


Figure 6.10: The minimum overhead Chaos will add to the scenario of instant data sharing. It shows that the overhead increases with the number of vehicles. This is natural as the number of vehicles increases both the payload size needed to be processed, the number of slots required for a full round, as well as the number of times a decision has to be taken by the vehicles.

Figure 6.9 shows our implementation compared with the best possible implementation. We note that the best implementation using Chaos is faster than the results for this thesis's implementation, and adds a low overhead to the optimal implementation of no sharing needed at all. Figure 6.9 shows that for a low number of vehicles the difference is rather small in relation to the complete scenario time. The majority of the time spent is related to the vehicles physically driving, spending a low amount of time needing to communicate. They only need to make the decision to drive through the intersection or not relatively few times compared to the scenario with a larger number of vehicles.

We also note that the optimal time, and the time using Chaos without any processing overhead, are very close to each other. Figure 6.10 shows the difference between the two in more detail. This indicates that the relative overhead of Chaos, with our assumptions regarding processing time, is small compared to a solution with instantaneous sharing of data, with less than a second longer completion time for the scenario with a running time measured in minutes. Figure 6.10 confirms this, with a difference in completion times in sub-second levels.

6.7 Discussion of Results

We find in our evaluation that we can successfully complete a cooperative driving scenario of an intersection with different numbers of vehicles. We analyze the aspects of the scenario, and which parts are related to the implementation, and which are related to Chaos. This section discusses these results.

6.7.1 Viability of Chaos

This section describes the viability of Chaos related to the scenario. It summarizes the major findings of the evaluation of Chaos for cooperative driving.

Intersection Scenario

For the particular scenario of the intersection, the vehicles can successfully cooperate to drive through the intersection, using Chaos to communicate. The performance of the scenario has opportunities for enhancement, mostly by using a smarter algorithm, as each vehicle that needs to stop at the intersection slows down the speed. This overhead dwarfs the time needed for the communication. Implementing a more powerful algorithm, perhaps allowing the vehicles to pass the intersection without ever needing to come to a stop, will considerably speed up the scenario.

However, implementing a more complex scenario might put higher constraints on the communication. Figure 6.8 shows the round lengths, which correlates to how fast a vehicle can get new shared data. For few vehicles this round length is small; less than 100 milliseconds, and less than 400 milliseconds for up to 26 vehicles. It is not infeasible that improved algorithms can be built using this update interval, with good performance.

Optimal Chaos Implementation

Already for our implementation, we find it viable to use Chaos for cooperation between the vehicles. We show that Chaos, when having the best possible hardware and implementation, allows data sharing between vehicles even faster, as the data sharing is not bound by the processing power. We can see our implementation as one of the more inefficient scenarios. By concluding that our inefficient scenario is good enough in turn allows us to conclude that more efficiently implemented scenarios are even more viable.

We show from our data that Chaos works well to be used in the scenario of intersection driving, both for few to a lot of vehicles.

6.7.2 Analysing the Results

From the performance measurements, evaluation of the different contributing factors, and parametrization of Chaos brought up in the previous sections, we make some observations:

- The scenario is completed successfully without any crashes between vehicles.
- The solution, in almost all aspects, scales linearly with the total number of participating vehicles.
- The time needed for a Chaos round, and subsequently to share the data between the vehicles, is in the size of a few hundred milliseconds for around 25 vehicles, and lower with less vehicles.
- The reliability of the data sharing in the scenario is 100%, with no incomplete rounds, as we chose configuration parameters that always resulted in successful rounds.
- By parametrizing Chaos, assuming the most efficient implementation and most powerful hardware, we get a solution very close to the optimal solution; which is no delay at all for the data sharing. The difference for our scenario is in subsecond levels for 26 vehicles.
- The total overhead of the data sharing, in relation to the constraints of the scenario and its implementation, is low. A Chaos round takes approximately 400 milliseconds for 26 vehicles, with a lower number for less vehicles. This gives a lower bound for the data sharing at just a few percent compared to the time to physically move the vehicles.

These results follow the promised properties of Chaos, which works well even with large node counts and densities, and has high reliability and efficiency [15].

6.7.3 Limitations

This thesis shows the use of Chaos for the proposed scenario, showing a successful completion of the scenario. However, there are parts which might be of interest with regard to Chaos and its role in cooperative driving and inter-vehicle communication that is not discussed.

The first is that of a more thorough evaluation of the effects of mobility on Chaos. We evaluate one part of this aspect, as we update the positions of all simulated motes to their respective vehicles' positions. Hence, the motes will have the correct position in relation to each other during all communication. We include the effect of the signal strength related to the vehicles' movements, as they change positions.

However, we do not evaluate the impact of the actual movement in relation to the performance of the communication. With moving objects communicating with each other, the effects of multipath propagation and fading occur, which have an impact on the communication. This is a phenomenon that results in the radio signals

reaching the receiving radio medium by more than one path. This affects both constructive and destructive interference, which is used for the completion phase for Chaos, as well as phase shifting of the signal [15]. Movement of nodes adds to this effect, giving a phase change over a small amount of time [25]. The speed of the communicating vehicles can thus have an impact on the communication, not just related to their position but their actual speed in relation to each other [25][1].

Due to limitations in our simulation environment, these aspects are not present in our evaluation. There is no support in Cooja for evaluating the effects that the mobility of motes has on the communication. Our evaluated scenario does not impose any significant movement speeds, especially when close to the intersection. Thus, we assume the impact of movement for this particular scenario to be quite negligible, but only experimental results can prove this fully.

Another limitation is that our evaluation of Chaos is done in a very controlled environment. We only evaluate the direct communication using Chaos, but we do not include the aspects of vehicles joining and leaving the network, instead recommending the use of other solutions to set up the VANET for the scenario. This can perhaps be done using Chaos, with for example all nodes sending out a broadcast message whenever they get close to the intersection, that they want to initiate a Chaos round, and where they assume all nearby vehicles will answer that message. They then do this between every round, to only have vehicles that want to pass the intersection to be part of the communication. Another way is using a centralized point of communication, perhaps installed near every intersection, which the vehicles can connect to and which then sets up the VANET. This is out of scope for this thesis, but can be of interest (see Future Work at section 7.1). Additionally, vehicles in our scenario are always in range of each other, as a vehicle driving out of range should leave the scenario, since it has successfully passed the intersection.

In the same way our scenario is restricted to only evaluating the effects on a vehicle intersection. While we from this get insight into the capabilities of Chaos of interest also for other scenarios, a practical test of these scenarios is needed for the best results. Additionally, our scenario is restricted to 26 vehicles, as that is the maximum we can fit into a single packet for the platform tested on. However, there is nothing in our work that indicates that it is not possible to have more vehicles in a single scenario, in fact we see that we only have a linear increase in complexity for most parameters. This is possible by either increasing the packet size or performing multiple rounds before aggregating the results. Future work might find it interesting to study the impact of this in practice.

7

Future Work & Conclusion

This chapters brings up possible future work related to the thesis, with suitable areas for further research. The chapter ends with the conclusion of the thesis.

7.1 Future Work

The thesis focuses on implementing a scenario for cooperative driving, using Chaos for the data sharing. By doing this, information about Chaos and its use in the domain of inter-vehicle communication has been found. However, as can be seen in the limitations section of the thesis, there are things which can be further explored regarding Chaos being a viable alternative.

The most prominent course of action would be to evaluate the effects of mobility on Chaos. This includes not just having the correct positions of the vehicles at all times, but also factoring in the actual movement of all vehicles.

Additional scenarios, preferably with different constraints than the scenario implemented in this thesis, would also be a useful area of further research. This way the results of this thesis can be compared to results for different scenarios, and it can be evaluated if Chaos is feasible also in other scenario contexts.

To get the most accurate results, it would be preferred to test the results of this thesis in a real-world setting, using actual physical hardware, mounted on vehicles. This eliminates possible shortcomings of the simulation system, and is a logical next step in evaluating Chaos.

Different techniques to enable leaving and joining the VANET in which Chaos is used within is also of use, to allow a more fully featured vehicle communication system.

7.2 Conclusion

It has been shown that the use of Chaos in the domain of cooperative driving is viable, to the point that it can be utilized for a scenario of vehicles driving through an intersection. The vehicles can safely complete the scenario, deciding their actions based on data shared by other vehicles. We show that the scenario is fully functional using Chaos. The data sharing is efficient, with round times in a few hundred milliseconds for 26 vehicles, and increasing linearly. By parametrizing Chaos in the context of the scenario, we see that most parts related to slower performance of the data sharing is due to our implementation's constraints as opposed to Chaos itself.

Chaos is implemented as the communication primitive, with the scenario specific algorithm being applied between Chaos rounds. The scenario requires processing of all vehicles' data, aggregating the data and distributing it to all vehicles. These are properties which Chaos handles well, and which was found to work well for the implemented scenario.

The characteristics of the data sharing, with its ad hoc nature and good efficiency, make it interesting for even more scenarios than the one implemented. There are no clear obstacles found in this thesis that prevent Chaos from being adapted to more scenarios, using the same underlying data sharing techniques.

Chaos fits the domain well, with high reliability, good efficiency, and scalability. Its distributed nature makes it a viable technology for use cases where entities need to communicate in an ad hoc manner, sharing the data in an all-to-all fashion.

Chaos does not solve the problem of vehicle communication, and cooperative driving, by itself. It depends on other technologies, for example to know which vehicles should participate in a specific scenario, and to handle vehicles leaving and joining. Chaos is also bound by the maximum size of a single packet for its data sharing. This requires efficient use of which data to share, as if the data payload size overtakes the packet size, the efficiency of Chaos is affected.

In summary, we found that Chaos is a viable solution for cooperative driving for our implemented scenario. When used with other technologies for establishing the VANET, Chaos provides high reliability and efficiency, even for a larger number of participating vehicles.

Bibliography

[1] Sabina Baraković, Suad Kasapović, and Jasmina Baraković. "Comparison of MANET routing protocols in different traffic and mobility models". In: *Telfor Journal* 2.1 (2010), pp. 8–12.

[2] Michael Behrisch et al. "SUMO–Simulation of Urban MObility". In: *The Third International Conference on Advances in System Simulation (SIMUL 2011), Barcelona, Spain*. 2011.

[3] Gabriel R de Campos, Paolo Falcone, and Jonas Sjoberg. "Autonomous cooperative driving: a velocity-based negotiation approach for intersection crossing". In: *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on*. IEEE. 2013, pp. 1456–1461.

[4] Gustavo Carneiro. "NS-3: Network Simulator 3". In: *UTM Lab Meeting April*. Vol. 20. 2010.

[5] ContikiOS-Contributors. *Cooja Source*. Github. 2016. URL: `https://github.com/contiki-os/contiki/tree/master/tools/cooja/java/org/contikios/cooja` (visited on 01/11/2016).

[6] Manjunath Doddavenkatappa, Mun Choon Chan, and Ben Leong. "Splash: Fast Data Dissemination with Constructive Interference in Wireless Sensor Networks." In: *NSDI*. 2013, pp. 269–282.

[7] Richard Dowling, Alexander Skabardonis, and Vassili Alexiadis. *Traffic analysis toolbox volume III: guidelines for applying traffic microsimulation modeling software*. Technical Report. U.S. Department of Transportation, 2004, p. 96.

[8] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. "Contiki-a lightweight and flexible operating system for tiny networked sensors". In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE. 2004, pp. 455–462.

[9] Federico Ferrari et al. "Efficient network flooding and time synchronization with Glossy". In: *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE. 2011, pp. 73–84.

[10] Google. *Protocol Buffers*. Google. 2016. URL: `https://developers.google.com/protocol-buffers/` (visited on 01/08/2016).

[11] Javier Gozalvez et al. "iTETRIS: the framework for large-scale research on the impact of cooperative wireless vehicular communications systems in traffic efficiency". In: *Information and Communications Technologies (ICT-MobileSubmit 2009)* (2009).

[12] Erico Guizzo. “How google’s self-driving car works”. In: *IEEE Spectrum Online, October* 18 (2011).

[13] Katsuya Hasebe, Akihiro Nakayama, and Yūki Sugiyama. “Dynamical model of a cooperative driving system for freeway traffic”. In: *Physical review E* 68.2 (2003), p. 026102.

[14] IEEE. *IEEE 802: LOCAL AND METROPOLITAN AREA NETWORK STANDARDS*. Institute of Electrical and Electronics Engineers. 2011. URL: `http://standards.ieee.org/getieee802/download/802.15.4-2011.pdf` (visited on 02/05/2016).

[15] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. “Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale”. In: *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2013, p. 1.

[16] Fan Li and Yu Wang. “Routing in vehicular ad hoc networks: A survey”. In: *Vehicular Technology Magazine, IEEE* 2.2 (2007), pp. 12–22.

[17] Li Li and Fei-Yue Wang. “Cooperative driving at blind crossings using intervehicle communication”. In: *Vehicular Technology, IEEE Transactions on* 55.6 (2006), pp. 1712–1724.

[18] S McCanne and S Floyd. “ns Network simulator–Version 2”. In: *URL: http://www.isi. edu/nsnam/ns* (1998).

[19] MEMSIC. *TelosB Mote Platform*. MEMSIC Inc. 2015. URL: `http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf` (visited on 01/22/2016).

[20] MSPGCC-Authors. *The standard library functions*. GNU. 2015. URL: `http://mspgcc.sourceforge.net/manual/x1109.html` (visited on 01/21/2016).

[21] Vinod Namboodiri, Manish Agarwal, and Lixin Gao. “A study on the feasibility of mobile gateways for vehicular ad-hoc networks”. In: *Proceedings of the 1st ACM international workshop on Vehicular ad hoc networks*. ACM. 2004, pp. 66–75.

[22] Fredrik Österlind. “A sensor network simulator for the Contiki OS”. In: *SICS Research Report* (2006).

[23] Charles Perkins, Elizabeth Belding-Royer, and Samir Das. *Ad hoc on-demand distance vector (AODV) routing*. Tech. rep. 2003.

[24] Michal Piorkowski et al. “TraNS: realistic joint traffic and network simulator for VANETs”. In: *ACM SIGMOBILE Mobile Computing and Communications Review* 12.1 (2008), pp. 31–33.

[25] Theodore S Rappaport et al. *Wireless communications: principles and practice*. Vol. 2. Prentice Hall PTR New Jersey, 1996.

[26] Michele Rondinone et al. “ITETRIS: a modular simulation platform for the large scale evaluation of cooperative ITS applications”. In: *Simulation Modelling Practice and Theory* 34 (2013), pp. 99–125.

[27] Aviva Rutkin. *Autonomous truck cleared to drive on US roads for the first time*. New Scientist magazine. 2015. URL: `https://www.newscientist.com/article/dn27485-autonomous-truck-cleared-to-drive-on-us-roads-for-the-first-time/` (visited on 11/02/2015).

[28] Tesla-Motors-Team. *Your Autopilot has arrived*. Tesla Motors. 2015. URL: http://www.teslamotors.com/blog/your-autopilot-has-arrived (visited on 11/02/2015).

[29] Axel Wegener et al. "TraCI: an interface for coupling road traffic and network simulators". In: *Proceedings of the 11th communications and networking simulation symposium*. ACM. 2008, pp. 155–163.