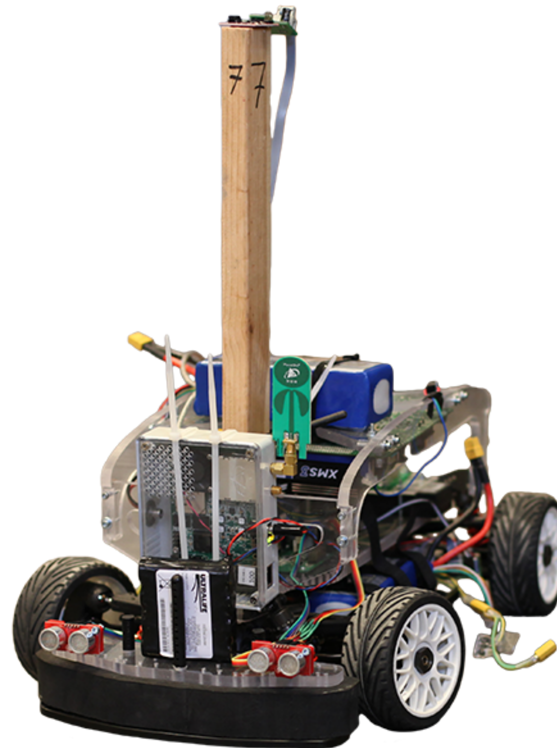
CHALMERS



Gulliver - Vidareutveckling av ett system för testning av autonoma bilar

Kandidatarbete inom Data- och Informationsteknik

ERIC AHLBERG
JOAKIM DANIELSSON
MARCUS ISAKSSON
SEBASTIAN IVARSSON
AXEL JOHNSSON

Chalmers tekniska högskola
Institutionen för Data- och Informationsteknik
Göteborg, Sverige, Juni 2014

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Gulliver - Vidareutveckling av ett system för testning av autonoma bilar

Kandidatarbete DATX02-14-26

Eric Ahlberg,
Joakim Danielsson,
Markus Isaksson,
Sebastian Ivarsson,
Axel Johnsson

© Eric Ahlberg, Joakim Danielsson, Marcus Isaksson
Sebastian Ivarsson och Axel Johnsson, Juni 2014

Examinator: Arne Linde

Chalmers tekniska högskola
Institutionen för Data- och Informationsteknik
SE-412 96 Göteborg
Sweden
Telefon + 46 (0)31-772 1000

Institutionen för Data- och Informationsteknik
Göteborg, Juni 2014

Abstract

With an increasing population and new transportation habits, many problems regarding the traffic system will become more prominent. These include road congestion, traffic accidents and emission of carbon dioxide. Many of these problems could be reduced by letting computers drive for us. Developing driverless cars is however a very expensive process and there are many difficulties. At Chalmers there is a project named Gulliver, where a simulated computer model is combined with small physical cars to achieve a good combination of realistic and cost effective tests.

This bachelor thesis further develops the Gulliver project, and the goal is to improve and develop the test-bed as well as to evaluate new technologies that can be relevant in the future of the project. The thesis has focused on three subprojects:

- Improving the localization system of the cars by integrating the software GulliView into the Gulliver graphical tool. GulliView enables the use of an independent source in the form of a web camera that uses image recognition to localize the cars.
- Improving the graphical tool that is used when testing in order to speed up and simplify the testing procedure.
- Evaluating the algorithms UDP Broadcast and Paxos in terms of performance, especially the time it takes to reach consensus, for direct communication between the cars.

Keywords: Gulliver, autonomous miniature car, test-bed, driverless cars, Paxos

Sammandrag

Med ökad befolkning och nya transportvanor kommer många trafikproblem såsom köbildningar, olyckor och koldioxidutsläpp bli allt mer påtagliga. Flera av dessa problem kan reduceras om man skulle låta datorer köra bilarna åt oss. Att ta fram förarlösa bilar samt testa dem är dock en dyr process och innebär flera svårigheter. På Chalmers finns projektet Gulliver där man kombinerar en simulerad datormiljö med små fysiska robotbilar för att kunna få en bra kombination av datorsimulerade och verklighetstrogna tester.

Detta kandidatarbete är en vidareutveckling av Gulliverprojektet och arbetets mål är att förbättra och utveckla testmiljön samt utvärdera nya tekniker som kan vara relevanta i Gulliverprojektets framtid. Arbetet har fokuserat på tre delprojekt:

- Förbättra bilarnas lokaliseringssystem genom att integrera mjukvaran GulliView i Gullivers grafiska verktyg. GulliView möjliggör användning av en oberoende källa i form av en webbkamera som använder sig av bildigenkänning för att lokalisera bilarna.
- Förbättra det grafiska verktyg som används vid testning för att snabba upp och göra testproceduren enklare.
- Utvärdera algoritmerna UDP Broadcast och Paxos med avseende på prestanda, speciellt tiden det tar att nå konsensus, för direktkommunikation mellan bilarna.

Nyckelord: Gulliver, autonom minibil, testbädd, förarlösa bilar, Paxos

Förord

Denna rapport har skrivits som en del av ett kandidatarbete vid Institutionen för Data- och Informationsteknik på Chalmers Tekniska Högskola. Arbetet har varit en rolig, utmanande och givande resa och vi i projektgruppen vill rikta ett speciellt tack till vår handledare Elad Schiller samt Thomas Petig för all hjälp och inblick de givit oss i Gulliverprojektet. Vi vill även tacka Claes Ohlsson från avdelningen för fackspråk och kommunikation för värdefull återkoppling under arbetet med rapporten.

Eric Ahlberg, Joakim Danielsson, Marcus Isaksson
Sebastian Ivarsson och Axel Johnsson, Göteborg 2014-06-06

Förkortningslista

AP Access Point
API Application Programming Interface
GUI Graphical User Interface
IP Internet Protocol
NTP Network Time Protocol
RAM Random Access Memory
REST REpresentational State Transfer
RCM Ranging and Communications Module
SQL Structured Query Language
TCP Transmission Control Protocol
UDP User Datagram Protocol
UI User Interface
USB Universal Serial Bus
UWB Ultra Wide Band
V2V Vehicle To Vehicle
XML eXtensible Markup Language

Innehåll

1 Inledning **1**
1.1 Bakgrund . . . 1
1.2 Syfte . . . 2
1.3 Problemformulering . . . 3
1.3.1 Kameralokalisering . . . 3
1.3.2 Förbättring av kartklienten . . . 3
1.3.3 Atomic Broadcast . . . 4
1.4 Metod . . . 4
1.5 Avgränsningar . . . 5
1.6 Relaterat arbete . . . 5
1.6.1 Kameralokalisering . . . 5
1.6.2 Förbättring av kartklienten . . . 6
1.6.3 Atomic Broadcast . . . 6
1.7 Vårt bidrag . . . 6

2 Teori **8**
2.1 Vad är Gulliver? . . . 8
2.1.1 Så fungerar Gulliver . . . 9
2.1.2 Gulliverbilens tidsbegränsningar . . . 9
2.2 Gulliverprojektets mjukvara - kartklienten . . . 10
2.2.1 Git . . . 10
2.2.2 C++ . . . 11
2.2.3 Qt . . . 11
2.3 Kameralokalisering . . . 11
2.3.1 UDP . . . 11
2.3.2 TCP . . . 11
2.3.3 Python . . . 12
2.4 Förbättring av kartklienten . . . 12
2.4.1 XML . . . 12
2.5 Atomic Broadcast . . . 12
2.5.1 Konsensusproblemet . . . 13
2.5.2 Algoritmen Atomic Broadcast . . . 13
2.5.3 C . . . 13
2.5.4 Paxos . . . 14
2.5.5 UDP Broadcast . . . 15
2.5.6 Flask . . . 16
2.5.7 SQLite . . . 16
2.5.8 Tidssynkronisering med NTP . . . 16

2.5.9 Störningar över trådlöst nätverk 17

3 Metod och Genomförande **18**
3.1 Kameralokalisering 18
3.1.1 Modifiering av kartklienten 18
3.1.2 Kommunikation med kameran 19
3.1.3 Sammankoppling av kamera och bilar 19
3.1.4 Testning av systemet 20
3.2 Förbättring av kartklienten 21
3.2.1 Spara och ladda konfigurationer 21
3.2.2 Ta bort bilar 21
3.2.3 Skicka samma rutt till alla bilar 22
3.2.4 Byta färg på bilar 23
3.3 Atomic Broadcast 23
3.3.1 Problem med flera förslagsställare 23
3.3.2 Parametrar för utvärdering 23
3.3.3 Utformning av testmiljö för Paxos 24
3.3.4 Utformning av testmiljö för UDP Broadcast 24
3.3.5 Tidssynkronisering 25
3.3.6 Testning av Paxos 25
3.3.7 Testning av UDP Broadcast 25
3.3.8 Breddning av tester 25

4 Testresultat och validering **27**
4.1 Kameralokalisering 27
4.1.1 Nätverksprestanda 27
4.1.2 Validering av kameralokalisering 27
4.2 Tidsvinster med kartklientens utökade funktionalitet 29
4.2.1 Spara och ladda konfigurationer 29
4.2.2 Skicka samma rutt till alla bilar 30
4.2.3 Total tidsvinst 32
4.3 Validering av kartklientens utökade funktionalitet 32
4.3.1 Spara konfigurationer 33
4.3.2 Ladda konfigurationer 33
4.3.3 Ta bort bilar 33
4.3.4 Skicka samma rutt till alla bilar 34
4.3.5 Byta färg på bilar 34
4.4 Atomic Broadcast 34
4.4.1 UDP Broadcast 35
4.4.2 Paxos 38
4.4.3 Jämförelse av UDP Broadcast och Paxos 40
4.4.4 Nätverk utan störningar 40

5 Diskussion och slutsats **42**

5.1 Utvärdering av arbetsgång 42
5.2 Kameralokalisering 43
5.3 Förbättring av kartklienten 43
5.4 Atomic Broadcast 44
5.4.1 Nätverkets begränsningar 44
5.4.2 Loop-fördröjningen 44
5.4.3 Salvstorlekens påverkan 45
5.4.4 Problem med Paxos 45
5.4.5 Val av algoritm 45
5.5 Framtida utmaningar för autonoma bilar 47
5.5.1 Juridiska frågor 47
5.5.2 Etiska frågor 47
5.6 Slutsats 48

Referenser **51**

Bilaga A Källkod **52**
A.1 Simulering av webbkamera 52
A.2 Uppritning av grafer 54
A.3 Sessionshantering för Atomic Broadcast 55
A.4 Paxosnod 60
A.5 Paxosklient 62

1 Inledning

Förarlösa bilar är en högst aktuell fråga i dagens samhälle då de under de kommande åren kommer att börja rulla på våra vägar. Projektet Gulliver erbjuder en unik möjlighet att testa bilar i verkliga situationer utan att riskera varken människoliv eller stora summor pengar.

Denna rapport beskriver en vidareutveckling av Gulliver där bilarnas lokaliseringssystem förbättrats, det grafiska verktyg som används vid testningen av systemet har effektiviserats samt att de två algoritmerna UDP Broadcast och Paxos utvärderats för användning vid direktkommunikation mellan bilarna.

I detta kapitel beskrivs först bakgrunden till arbetet och förarlösa bilar i allmänhet. Därefter behandlas arbetets syfte, problemformulering samt dess metod och avgränsningar. Vidare redogörs även för relaterat arbete samt hur detta kandidatarbete bidragit till Gulliverprojektet.

1.1 Bakgrund

I Sverige sker varje år hundratals olyckor i trafiken (Transportstyrelsen, 2012). I och med Sveriges riksdags introduktion av projektet Nollvisionen finns ett uttalat mål där inga människor varken ska skadas eller dödas i trafiken (Trafikverket, 2012). Vägen dit är dock lång och det finns många risker som behöver elimineras för att vi ska kunna färdas säkert på vägarna.

Studier visar att den främsta orsaken till olyckor i trafiken är misstag som begåtts av föraren (Forward, 2008). Eftersom den mänskliga faktorn alltid kommer finnas där behövs det istället andra metoder för att få bukt med problemet. Ett alternativ är att ta tekniken till sin hjälp, antingen genom att assistera föraren vid kontrollen av fordonet eller att helt automatisera körningen. Följande arbete kommer att fokusera på det sistnämnda.

För många kan förarlösa bilar låta som något hämtat ur en Science fiction-film och kan tyckas orealistiskt inom den närmaste framtiden, men faktum är att denna teknik existerat sedan länge. Google har till exempel modifierat vanliga personbilar så att de kan köras utan förare och de har sammanlagt kört över 48 000 mil utan en enda olycka (Urmson, 2012). Men det är inte bara Google som testat förarlösa bilar. Volvo, som alltid profilerat sig som ett bilföretag med starkt fokus på säkerhet, har inlett ett samarbete med Göteborgs Stad för att driftsätta ett projekt med självkörande bilar framförda av privatpersoner på Göteborgs gator från och med år 2017 (Volvo Cars, 2013). Detta är

ett steg på vägen mot den vision Volvo satt upp: att ingen människa varken ska skadas allvarligt eller dödas i en nyproducerad Volvobil år 2020 (Volvo Cars, 2008).

Att helt automatisera bilkörningen kräver dock helt nya bilar med avancerade datorsystem som kan hantera alla de situationer som kan uppkomma i trafiken. Utvecklingen av sådana bilar kräver mycket forskning och ny teknik, men det behövs också utförlig testning av systemen. Med syfte att minska kostnaderna för denna testning startades ett forskningsprojekt kallat Gulliver på Institutionen för Data- och Informationsteknik på Chalmers år 2011 (Pahlavan, Papatriantalou och Schiller, 2011). Med Gulliver hoppas man kunna underlätta utvecklingen av förarlösa bilar genom att minska både kostnaden för testning samt tiden det tar att gå från idé till implementering av ny teknik. På så sätt kan den totala utvecklingskostnaden för förarlösa bilar reduceras.

För att testa helautomatiserade fordonssystem i full skala krävs mycket resurser, med bilar som har höga driftkostnader och kräver stora testanläggningar. Sådana system kan ha kostnader på cirka hundratusen till en miljon euro per bil, med testanläggningar som kan kosta tiotals miljoner euro, att jämföra med cirka 2000 euro för en Gulliverbil (Pahlavan, Papatriantalou och Schiller, 2011).

Gulliver erbjuder en testmiljö bestående av eldrivna miniatyrbilar samt ett grafiskt verktyg för övervakning och styrning av testningen, kallad kartklient. Kartklienten kan köras på en vanlig bärbar dator uppkopplad på samma nätverk som bilarna. Miniatyrbilar kan ge mer verklighetstrogna tester än simuleringar, samtidigt som de är mer kostnadseffektiva i jämförelse med fullskaliga bilar.

Förutom att öka säkerheten i trafiken kan förarlösa bilar även bidra till en bättre miljö. Transportsystemet är till stor del beroende av fossila bränslen och står för nästan en tredjedel av Sveriges totala utsläpp av växthusgaser (Naturvårdsverket, 2013). Förbränning av fossila bränslen står för det största bidraget till växthuseffekten i Sverige och i övriga världen. En av fördelarna med självkörande bilar är att de kan köras effektivare än traditionella bilar. Volvo visade upp fungerade fordonståg år 2012 då ett antal bilar kunde följa ett ledfordon automatiskt på motorvägen. Mätningar efter Volvos testkörning visar att fordonståg kan reducera energianvändningen med upp till 20 % (Volvo Cars, 2012).

Med detta i åtanke finns det mycket som tyder på att förarlösa bilar - utvecklade med hjälp av småskaliga testmiljöer - är en attraktiv väg mot ett säkrare samhälle och samtidigt ett välkommet tillskott i jakten på minskade utsläpp av växthusgaser.

1.2 Syfte

Syftet med arbetet är att göra det lättare att testa och köra småskaliga förarlösa bilar. Detta uppnås genom att bygga vidare på och förbättra Gulliver samt utvärdera nya tekniker som skulle kunna användas.

1.3 Problemformulering

Då Gulliver är ett pågående forskningsprojekt utvecklas det kontinuerligt och det finns utrymme för förbättringar inom många områden. Flera av de existerande funktionerna är i behov av att förbättras och flera nya funktioner önskas för att underlätta användningen av systemet. I samråd med Gulliverprojektets ledning har följande tre delprojekt valts ut i syfte att effektivisera och förbättra Gulliver: *kameralokalisering*, *förbättring av kartklient* samt *Atomic Broadcast*.

1.3.1 Kameralokalisering

Precisionen för Gulliverbilens manövrar och dess maximala hastighet under testning är kopplad till positioneringssystemets exakthet. För tillfället finns det bara en lokaliseringskälla, vilken grundar sig på en UWB-teknik (Gezici et al., 2005) som använder sig av ett flertal RCMs. Tekniken bygger på sändning av radiovågor över ett brett frekvensintervall och lämpar sig väl för tillämpningar där information snabbt ska sändas över korta sträckor. Detta positioneringssystem har dock svårt att hantera störningar och mätbrus vilket leder till felaktiga värden och onödiga korrektioner av styrsystemet. En konsekvens av detta är att bilen har svårigheter att hålla sin givna kurs. För att lösa problemet krävs bättre positioneringsdata vilket kan uppnås genom att använda flera oberoende tekniker för positionering, då dessa inte är känsliga för samma typer av störningar. Med flera oberoende källor kan en mer korrekt position beräknas.

Det positioneringssystem som ska integreras är ett existerande system som använder sig av en videokamera som med hjälp av bildigenkänning läser av och returnerar bilarnas aktuella position. Ett första steg för att lägga till denna nya lokaliseringskälla är att implementera stöd i kartklienten för att kunna ta emot och presentera dess positionsdata.

1.3.2 Förbättring av kartklienten

När man testar Gulliversystemet använder man sig av en kartklient på en dator för att övervaka och kontrollera testresultaten. Man vill ofta köra sina test med flera olika inställningar och i olika miljöer för att få så bra och trovärdiga test som möjligt. Kartklienten saknar dock flera viktiga funktioner som gör det smidigt att växla mellan dessa testsessioner. Detta har lett till att man inte kunnat upprepa sina test och att det har varit tidskrävande att byta mellan testen. Att kunna upprepa test på exakt samma sätt är en förutsättning för att kunna genomföra verifierbara tester och jämföra resultat från olika testsessioner.

Målet med uppgiften var först och främst att förkorta tiden det tar att förbereda testningen men även att förenkla testproceduren. Detta ska lösas genom att lägga till funktioner

som löser de mest tidskrävande förberedelsemomenten såsom att ställa in inställningar för rummet man testar i samt att skicka samma rutt till alla bilar. Även funktioner för att ta bort och byta färg på bilar efterfrågades för att göra testsessionerna mer effektiva och lättförståeliga.

1.3.3 Atomic Broadcast

För att kunna simulera verkliga trafiksituationer involverande flera bilar krävs stöd för överenskommelse mellan bilarna. En förutsättning för att uppnå detta i ett distribuerat system är någon form av direktkommunikation mellan bilarna där man kan veta att meddelanden kommer fram och att de kommer fram i rätt ordning. Detta kallas Total Order Broadcast eller Atomic Broadcast (Défago, Schiper och Urbán, 2004). Ett exempel där Atomic Broadcast är användbart är när bilar möts i en korsning och tillsammans måste bestämma vem som ska köra först för att undvika en kollision.

Eftersom kommunikationen i testsystemet sker över ett opålitligt trådlöst nätverk kan både långa förseningar och förlust av meddelanden förekomma. För att uppnå pålitlig kommunikation över ett sådant nätverk krävs någon sorts algoritm för att säkerställa att alla bilar får korrekt information så att ett riktigt beslut kan fattas. Den algoritm som används i Gulliver är en simpel implementation som använder sig av UDP Broadcast. Denna saknar helt garantier att det beslut som fattas är korrekt.

Det finns algoritmer som hanterar de omständigheter som finns i Gulliver, ett exempel är Paxos (Lamport, 1998). Nackdelen med en sådan algoritm är att den kräver många fler meddelanden för att nå överenskommelse, och det är därför inte helt klart ifall det kommer ge bättre prestanda än UDP Broadcast i den nätverksmiljö Gulliver körs i. För att ta reda på detta behöver dessa två algoritmer utvärderas med avseende på prestanda och det som är av särskilt intresse är tiden det tar att komma överens.

1.4 Metod

Då arbetet är uppdelat i tre delprojekt med olika frågeställningar har olika metoder använts i respektive delprojekt. Något som dock varit gemensamt för alla delprojekt är användningen av en agil systemutvecklingsmetodik där ett iterativt arbetssätt förespråkas och fokus ligger på flexibilitet. Eftersom arbetet syftar till att utveckla ett redan existerande projekt har det i framför allt delprojekten *kameralokalisering* samt *förbättring av kartklienten* lagts tid på studier om nuvarande systemens funktion samt implementation.

Utvecklingen av kameralokaliseringen har inneburit inläsning om berörda tekniker samt hur integrering i Gulliver ska ske, en experimentell fas för att nå en lösning och en utvärderingsfas där implementationen testats och validerats. Förbättringen av kartklienten har inte innefattat inläsning i samma utsträckning utan där har fokus legat på utveckling

och förbättring av redan existerande programvara. Delprojektet Atomic Broadcast har haft en mer vetenskaplig karaktär då stort fokus lagts på studier av redan existerande algoritmer samt implementationer. Därefter har experiment gjorts för att utvärdera olika implementationers prestanda utifrån relevanta mått.

1.5 Avgränsningar

Arbetet syftar till att vidareutveckla redan existerande mjukvara samt att stödja bakåtkompatibilitet i den mån det går och då framför allt när det gäller existerande algoritmer, exempelvis manöver för filbyte, korsning av väg samt fordonståg. Vi kommer däremot inte ägna oss åt optimering, vidareutveckling eller nyutveckling av hårdvara.

De algoritmer som utvärderats under arbetets gång har implementerats för testning men inte för slutlig integrering i själva Gulliver då detta ligger utanför arbetets ramar.

Även om kopplingar till verkliga trafiksituationer existerar så syftar detta arbete specifikt till utveckling och optimering utifrån Gullivers förutsättningar och villkor. Experiment, utvärderingar och slutsatser är skrivna med detta i åtanke vilket innebär att de inte nödvändigtvis är giltiga när det gäller trafiksituationer under andra förhållanden.

1.6 Relaterat arbete

Det pågår i dagsläget mycket arbete för att utveckla förarlösa bilar, både inom industrin och inom akademin. Många av de stora biltillverkarna har projekt för att utveckla sina existerande bilar och göra dem mer säkra, samtidigt som många universitet har egna projekt som utvecklar teknik som kan användas i framtiden (Burns, 2013). Den stora skillnaden med Gulliverprojektet i jämförelse med dessa är att det är ett småskaligt system, vilket innebär att kostnaderna hålls nere och effektiviteten ökas då testningen av förarlösa bilar blir mer lättillgänglig. För de områden arbetet har behandlat finns det relaterat arbete specifikt för vart och ett av de olika delprojekten som varit intressant.

1.6.1 Kameralokalisering

Problemet med lokalisering av robotar är mycket vanligt och det finns en mängd olika metoder för att lösa det. I Gulliverprojektet har man valt en lösning som använder sig av bildigenkänning. I ett projekt av Fiala (2004) används ett liknande lokaliseringssystem som med hjälp av en webbkamera överblickar och styr robotarna. Däremot kombineras detta inte med indata från en annan typ av lokaliseringssystem som det slutliga målet är i Gulliver.

Parallellt med detta kandidatarbete har ett projekt vid namn GulliView utförts med syftet att framställa ett system för lokalisering med hjälp av en webbkamera och bildigenkänning (Hangal och Söderberg-Rivkin, 2014). Resultaten av detta projekt har använts och byggts vidare på för att kunna integreras i Gullivers kartklient.

1.6.2 Förbättring av kartklienten

Det mest relevanta arbetet för detta delprojekt har varit det som behandlar utvecklingen av Gullivers kartklient (Dahlgren et al., 2012), som är den mjukvara som har byggts vidare på under arbetet. Det finns andra liknande projekt, exempelvis SUMO (Behrisch et al., 2011), där man till skillnad från i Gulliver använder ren simulering för testning av fordonssystem.

1.6.3 Atomic Broadcast

Det finns en mängd prestandatest för den distribuerade algoritmen Paxos (De Prisco, Lampson och Lynch, 1997; Primi, 2008; Vieira och Buzato, 2009), men dessa fokuserar mest på implementationen av algoritmen och använder sig av snabba stabila nätverksanslutningar över trådbundet Ethernet. Dessa tester är dock inte riktigt representativa för Gulliverprojektet, där fördröjningar över det trådlösa nätverket kan vara flera storleksordningar större än skillnaderna i exekveringstid för de olika tillgängliga implementationerna.

1.7 Vårt bidrag

Vårt bidrag till Gulliverprojektet består av tre olika typer av uppgifter vilket har lett till att förbättringarna har tre helt olika typer av karaktär. Vårt arbete har resulterat i följande förbättringar i Gulliverprojektet:

- Implementerat stöd för en videokamera som en ytterligare lokaliseringskälla för att möjliggöra förbättring av bilens förmåga att hålla sin kurs och köra i en högre hastighet.
- Genom utveckling och förbättring av kartklienten har förberedelsetiden för testning sänkts från cirka 20 till 5 minuter.
- Utvärderat vilken algoritm som bör användas för direktkommunikation mellan bilarna genom att utförligt testa implementationer av de två algoritmerna UDP Broadcast och Paxos i ett flertal miljöer och med ett stort urval av parametrar. Fokus för utvärderingen har legat på tiden det tar att nå konsensus.

Med hjälp av dessa förbättringar har användarupplevelsen förbättrats, testningsproceduren effektiviserats samt ett underlag tagits fram för beslut om framtida integrering av kommmunikationsprotokoll.

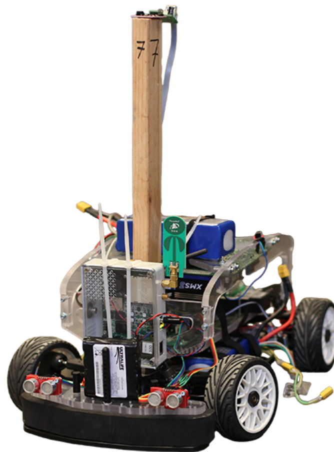
2 | Teori

I detta kapitel kommer den teori som behövts läsas in under arbetets gång tas upp. Kapitlet är indelat i fem delar, där de två första delarna beskriver den allmänna kunskapen som behövs för att arbeta i Gulliverprojektet, dels genom att beskriva Gulliver i allmänhet samt kartklienten i synnerhet. De tre efterföljande delarna beskriver den nödvändiga teori som är relaterad till vart och ett av de tre delprojekten.

2.1 Vad är Gulliver?

Gulliver är som tidigare nämnts en testmiljö för självkörande bilar i miniatyrformat. Syftet med Gulliver är att på ett kostnadseffektivt och realistiskt sätt kunna testa olika trafiksituationer och tekniker för att bidra till utvecklingen av självkörande bilar. Bilarna använder sig av sensorer för att känna av sin omgivning och på så sätt kunna köras på ett säkert sätt. De är även uppkopplade på samma Wi-Fi nätverk som den dator som kör kartklienten för att kunna kommunicera.

Innan arbetets början kunde Gulliverbilarna (se figur 2.1) bland annat autonomt följa en utsatt väg, köra på led, känna av och bromsa för hinder framför bilen, göra omkörningar, utföra enklare filbyten samt styras manuellt med hjälp av en Xbox-kontroll.



Figur 2.1: Den självkörande Gulliverbilen.

2.1.1 Så fungerar Gulliver

För att kunna köra tester på Gulliverbilarna och dess mjukvara behövs ett antal inställningar ställas in såsom referenspunkterna för bilarnas lokaliseringssystem och rutterna som bilarna ska köra. Vid testning behöver först ett koordinatsystem definieras. Detta görs genom att sätta ut tre referenspunkter, så kallade ankare, i rummet som man testar bilarna. Ankarna, som visas i figur 2.2, används av bilarnas lokaliseringssystem för att räkna ut var de befinner sig och sedan skicka informationen till kartklienten. I kartklienten behöver man manuellt mata in positionen för dessa referenspunkter på nytt varje gång man utför ett test.

Figur 2.2: Ankare för lokaliseringssystemet bestående av en mikrokontroller och en RCM-antenn.

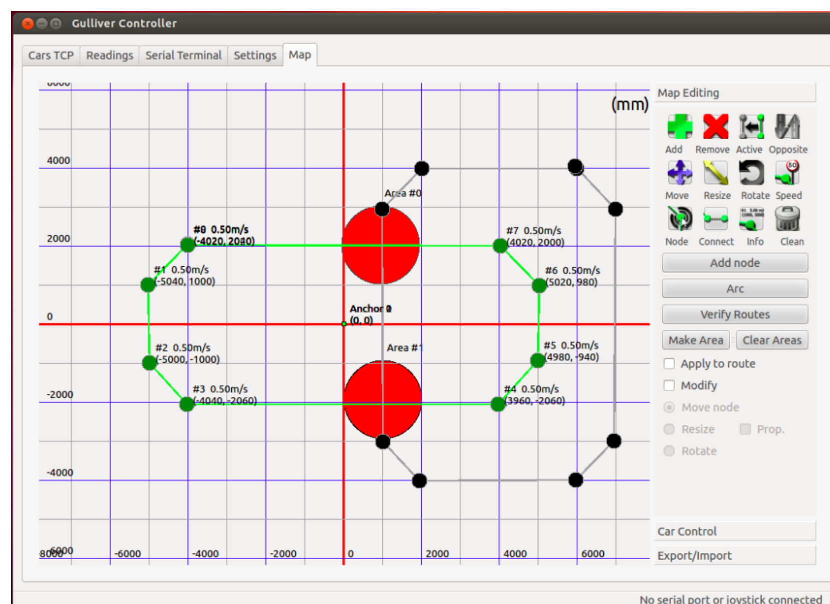
För att Gulliverbilen ska kunna navigera följer den en rutt uppbyggd av ett antal punkter, kallade noder. Dessa noder består av en position, en hastighet som ska hållas av bilen efter att den har passerat noden och ett index. När bilen blir tilldelad en rutt åker den först till noden med lägsta indexet och sedan till nästa index och så vidare. Dessa rutter ritas ut av användaren i kartklienten, för att sedan laddas upp till de bilar man vill ska följa rutten.

2.1.2 Gulliverbilens tidsbegränsningar

Då Gulliverbilen är ett realtidssystem är inte bara beräkningarnas korrekthet viktig utan även att de klarar tidskraven. Bilens logik exekveras var 10:e millisekund, vilket gör att det finns ett önskemål att data skall vara tillgänglig inom detta intervall. Det finns dock data som inte alltid klarar detta tidskrav, exempelvis data skickad över det trådlösa nätverket mellan bilarna, och då måste hänsyn tas till hur gammal denna data är. För denna data finns andra tidskrav på hur gammal datan får vara innan den anses ointressant, vilket beror på vad datan används till. Det skulle exempelvis kunna vara hårdare krav på kommunikationsdatan mellan bilarna när de följer varandra i ett fordonståg i hög hastighet än inkommande data från kartklienten med icke tidskritisk testinformation.

2.2 Gulliverprojektets mjukvara - kartklienten

Gullivers kartklient består av ett projekt med öppen källkod som hämtades hem från versionshanteringshemsidan Bitbucket. Denna kod skulle användas för att lösa de två delprojekt som innebar förbättring och utveckling av funktioner i Gullivers mjukvara. Den del av Gullivers mjukvara som rörde detta kandidatarbete är utvecklat i programspråket C++ och uppbyggt på applikationsramverket Qt. Detta var nytt för flera i gruppen och innebar att en stor del av inläsningen gick åt till att lära sig språket samt att få en förståelse för den redan befintliga koden. I figur 2.3 kan man se hur kartan ser ut i kartklienten.



Figur 2.3: Gullivers kartklient med två stycken rutter inlagda där de röda cirklarna markerar korsningar.

2.2.1 Git

För att möjliggöra utveckling i samma kodbas av flera användare samtidigt och säkerställa att viktig data inte skulle kunna gå förlorad har versionshanteringssystemet Git använts (Loeliger, 2009). Git skapades ursprungligen för att hantera källkoden till Linuxkärnan och är skrivet med fokus på att optimera prestanda. Fördelen med att använda Git inom Gulliverprojektet är att man kan utgå från en gemensam kodbas som man blivit tilldelad, utveckla självständigt och sedan slå samman när funktionaliteten är färdigtestad och väl genomarbetad.

2.2.2 C++

C++ uppfanns år 1985 av Bjarne Stroustrup som en kraftig utvidgning av det redan befintliga programsspråket C (Ellis och Stroustrup, 1990). Under 1990-talet växte sig språket mycket populärt och blev det första stora objektorienterade programsspråket. Eftersom den existerande mjukvaran redan var skriven i C++ så var det ett naturligt val att fortsätta använda sig av det. C++ kan med hjälp av ett utökande bibliotek kallat Qt användas för att utveckla plattformsoberoende grafiska applikationer vilket gör att det lämpar sig väl inom Gulliverprojektet.

2.2.3 Qt

Qt är ett deklarativt UI-språk med ett plattformsoberoende C++ bibliotek som ger användaren möjlighet att nå olika typer av system med en och samma kodbas (Blanchette och Summerfield, 2008). I Gulliverprojektet används Qt för att underlätta den grafiska hanteringen med C++. Qt innehåller ett stort antal C++ bibliotek med API:n för det mesta som kan behövas för att skapa en applikation. Detta tillsammans med att Qt är plattformsoberoende gör Qt till ett bra val i Gulliverprojektet.

2.3 Kameralokalisering

Då kommunikationen mellan kameran och Gullivers kartklient sker över nätverk behövdes inläsning av nätverksprogrammering samt om transportprotokollet UDP ske. För att underlätta testning i ett tidigt stadium användes programspråket Python för att simulera den riktiga webbkameran som datakälla.

2.3.1 UDP

UDP är ett förbindelselöst dataprotokoll som prioriterar hög hastighet framför tillförlitlighet för att kunna skicka data mellan datorer (Postel, 1980). Till skillnad från transportprotokollet TCP (Postel, 1981) som garanterar leverans, är det inte säkert att ett UDP-datagram kommer fram till sin mottagare. UDP är lämpligt att använda i kameralokaliseringen då det är viktigare att paket kommer fram snabbt än att alla paket garanterat kommer fram.

2.3.2 TCP

TCP är ett förbindelseorienterat dataprotokoll som till skillnad från UDP prioriterar tillförlitlighet framför snabbhet. TCP tillhandahåller flera underliggande mekanismer som gör att paket som skickas garanterat kommer fram till mottagaren. TCP används

inom Gulliver för att etablera kopplingen mellan kartklienten och respektive bil, men det används även inom Paxos-algoritmen för att säkerställa de garantier som algoritmen erbjuder.

2.3.3 Python

Python är ett programspråk av typen skriptspråk som utvecklades år 1991 av Guido van Rossum (Rossum, 1995). Python möjliggör objektorienterad, funktionell och imperativ programmering och har stort fokus på läsbarhet och förståelighet. För testningen användes Python dels för att det är ett språk som det går fort att skriva enklare program i men även för att det finns många användbara bibliotek, exempelvis för nätverkskommunikation, datahantering och matematisk statistik. Detta gjorde att språket lämpade sig särskilt väl i projektet med kameralokalisering där ett skript som simulerade webbkamerans funktion utvecklades istället för att arbeta mot den riktiga webbkameran.

2.4 Förbättring av kartklienten

Då arbetet med att förbättra kartklienten fortsatte efter projektet med kameralokaliseringen kunde den kunskap som tillgodogjorts användas även i detta delprojekt. Det krävdes dock vidare inläsning av XML då det användes för kartklientens filhantering.

2.4.1 XML

XML är en standard för strukturmärkning av elektroniska dokument. I språket kan man definiera märkord för att ge texten en innehållsmässig struktur (Bray et al., 1998). Det fanns redan vissa funktioner i kartklienten som använde sig av XML vilket gjorde att gruppen valde att fortsätta använda sig av detta språk för funktionerna spara och ladda konfigurationer.

2.5 Atomic Broadcast

För att utvärdera algoritmer som skulle kunna användas vid kommunikation mellan Gulliverbilarna har mycket tid ägnats åt litteraturstudier. Med syfte att skapa en förståelse kring problematiken med distribuerade system samt algoritmen Paxos och dess tillämpningsområden studerades inledningsvis en artikel skriven av Paxos upphovsman Leslie Lamport (Lamport, 1998). Dessutom studerades artiklar som beskriver Paxos med mer konkreta exempel, så som *Paxos by example* skriven av MacDonald (2012). För att

utveckla kunskap kring hur en prototyp för prestandatester skulle utformas studerades därefter den existerande implementation av algoritmen som valts ut i samråd med Gulliverprojektets ledning vid namn LibPaxos (LibPaxos, 2013).

Eftersom Atomic Broadcast-projektet är uppdelat i två delar, en del där UDP Broadcast utvärderas och en del där den distribuerade algoritmen Paxos utvärderas, behövde flera olika ämnen studeras. Ett gemensamt behov för de båda delarna var att snabbt och enkelt kunna analysera den insamlade datan från experimenten. För detta användes en SQLite-databas för lagring av datan samt Python för analysen.

2.5.1 Konsensusproblemet

Ett av de mest grundläggande problemen inom datavetenskap är konsensusproblemet, vilket handlar om att nå enighet mellan ett antal opålitliga processer i ett distribuerat system. Inom projektets gränser innebär en opålitlig process en Gulliverbil som när som helst kan tappa anslutningen till övriga bilar. Om några av processerna föreslår värden ska den algoritm som löser problemet garantera att endast ett av dessa värden godtas av de andra processerna i systemet. Om inget värde har föreslagits får inget värde heller godtas.

2.5.2 Algoritmen Atomic Broadcast

Atomic Broadcast, även kallad Total Order Broadcast, är en familj algoritmer som löser konsensusproblemet (Défago, Schiper och Urbán, 2004). Atomic Broadcast garanterar tillförlitlig kommunikation i ett distribuerat nätverk vilket gör den till ett intressant alternativ då Gulliverbilarna tillsammans utgör just ett distribuerat nätverk där man önskar tillförlitlig kommunikation. För att implementera Atomic Broadcast används ofta en algoritm vid namn Paxos (Lamport, 2001). Enligt många är Paxos en komplicerad och svårförståelig algoritm, och därför har mycket arbete lagts ner för att studera och förstå Paxos. På internet kan man hitta flera existerande implementationer av Paxos, där en av dessa är LibPaxos, ett öppet och fritt projekt skrivet i programspråket C. För att kunna använda LibPaxos krävdes en ordentlig fördjupning i dess funktionalitet och API.

2.5.3 C

C är ett maskinnära programspråk som utvecklades under 1970-talet (Kernighan och Ritchie, 1978). Eftersom C är maskinnära kan programmen göras mycket snabba och effektiva. Språket är även kompatibelt med en mängd olika hårdvarukombinationer och anses vara mycket portabelt. Detta gör C till ett lämpligt val för att implementera en distribuerad algoritm som Paxos.

2.5.4 Paxos

Paxos är en distribuerad algoritm som kan användas för att implementera Atomic Broadcast. En distribuerad algoritm är en algoritm som kör på flera, utspridda datorer, exempelvis över ett lokalt nätverk. Fördelen med att använda sig av Paxos är att paket som skickas garanterat kommer fram samt att systemet fortsätter fungera så länge en majoritet av de medverkande datorerna är vid liv.

I Paxos agerar varje dator som en nod, vars mål är att få ett värde antaget. Paxos garanterar då att alla medverkande noder får reda på att detta värde har antagits. Algoritmen kan sedan upprepas i flera rundor, där noderna har ett antal värden de vill få antagna, och fortsätter att försöka få sina värden antagna runda efter runda. När ett värde blivit antaget kan noden övergå till att föreslå nästa värde i efterföljande rundor. Paxos innehåller flera komponenter som ger vissa garantier om det värde som antas.

- Icke trivialitet - Det värde som antas i en runda har blivit föreslaget av minst en nod i den rundan.
- Säkerhet - Bara ett värde kan antas i varje runda.
- Framsteg - Om minst ett värde föreslås, kommer alltid ett värde antas.

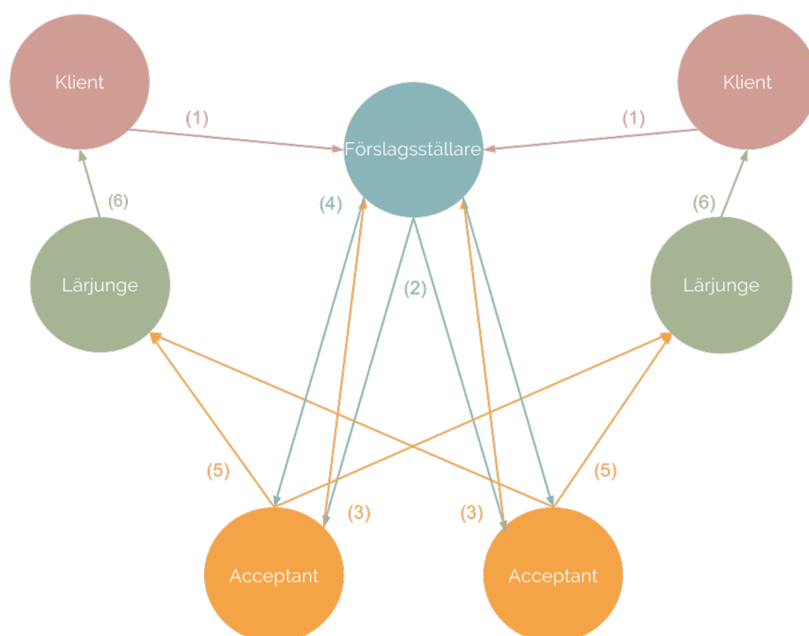
När algoritmen körs innehar varje nod en eller flera av följande roller: klient, förslagsställare, acceptant eller lärjunge.

I figur 2.4 visas en skiss över kommunikationen under en runda av Paxos. De olika meddelandena som skickas visas som siffror och förklaras löpande. En runda börjar med att en klient skickar ett värde v_1 till en förslagsställare vilket visas i figuren som (1). Denna förslagsställare tilldelar detta värde sekvensnumret n_1 och skickar sedan en förfrågan om ett löfte till en majoritet av acceptanterna, vilket ses som (2) i figuren. Löftet består av att acceptanten lovar att inte acceptera ett värde med lägre sekvensnummer än n_1.

Acceptanten svarar att den ska hålla löftet endast om den inte tidigare utfärdat ett löfte om ett sekvensnummer högre än n_1 (visas som (3) i figuren). Ifall acceptanten tidigare har accepterat ett värde v_x för sekvens n_x, returnerar den även dessa. Om förslagsställaren får svar om att en majoritet av acceptanterna håller sitt löfte skickar den sin förfrågan till dem (meddelande (4) i figuren). Denna förfrågan innehåller sekvensnumret n_1 och, om inget värde $v_x, x \neq 1$ returnerats, värdet v_1. Ifall ett eller flera sekvensnummer $n_x, x \neq 1$ returnerats, väljs värdet v_x från det förslag x med högst sekvensnummer n_x.

Så länge inte en acceptant har mottagit en ny förfrågan om löfte med ett sekvensnummer högre än n_1 accepterar den värdet och meddelar alla medverkande lärjungar (meddelande (5) i figuren). När lärjungarna fått information om att ett värde blivit accepterat av en majoritet av acceptanterna är värdet antaget och resultatet meddelas till alla klienter som ses med (6) i figuren.

Paxosalgoritmen används flitigt inom industrin, bland annat av Google för hantering av



Figur 2.4: Skiss över ett exempel av en instans av Paxos där cirklarna motsvarar rollerna i algoritmen och pilarna de meddelanden som skickas. Ordningen ges av siffrorna vid pilarna.

delade resurser i företagets datacenter (Chandra, Griesemer och Redstone, 2007). Även om användningen av Paxosalgoritmen inom distribuerade system är vida utbredd finns det relativt lite information kring det ingenjörsmässiga arbetet som krävs för integrering i ett befintligt system, sannolikt för att många av implementationerna som används är utvecklade av företagen och är således företagshemligheter.

2.5.5 UDP Broadcast

För att utvärdera prestandan för Atomic Broadcast har jämförelser gjorts med den mycket simplare algoritmen UDP Broadcast. Principen för UDP Broadcast är att sända det föreslagna värdet på datorns så kallade broadcast-adress, vilket innebär att värdet skickas till alla datorer inom samma nätverk. Till skillnad från Paxos, som tillhandahåller vissa garantier, så är en nackdel med UDP Broadcast att det helt saknas garantier för leveransen av det UDP-paket som innehåller det föreslagna värdet. En fördel med detta är dock att det i teorin behöver skickas färre paket vilket innebär att tiden det tar för ett värde att komma fram minskar så länge nätverksanslutningen är stabil. Då Gulliver är ett tidskritiskt realtidssystem är tidskraven hårda och det är önskvärt med en snabb leveranstid av paket, men det är också viktigt att paketen faktiskt kommer fram. Ett sätt att öka sannolikheten att ett paket kommer fram med UDP är att skicka varje paket flera gånger. Detta sker dock på bekostnad av överföringshastigheten då det leder till mer trafik över nätverket.

Då UDP Broadcast skulle studeras behövdes kunskaper dels om UDP, men även vidare kunskaper om Python. Återigen valdes Python tack vare enkelheten att skapa skript som kommunicerar över nätverket.

2.5.6 Flask

Flask är ett ramverk för att utveckla dynamiska hemsidor och webbapplikationer i Python (Ronacher, 2014). Flask tillhör kategorin mikroramverk och enligt utvecklarna av Flask innebär det att ramverket ska hållas simpelt men med möjlighet för utbyggnad. Funktionaliteten hos Flask inkluderar inbyggd utvecklingsserver, stöd för enhetstestning, rutthantering enligt REST-principer samt stöd för Unicode. Vid skapande av webbtjänster där hög utvecklingstakt står i fokus kan Flask vara väldigt kraftfullt då dess flexibilitet ger användaren stor frihet att utvidga funktionalitet efter behov.

2.5.7 SQLite

SQLite är ett relationsdatabashanteringssystem som fokuserar på snabbhet och kompakthet (Owens och Allen, 2006). Det har stor kompatibilitet med de flesta operativsystem och programmeringspråk vilket gör det lätt att använda. Det behövs heller ingen separat server-process utan SQLite inkluderas som ett mjukvarubibliotek. Tillsammans med Python kan därför SQLite användas för att enkelt lagra testdata, men också för att få tag på relevant data då den relationsbaserade datamodellen gör det enkelt och flexibelt att arbeta med stora mängder data.

Till skillnad från vanliga datastrukturer - såsom listor och tabeller i Python - där man måste veta var och hur datan är lagrad för att få fram den, kan man med en databas få fram datan mycket enklare. Detta på grund av den underliggande relationsmodellen som möjliggör förfrågningar utifrån datans struktur istället för dess position.

2.5.8 Tidssynkronisering med NTP

Ett problem vid exakt tidsmätning mellan olika datorer över nätverk är att datorernas klockor är osynkroniserade, vilket kan ha en stor inverkan på det slutliga testresultatet. För att så gott som möjligt synkronisera klockorna används protokollet NTP (Mills, 1991). NTP fungerar genom att man på de olika datorerna definierar ett kluster av olika NTP-servrar som används som gemensam källa för tid. Därefter beräknas en avdrift på respektive dators klocka vilken sedan kompenseras för att till slut få fram en gemensam tid.

2.5.9 Störningar över trådlöst nätverk

Ett problem med att använda sig av trådlösa nätverk på de vanliga frekvensbanden 2,4 och 5 GHz är förekomsten av olika typer av störningar. Effekterna av detta är bland annat att paket förloras, hastigheten försämras eller i värsta fall att nätverket helt och hållet störs ut. Källorna till dessa störningar kan vara andra apparater som opererar inom samma frekvensintervall, exempelvis andra trådlösa nätverk, mikrovågsugnar, billarm, bluetooth-signaler eller mobiltelefoner.

Då störningar för ett säkerhetskritiskt system skulle kunna leda till olyckor är det viktigt att de förekommer i så liten utsträckning som möjligt. Därför finns det ett separat frekvensband på 5,9 GHz, kallat V2V (Stojaspal, 2013), avsett för att endast användas för kommunikation mellan olika fordon. Då detta frekvensband är begränsat till bilindustrin och inte får användas för annat, är det bara möjligt att testa på de vanligaste frekvensbanden för Wi-Fi: 2,4 och 5 GHz. För att simulera en miljö liknande V2V kan dock tester utföras där störningarna är minimerade, exempelvis på en avlägsen plats.

3 | Metod och Genomförande

I detta kapitel behandlas tillvägagångsättet som använts under arbetets gång. De tre delprojekten beskrivs var för sig och i dessa avsnitt tas intressanta aspekter från arbetet upp. Gemensamt för alla projekten är användningen av en agil systemutvecklingsmetodik kallad Scrum (Schwaber, 1997) samt versionshanteringssystemet Git.

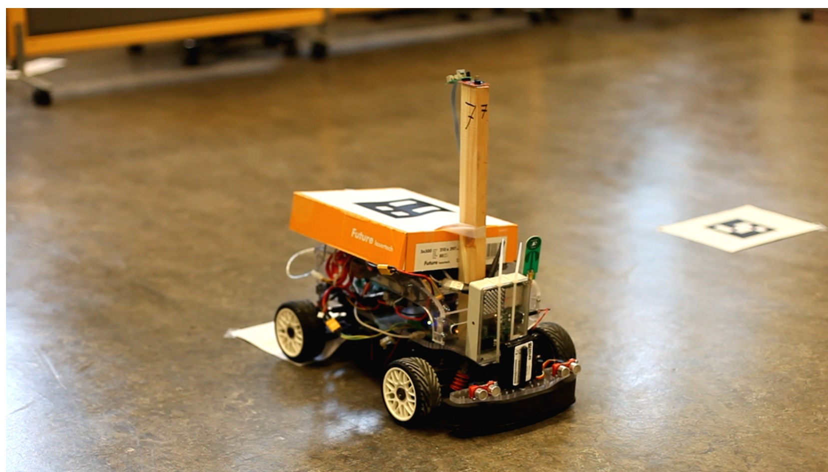
3.1 Kameralokalisering

För att förbättra bilarnas lokalisering krävs mer exakt positioneringsdata. Eftersom det nuvarande lokaliseringssystemet, som är baserat på UWB, är känsligt för störningar och mätbrus vill man eliminera dessa genom att använda flera lokaliseringskällor som inte påverkas av samma typer av störningar. I Gulliverprojektet har man bestämt sig för att använda ett system som med hjälp av bildigenkänning beräknar bilarnas positioner. För att göra detta filmas bilarna med en webbkamera som beräknar deras position relativt ett koordinatsystem man definierar med hjälp av April-taggar (Olson, 2011) på golvet. För att kameran ska kunna känna igen bilarna behöver även bilarna utrustas med April-taggar, se figur 3.1.

Systemet kan köras på en separat dator med en inkopplad webbkamera. Denna dator kör en server som tar emot och buffrar data över bilarnas position från kameran. Vid förfrågan över nätverket svarar den genom att skicka ut datan från bufferten. Eftersom utvecklingen av detta system är i ett tidigt stadie vill man först och främst ha ett sätt att kontrollera hur datan från det nya systemet stämmer överens med det befintliga positioneringssystemet.

3.1.1 Modifiering av kartklienten

Det första steget som behövde tas för att använda den nya lokaliseringskällan i Gulliverprojektet var att implementera stöd för ytterligare positionsdata i kartklienten. För att göra detta behövde delar av den underliggande strukturen i kartklienten och det grafiska gränssnittet ändras. Kartklienten utökades för att kunna lagra två olika sorters positionsdata för varje bil samt för att kunna rita ut båda dessa positioner för respektive bil.



Figur 3.1: Gulliverbilen utrustad med en April-tagg på taket.

3.1.2 Kommunikation med kameran

För att kunna kommunicera med servern behövde kartklienten utökas så att den kunde skicka förfrågningar över UDP och ta emot svar från kameran innehållande positionsdata. UDP valdes på grund av dess snabbhet och enkelhet, då äldre data är ointressant och paketens leveranstid bör minimeras. Förfrågningarna skickades i form av paket innehållandes endast två fält, typ och undertyp, medan kamerans svarspaket hade en struktur enligt tabell 3.1.

Tabell 3.1: Strukturen för kamerans svarspaket innehållandes lokaliseringsinformation för n bilar.

0	31	63	95	127
Typ	Undertyp	Sekvensnummer	Tid (s)	Header
	Tid (μs)		Antal bilar	
Bil-id	X-position	Y-position	Riktning	Bil 1
		⋮		Bil 2 … n

3.1.3 Sammankoppling av kamera och bilar

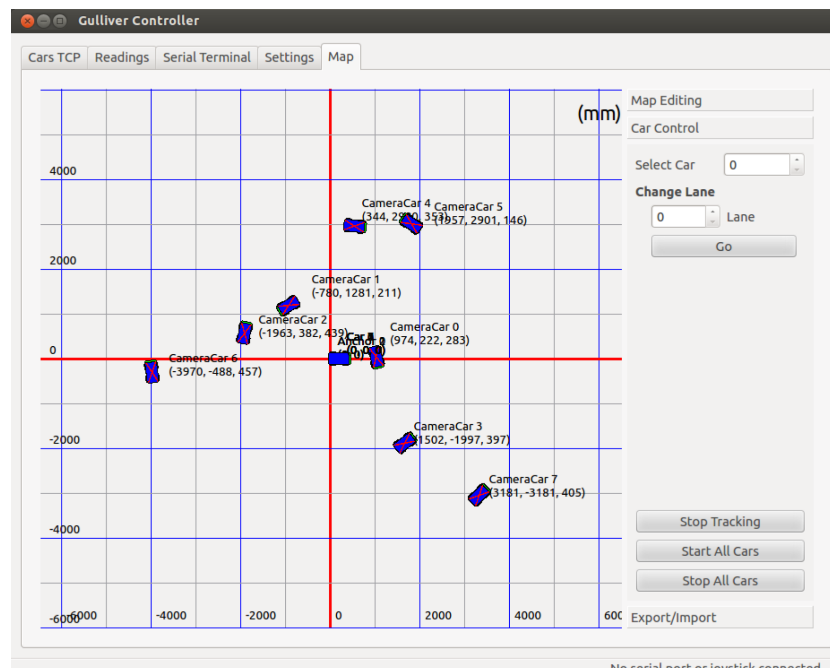
Som tidigare beskrivits använder sig kameran av April-taggar för att identifiera bilarna. Kameran omvandlar varje tagg till ett id-nummer, som dock inte nödvändigtvis är samma som det id-nummer kartklienten använder för att identifiera bilen. För att kunna koppla samman kameradatan med rätt bil behövde därför kartklienten modifieras så

att bilarna kan ha ett kamera-id som manuellt kan skrivas in. Det är detta id som kameran associerar med bilens April-tagg. När systemet skall användas behöver, förutom kamerans id-nummer, även kamera-serverns IP-adress och portnummer anges. För detta skapades fält i fliken “Localization” under “Settings” samt knappen “Track all cars”, som börjar skicka förfrågningar till kamera-servern med ett bestämt intervall.

3.1.4 Testning av systemet

Under projektets gång saknades möjlighet att utföra tester med den riktiga webbkameran då även denna mjukvara var under utveckling. Med hjälp av en specifikation över det slutliga gränssnittet och paketstrukturen kunde dock kameran simuleras och utveckling ske parallellt. Simuleringen av kameran gjordes med hjälp av ett Python-skript som agerade server för att ta emot kartklientens förfrågningar och svara med paket på samma format som var givet i specifikationen (se bilaga A.1 för ett utdrag ur koden). I figur 3.2 visas en körning från en simulering med detta skript där bilarna kör enligt cirkelformade rutter med olika radier.

När kameramjukvaran var färdigställd utfördes även tester i den riktiga Gullivermiljön där delprojektets funktionalitet validerades. Proceduren för detta beskrivs i avsnitt 4.1.2.



Figur 3.2: Testning av kameralokalisering med en Python-server som producerar positionsdata för 8 simulerade kamera-bilar.

3.2 Förbättring av kartklienten

Tillsammans med Gulliverprojektets ledning gick gruppen igenom de olika delarna av kartklienten för att hitta förbättringsmöjligheter. Flaskhalsar i systemet undersöktes och förslag på funktioner som skulle kunna lösa dessa problem togs fram. De funktioner som antogs var: ladda och spara konfigurationer, skicka inställningar till alla bilar och ta bort bilar samt byta färg på dem.

3.2.1 Spara och ladda konfigurationer

Målet med denna uppgift var att implementera ett stöd för att spara och ladda alla inställningar och rutter i kartklienten med bara ett knapptryck. Dessa inställningar kunde tidigare bara sparas som XML-filer men inte laddas. Kartklientens olika inställningar sparades från olika delar av gränssnittet, i separata filer, och det saknades möjligt att välja var dessa inställningar skulle sparas i datorn.

Inledningsvis behövde användaren få möjlighet att välja var inställningarna ska sparas. Detta löstes med en dialogruta där användaren kan välja namn och plats för XML-filen. Eftersom inställningarna ursprungligen sparades i flera olika filer behövde man länka ihop funktionerna i koden där de olika inställningarna sparas, och då metoden för detta skiljde sig åt på olika platser i koden standardiserades sparandeprocessen. För detta användes ett verktyg kallat QXmlStreamWriter, från biblioteket Qt, som skriver XML-kod med angivna parametrar till en fil. Detta såg till att all XML-kod skrevs ut på samma sätt, vilket var speciellt viktigt då dessa filer sedan skulle läsas av automatiskt vid laddning, samt att alla inställningarna sparades till samma fil. Detta krävde att XML-strukturen ändrades för att kunna göra den mer omfattande och utbyggbar. För att därefter kunna ladda de sparade inställningarna på ett snabbt och effektivt sätt användes verktyget QXmlStreamReader.

3.2.2 Ta bort bilar

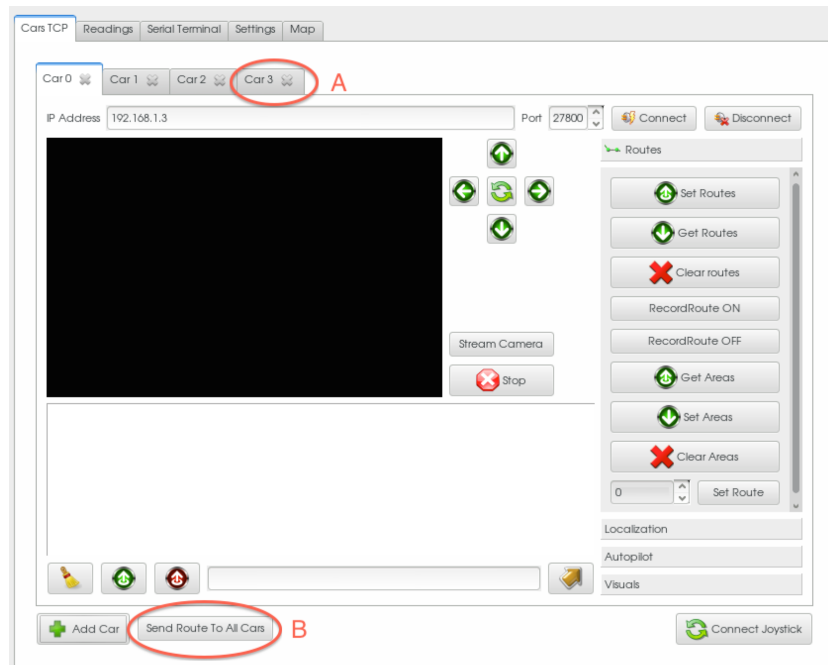
För att göra kartklienten mer lättanvänd önskades en funktion som gjorde det möjligt att ta bort bilar ur kartklienten. När man tidigare ville ta bort bilar var man tvungen att starta om kartklienten, vilket gjorde att även andra inställningar gick förlorade.

Detta löstes genom att lägga till en knapp i bilens flik under “Cars TCP” vilket visas i figur 3.3. För att implementera den nya funktionen och samtidigt behålla kompatibilitet med kartklientens övriga funktioner behövdes dock hänsyn tas till hur hanteringen av bilarna såg ut och användes, vilket medförde en del komplikationer.

Det underliggande problemet var att bilarna identifieras efter sin plats i en lista. Om exempelvis en bil i mitten av listan togs bort, skulle identifikationen av de bilar som låg efter i listan bli fel och programmet sluta fungera. Ett alternativ hade varit att byta ut

den befintliga datastrukturen och identifiera bilarna på något annat sätt än deras plats i listan. Detta hade dock inneburit att mycket kod hade behövts skrivas om, vilket hade varit väldigt omfattade.

För att behålla kompatibilitet med de övriga funktionerna så tilläts borttagna bilar istället ligga kvar med en märkning om att de var borttagna. När programmet sedan söker efter bilen kontrolleras först om bilen är märkt som borttagen eller ej.



Figur 3.3: Vy över kartklientens bilhantering. (A) visar funktionen för att ta bort bilar och (B) visar funktionen för att skicka samma rutt till alla bilar.

3.2.3 Skicka samma rutt till alla bilar

Med syftet att minska uppstartstiden vid testning önskades en funktion för att kunna skicka samma rutt till alla bilar. Tidigare hade man behövt gå in på varje enskild bil och lägga till samma rutt vilket var tidskrävande och ointressant ur testsynpunkt.

Målet var att man genom bara en knapptryckning skulle kunna skicka all nödvändig information till testets deltagande bilar. Knappen, som fick namnet "Send Route To All Cars", placerades under fliken "Cars TCP" vilket visas i figur 3.3. Den nya implementationen stödjer fortfarande att man kan skicka rutter till enskilda bilar.

I implementationen kontrollerar programmet listan med bilar och om bilen inte är märkt som borttagen så skickar programmet rutten till bilen med hjälp av funktionen "Set Routes".

3.2.4 Byta färg på bilar

Då man kör testsessioner med flera bilar kan det lätt bli rörigt att skilja bilarna åt i kartklienten då de ser likadana ut. Syftet med denna uppgift var att implementera stöd för att låta användaren ändra färg på individuella bilar i kartklienten. Detta utfördes genom att skapa en knapp där användaren kunde välja färg på den bil vars flik i gränssnittet de var inne på. För detta användes en dialogruta där användaren kunde välja färg, antingen genom att klicka med musen eller skriva in önskad färgs RGB-kod.

3.3 Atomic Broadcast

För att kunna utvärdera och analysera algoritmerna Paxos och UDP Broadcast som möjliga algoritmer för kommunikation mellan Gulliverbilarna behövdes en miljö som liknade Gullivers testmiljö i så stor utsträckning som möjligt sättas upp och ställas in. Bilarna testas i Chalmers lokaler där det kan uppstå störningar från andra trådlösa nätverk och därför utfördes även prestandatesterna i denna miljö. För att kontrollera nätverkets prestanda vid optimala förhållanden utfördes även tester i en störningsfri miljö.

3.3.1 Problem med flera förslagsställare

Under de inledande testerna av Paxos uppstod en del oförklarliga beteenden, bland annat dubbletter av överenskomna värden samt korrupta sekvensnummer. Efter att ha felsökt den egenutvecklade testimplementationen utan att ha funnit några fel kontaktades utvecklaren av det underliggande biblioteket LibPaxos. Det framkom då att implementationen saknade stöd för användning av flera förslagsställare. Detta var dock inte något som skulle implementeras inom en snar framtid vilket innebar att testningen fick begränsas till att använda sig av en förslagsställare. En nackdel med att använda endast en förslagsställare är att systemet blir mer känsligt för fallerande noder.

3.3.2 Parametrar för utvärdering

För att utvärdera algoritmernas prestanda genomfördes ett stort antal tester. Testerna utfördes i flera olika miljöer, och de parametrar som varierades var: loop-fördröjning, nyttolast, salvstorlek, antal noder och transportmedium.

Loop-fördröjningen, eller fördröjningstiden mellan sändningen av varje paket, innebär i praktiken hur ofta en överenskommelse skulle kunna nås. Denna tid önskas hållas så låg som möjligt, och i Gulliverprojektet har tiderna 10 och 25 millisekunder valts ut. Nyttolasten anger storleken på paketen som skickas och har satts till 128, 256, 512 och

1024 bytes. Det är ännu inte bestämt vilken storlek Gulliverbilarna kommer använda sig av, men för att täcka ett rimligt intervall användes dessa värden.

I ett försök att minska förlusten av paket kan varje paket skickas flera gånger, vilket bestäms av salvstorleken, som sattes till antingen 1, 3, 5 eller 7 paket. Detta gäller endast UDP Broadcast, då Paxos garanterar leverans av alla paket. Antalet noder sattes till 2, 3 eller 4 på grund av hårdvarubegränsningar. De transportmedium som undersöktes var Wi-Fi på både 2,4 och 5 GHz samt Ethernet. För att kunna testa alla möjliga kombinationer av dessa parametrar sattes en databas upp där all testdata kunde sparas. Testen skedde i sessioner, och varje session sparades separat i databasen. På detta sätt kunde grafer och annan data genereras enkelt, flera gånger, utan att behöva köra om testerna.

För sammanställning av data från flera olika testkörningar och datorer så krävdes ett effektivt sätt att samla och kategorisera data samt göra den tillgänglig för vidare analys. I projektet sattes en webbserver upp med hjälp av Python och ramverket Flask för att ta emot och spara data i en databas, analysera och generera grafer samt intressant statistik över respektive experiment. I bilaga A.2 samt A.3 visas intressanta utdrag av koden som användes för detta.

3.3.3 Utformning av testmiljö för Paxos

Genom att utgå från den redan existerande exempelkoden som finns i LibPaxos kunde ganska snabbt en testimplementation färdigställas där grundläggande kommunikation över nätverket kunde genomföras.

Denna testimplementation blev en bas för de modifikationer som senare gjordes av koden för att bygga ett testprogram. Detta program består av ett huvudprogram som startar två trådar med ett nodprogram som utför de olika rollerna i Paxos-algoritmen och ett klientprogram vars uppgift är att skicka värden till förslagsställaren med en bestämd fördröjning (så kallad “loop-fördröjning”) och att logga när dessa värden blir antagna av algoritmen. På så sätt kan tiden det tar att komma överens om värden mätas och analyseras. I bilaga A.4 respektive A.5 finns de intressanta styckena från implementationen av nod och klient.

3.3.4 Utformning av testmiljö för UDP Broadcast

Under inledningen av arbetet utvecklades ett enkelt Python-skript för sändning av UDP-paket mellan flera datorer för att se att allt fungerade som önskat. Därefter implementerades stöd för variabla parametrar som nyttolast, salvstorlek och loop-fördröjning. När systemet fungerade och det var dags att köra en stor mängd tester där de olika parametrarna skulle varieras, automatiserades processen genom användning av ett Python-skript för kontroll av de deltagande datorerna.

3.3.5 Tidssynkronisering

Då majoriteten av de tester som genomfördes handlade om tidsmätning på millisekundnivå så var det av stor vikt att de olika datorernas klockor var synkroniserade i så stor utsträckning som möjligt. Genom att använda en NTP-server kunde en av testets medverkande datorer agera server medan de andra agerade klienter som skickade förfrågningar om den aktuella tiden och justerade sin klocka efter detta, och under majoriteten av testerna automatiserades denna procedur med hjälp av ett skript.

Då leveranstiden för ett paket kunde vara så låg som under en millisekund vid användning av Ethernet som transportmedium syntes dock indikationer på att tidssynkronisering med NTP inte var tillräckligt precis. Eftersom arbetets fokus ligger på kommunikation över trådlöst nätverk ansågs detta acceptabelt då syftet med dessa tester snarare var att testa routerns och algoritmens generella prestanda än de exakta värdena.

3.3.6 Testning av Paxos

För att utföra testerna med Paxos kopplades de deltagande datorerna samman över ett gemensamt lokalt nätverk. Detta nätverk var antingen trådlöst, med en router eller mobiltelefon som AP, eller trådbundet med en switch. Innan testet startades synkroniserades datorernas klockor mot en NTP-server som kördes på en av de deltagande datorerna. Därefter startades Paxos-klientprogrammet på de olika datorerna samtidigt och testet kördes igång. Under testet loggade programmet hela tiden tidsstämplar för varje paket som skickades och togs emot. Dessa loggar samlades sedan ihop och skickades med ett skript till den dator som körde webbservern som hanterade datan och genererade grafer. Alla medverkande datorer kunde sedan se resultatet av körningen i en vanlig webbläsare.

3.3.7 Testning av UDP Broadcast

Testningen av UDP Broadcast utfördes på nästan samma sätt som för Paxos. På grund av det stora antalet möjliga kombinationer av parametrar med introduktionen av salvor, användes ett helautomatiserat skript för längre testsessioner.

3.3.8 Breddning av tester

Då trådlösa nätverk är känsliga för många olika typer av störningar kördes tester över flera nätverkskonfigurationer där både hårdvara och fysisk miljö varierades. Följande redogörelse förklarar hur dessa olika konfigurationer såg ut och varför de valdes ut.

För att ha referenstester att jämföra med samt säkerställa att routerns prestanda för paketförmedling inte skulle bli en flaskhals under testningen kördes även tester över

Ethernet.

I ett försök att undvika störningar som existerar på den vanligaste frekvensen 2,4 GHz samt för att få en ökad bandbredd kördes även tester med USB-nätverkskort som stödjer 5 GHz. Då det i fordonsindustrin finns ett dedikerat frekvensband, vilket används i V2V, kördes också tester på avlägsna platser utomhus i en miljö som skulle efterlikna detta där externa störningskällor minimerats. Eftersom tillgång till strömkälla saknades användes en trådlös accesspunkt i form av en mobiltelefon, vilket innebär att prestandan på det trådlösa nätverket sänktes i förhållande till en traditionell AP, men då det väsentliga var att undvika störningar var denna prestandaförlust acceptabel. För en rättvis jämförelse utfördes även dessa tester med samma AP i Chalmers lokaler.

4 Testresultat och validering

I detta kapitel redovisas resultat för de tre delprojekten. För kameralokaliseringen har funktionaliteten validerats genom ett test i en verklighetstrogen miljö. Förbättringarna av kartklienten har testats vid körningar med bilarna för att kontrollera funktionaliteten. Vad gäller Atomic Broadcast har syftet med projektet i sig varit att utföra tester av algoritmernas prestanda och resultaten av dessa tester visas i slutet av detta kapitel.

4.1 Kameralokalisering

Under arbetets gång utfördes kontinuerligt simulerade tester med hjälp av ett Pythonskript där bilarnas positioner simulerades. Detta gjorde det lätt att se när bilarna ritades upp på ett korrekt sätt. För att verifiera hela systemet utfördes ett test med riktiga Gulliverbilar, vilket beskrivs i avsnitt 4.1.2. För att säkerställa att datan kommer fram tillräckligt snabbt undersöktes leveranstiden för positioneringsdatan som kan ses i kommande avsnitt.

4.1.1 Nätverksprestanda

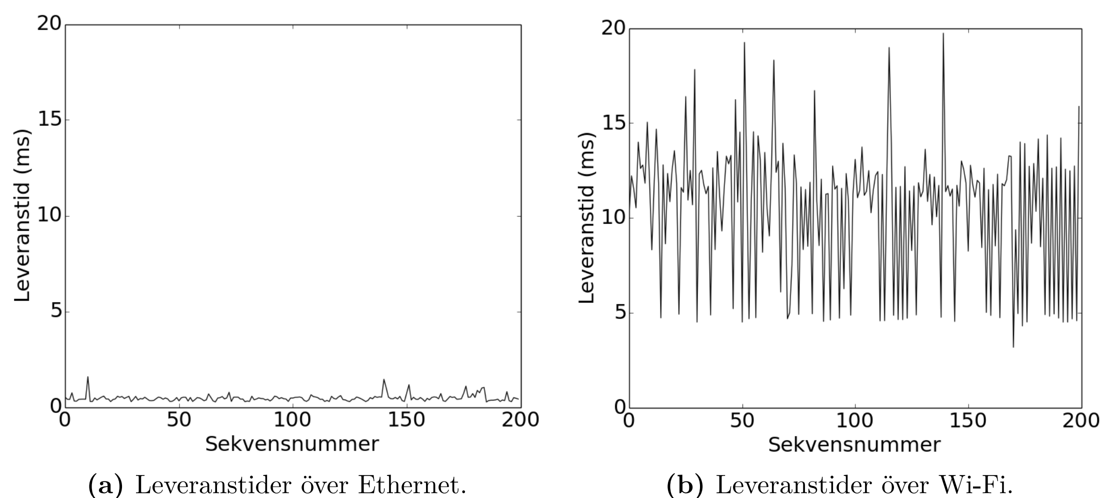
Under de experiment som utförts med Atomic Broadcast har det tydligt visat sig att en kritisk punkt för prestandan över det trådlösa nätverket är hur mycket och hur ofta paket skickas samt de störningar som förekommer. Med detta i åtanke utfördes tester för att kontrollera leveranstiden för positioneringsdatan från kameran. I figur 4.1 visas resultat från dessa test, med 200 skickade paket både över Wi-Fi och Ethernet, där man ser att leveranstiderna håller en stabil nivå under 20 millisekunder för Wi-Fi och under 2 millisekunder för Ethernet. Den nyttjade bandbredden är linjärt beroende av antalet bilar enligt ekvationen

$$b = \frac{h + k * n}{u}, \tag{4.1}$$

där b är bandbredden, h paketets header-storlek (konstant 40 bytes), k datastorleken per bil (konstant 16 bytes), n antalet bilar samt u uppdateringsfrekvensen.

4.1.2 Validering av kameralokalisering

För att säkerställa att systemet fungerade som önskat genomfördes ett valideringstest genom att koppla ihop systemet med webbkameran samt dess mjukvara för bildigenkän-



(a) Leveranstider över Ethernet.

(b) Leveranstider över Wi-Fi.

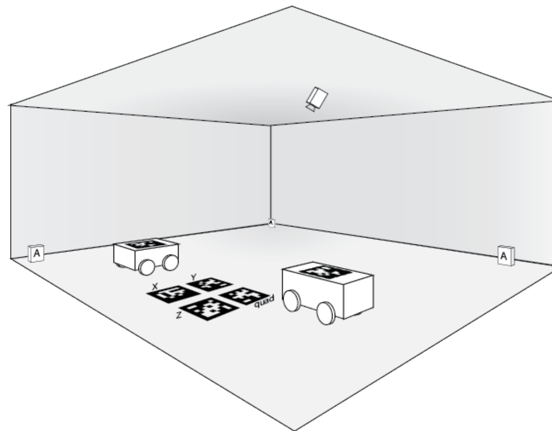
Figur 4.1: Tiden det tar från att en förfrågan skickas till att datan finns tillgänglig för kartklienten över olika transportmedium.

ning, och en testkörning med de fysiska bilarna genomfördes.

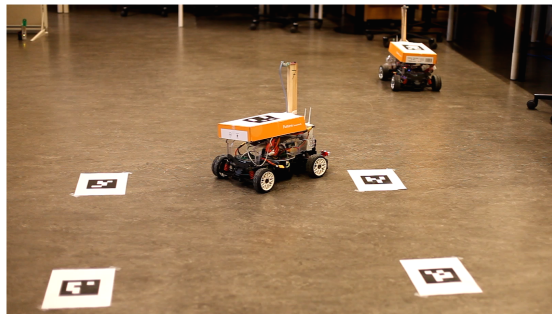
Inledningsvis behövde kamerans koordinatsystem definieras. Detta gjordes genom att sätta fast fyra April-taggar som representerar origo, x-axeln, y-axeln och quad (lutningen på planet) på golvet vilket illustreras i form av en skiss i figur 4.2.

Därefter mättes avstånden från April-taggarna till positionsankarna ut och dess koordinater beräknades och matades in i systemet för att den nya lokaliseringsdatan skulle gå att jämföra med det existerande lokaliseringssystemet. För att bilen skulle kännas igen av webbkameran så placerades en April-tag på bilens tak, vilket kan ses i figur 4.3. Webbkameran placerades i taket för att kunna överblicka en så stor del av testområdet som möjligt.

När kamerasystemet var förberett startades en körning med en bil som till en början bara använde sig av det redan existerande lokaliseringssystemet. Efter att ha kopplat på det nya systemet genom att använda knappen "Track all cars" visades ytterligare en markör i form av en bil markerad med ett kryss, som representerade kameralokaliseringen och man kunde se att bilens position uppdaterades korrekt i kartklienten med låg fördröjning och med en stabil uppdateringsfrekvens. I figur 4.4 ses hur kartklienten ritar ut positionsdatan från kameran tillsammans med datan från det äldre positioneringssystemet. Som synes i figur 4.4b åker bilarna som representerar kameralokaliseringen fram på sidan. Detta beror på att kamerans mjukvara ännu inte tar hand om vilken riktning bilarna kör i, även om stöd för det är implementerat i kartklienten.



Figur 4.2: Skiss på hur en uppriggad testsession ser ut med en webbkamera i taket, tre stycken ankare markerade med (A), fyra stycken April-taggar i golvet för att definiera koordinatsystemet samt två stycken Gulliverbilar utrustade med April-taggar för identifikation.



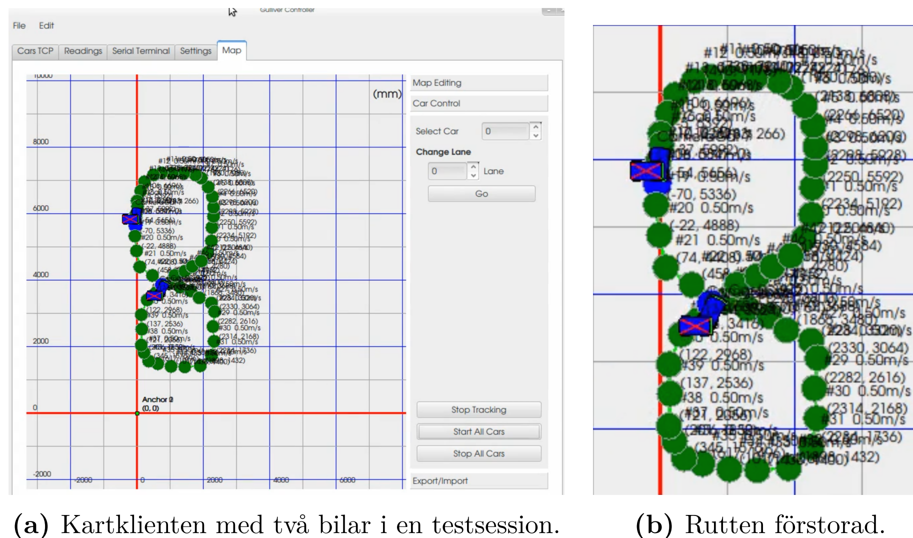
Figur 4.3: Gulliver-bilar utrustade med April-taggar. April-taggarna gör att webbkameran kan lokalisera bilen med hjälp av bildigenkänning.

4.2 Tidsvinster med kartklientens utökade funktionalitet

I detta avsnitt redogörs för de tidsbesparingar som uppnåtts genom den utökade funktionalitet som lagts till i kartklienten. Att minska uppstartstiden vid testning var primärmålet för uppdateringen av kartklienten, då detta resulterar i att användaren kan fokusera mer på testning än konfigurering av systemet.

4.2.1 Spara och ladda konfigurationer

Ofta vill man testa och visa upp systemet i olika rum och testmiljöer. Denna funktion ger användaren en möjlighet att ha olika inställningar för olika rum och snabbt rigga upp systemet i den lokal man ska använda systemet i. Figur 4.5 visar tre flödesdiagram som symboliserar förbättringen av funktionaliteten. Man kan se att första gången en rutt



(a) Kartklienten med två bilar i en testsession. **(b)** Rutten förstorad.

Figur 4.4: Kartklienten under testkörning med kameralokaliseringen. De blå bilarna representerar det befintliga lokaliseringssystemet och de blå bilarna med ett rött kryss representerar kameralokaliseringen.

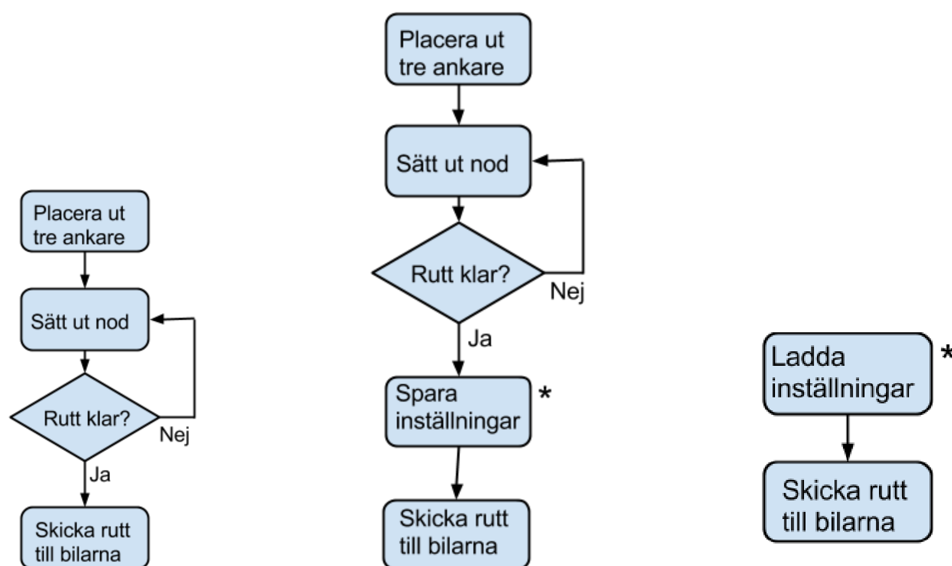
ritas upp görs detta på samma sätt som tidigare med skillnaden att rutten sparas då den skapas. Om samma rutt ska användas igen kan de gamla inställningarna laddas in vilket visas i figur 4.5c. Noderna markerade med “*” syftar på de nya funktionerna.

Att de sparade filernas storlek inte drastiskt skulle öka med antalet noder som sparades var också viktigt. I Gulliverbilens mjukvara finns i nuläget en inprogrammerad gräns på två rutter à 100 noder. Detta betyder att kartklienten kan behöva lagra så många som 200 noder per bil, vilket i framtiden skulle kunna innebära ett tusental noder.

Genom att skapa filer med olika antal noder och sedan läsa av deras storlek hittas samband mellan filstorlek och antal noder. Filerna sträckte sig från att innehålla inga noder alls upp till att innehålla 100 000 noder. Resultat presenteras i figur 4.6a. För att sedan hitta ett samband mellan tid att ladda och antal noder laddades dessa filer in i programmet igen. Tiden det tog för programmet att ladda filen lästes av genom att starta en timer när programmet började ladda filen och sedan läsa av tiden som fortlöpt när programmet laddat färdigt filen. På grund av att tiden det tar för programmet att ladda samma fil varierar, laddades varje fil hundra gånger var och ett medelvärde räknades ut. Figur 4.6b visar resultatet från detta test.

4.2.2 Skicka samma rutt till alla bilar

I nuläget finns det totalt åtta stycken Gulliverbilar. För att skicka en rutt till alla dessa hade man innan denna funktion lades till behövt utföra tre moment i kartklienten per bil. Nu kan man istället göra detta med endast ett knapptryck. I kombination med den

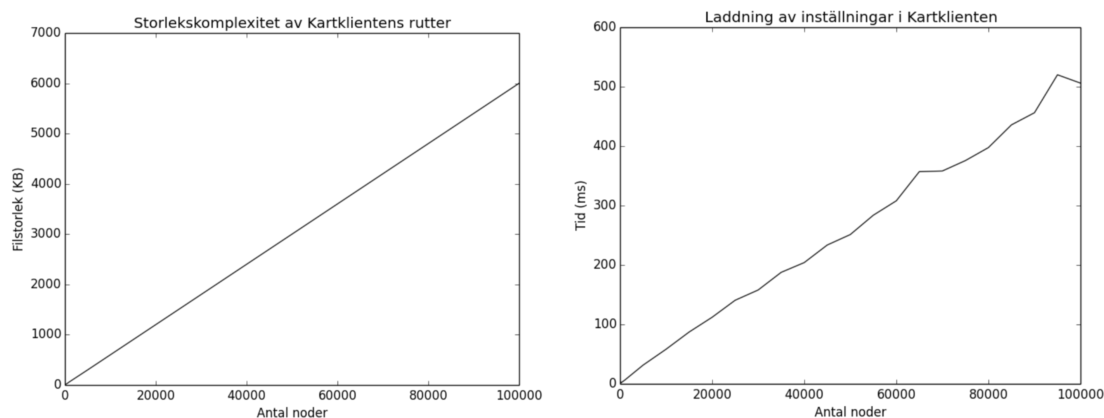


(a) Före implementationen.

(b) Efter implementationen vid första användningen av en rutt.

(c) Efter implementationen vid användning av tidigare sparad rutt.

Figur 4.5: Flödesscheman för att ställa in systemet och återanvända inställningar. Varje nod representerar ett moment som behöver utföras i kartklienten.



(a) Hur mängden noder påverkar storleken av den sparade XML-filen.

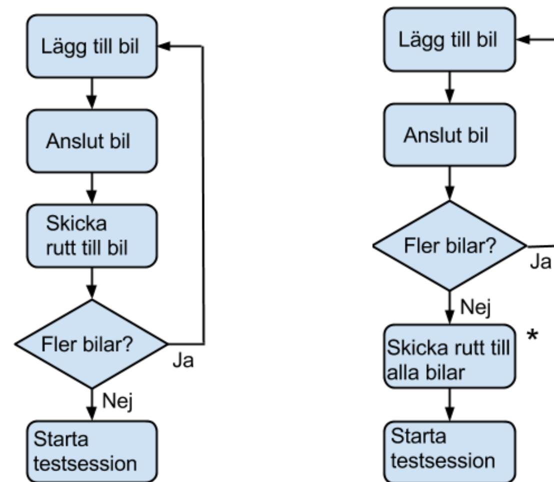
(b) Hur inladdningstiden påverkas av antal noder i XML-filen.

Figur 4.6: Nodantalets påverkan på prestandan.

nya funktionen för att spara och ladda inställningar, beskriven i avsnitt 4.2.1, kan man nu ha alla sina rutter sparade på datorn och snabbt skicka dessa till alla aktuella bilar. Detta gör funktionen till ett kraftfullt och tidssparande verktyg då man arbetar med kartklienten.

Om Gulliverprojektet växer och fler bilar används så skulle den tidigare lösningen varit svår att arbeta med då alla kommandon hade behövt utföras ett stort antal gånger för att alla bilar skulle få den aktuella rutten. Med hjälp av denna funktion kan man lätt använda sig av fler bilar utan att detta blir en flaskhals. Funktionen medför således ett mer skalbart system och möjliggör fler och mer omfattande tester.

I figur 4.7 kan man se två flödesscheman som visar den förbättring funktionen har bidragit till i programmet. I figur 4.7b kan man se att ett moment flyttats ut ur loopen vilket innebär att användaren slipper utföra tre stycken moment för varje varv. Den nod i figur 4.7b som är markerad med "*" syftar på den nya funktionen.



(a) Före implementationen. **(b)** Efter implementationen.

Figur 4.7: Flödesscheman för att skicka rutter till alla bilar där man i (b) kan se att ett moment har flyttats ut ur loopen. Varje nod representerar ett moment som behöver utföras i kartklienten.

4.2.3 Total tidsvinst

När man tidigare skulle rigga upp en testsession tog detta cirka 20 minuter. Man var även tvungen att ha vissa rutter sparade på bilarna för att inte förlora de som tidigare skapats. Efter testning och utvärdering tillsammans Gullivers ledning observerades det att förberedelsetiden för det förbättrade systemet sänkts till cirka fem minuter.

4.3 Validering av kartklientens utökade funktionalitet

Kartklientens förbättringar har resulterat i ett mer lättanvänt verktyg där användbara funktioner implementerats i syfte att underlätta för användaren att testa och vidareutveckla Gulliver. Kombinationen av förbättringarna innebär att komplexiteten av hela

systemet minskat vilket gör det mer lätthanterligt för användare som inte är så bekanta med systemet. För att validera de nya funktionerna genomfördes tester som följde en tydlig arbetsgång. Denna arbetsgång är beskriven i respektive avsnitt.

4.3.1 Spara konfigurationer

Det som gjordes för att validera denna funktion var:

1. Skriva in ett antal inställningar.
2. Rita ut rutter.
3. Trycka på "Save to file"-knappen.
4. Välja plats i filsystemet och bestämma namn med filändelsen .xml.
5. Öppna sparad fil och jämföra med avsedda inställningar och rutter.

Vid valet av inställningar som skrivs in i steg ett var det viktigt att testa inställningarnas begränsningar gällande tecken, formatering och längd, och samtidigt testa olika kombinationer av dessa för att göra testerna så utförliga som möjligt.

4.3.2 Ladda konfigurationer

Valideringen av denna funktion utfördes efter valideringen av funktionen att spara, som beskrivs i avsnitt 4.3.1 för att kunna använda den fil som då skapades. De steg som genomfördes var:

1. Trycka på "Load from file"-knappen under "Settings"-fliken.
2. Välja den tidigare sparade filen i dialogrutan.
3. Jämföra de inställningar och rutter som laddats med vad som står i den laddade filen.

4.3.3 Ta bort bilar

Funktionen att ta bort bilar ur kartklienten validerades genom att utföra följande steg:

1. Lägga till flera bilar i kartklienten.
2. Starta Python-skriptet som simulerar en webbkamera som datakälla för att få bilarnas position att kontinuerligt uppdateras i kartklienten.
3. Välja olika färger på bilarna för att kunna skilja dem åt.
4. Ta bort bilar i olika följd för att se att alla testfall fungerar.

5. Säkerställa att rätt bil försvinner från kartklienten.
6. Lägga till nya bilar för att se att även detta fungerar.

4.3.4 Skicka samma rutt till alla bilar

För att validera funktionen räckte det inte med att simulera ett test utan funktionen behövde testas direkt på Gulliverbilarna. De steg som genomfördes för att validera denna funktion var:

1. Koppla upp flera bilar på ett nätverk.
2. Skapa en rutt i kartklienten.
3. Trycka på "Send route to all cars"-knappen för att skicka rutterna till alla bilar uppkopplade på nätverket.
4. Koppla in bilarna till en dator och säkerställa att bilarna fått den korrekta rutten.

4.3.5 Byta färg på bilar

Följande steg utfördes för att validera att funktionen fungerade som önskat:

1. Skapa en bil i kartklienten.
2. Klicka på "Visuals"-knappen under fliken för den skapade bilen.
3. (Alt. A) Välja färg genom att klicka på önskad färg i dialogruta.
4. (Alt. B) Välja färg genom att skriva in önskad RGB-kod.
5. Observera bilarna i kartklienten och bekräfta att bilen fått rätt färg.

Dessa steg utfördes ett flertal gånger för att säkerställa att funktionen fungerade med flera bilar.

4.4 Atomic Broadcast

Genom många tester utförda både med UDP Broadcast och Paxos har en stor mängd data samlats i databaser. För att tydliggöra resultatet av dessa tester har ett antal grafer som visar på resultaten från specifika testsessioner valts ut. Med hjälp av dessa kan en slutsats dras om i första hand vilka parametrar som bör väljas för de båda algoritmerna, men även indikationer på vilken av algoritmerna som lämpar sig bäst för Gulliverprojektet.

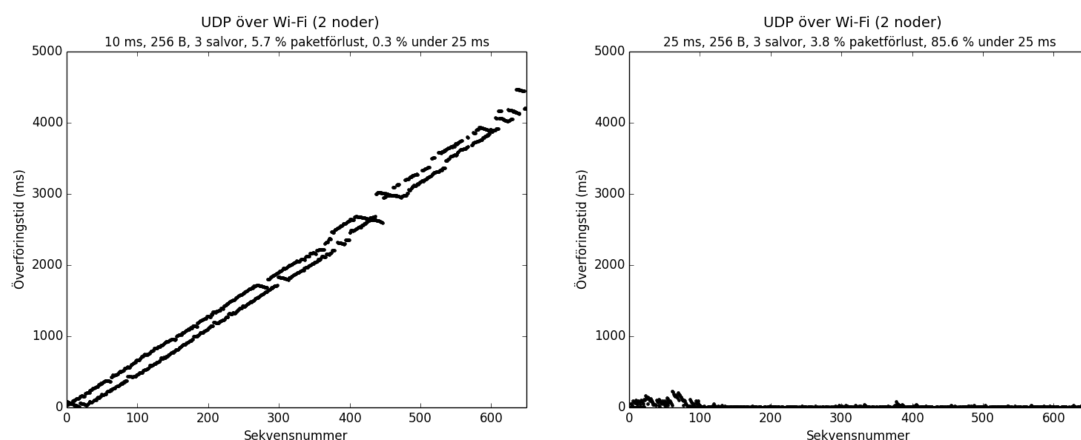
Nedan följer två avsnitt om de båda algoritmerna med utvalda grafer över testernas resultat som visar på hur resultaten ändras med variationer i parametervärden, men

även att det förekommer störningar i den trådlösa kommunikationen. Därför ägnas även ett avsnitt åt tester utförda i en störningsfri miljö. Dessutom finns ett avsnitt där de båda algoritmerna jämförs.

4.4.1 UDP Broadcast

I detta avsnitt följer en redovisning av de viktigaste resultaten från testkörningar med UDP Broadcast, där de tendenser som uppkom i testningen belyses med utvalda grafer. I dessa grafer syns överföringstiden (vertikala axeln) för ett antal paket (horisontella axeln) med varierade parameterinställningar. Överföringstiden definieras som den tid det tar från att paketet skickats till den tiden då alla medverkande noder har mottagit paketet. Om någon av noderna inte tagit emot paketet räknas det som förlorat. Av extra intresse är de paket som klarar överföringstiden 25 millisekunder, vilket är det prestandamål som valts ut (se avsnitt 2.1.2). I grafernas titlar syns statistik över de paket som förlorats samt de som klarar prestandamålet.

Ett önskemål från Gullivers ledning var att använda en loop-fördröjning på 10 millisekunder. Detta visade sig vara problematiskt då många av de tidiga testerna över Wi-Fi (2,4 GHz) resulterade i överföringstider på flera sekunder. Nätverket verkade regelbundet bli överbelastat och mönstret i figur 4.8a återupprepades. Även om det i vissa fall fungerade väl med den lägre fördröjningen beslöts tillsammans med Gulliverprojektets ledning att tidskravet kunde släppas på något och en loop-fördröjning på 25 millisekunder användes istället.



(a) Överbelastat nätverk från en testkörning med en loop-fördröjning på 10 millisekunder.

(b) Testkörning med samma parametrar som (a), förutom att loop-fördröjningen är 25 millisekunder.

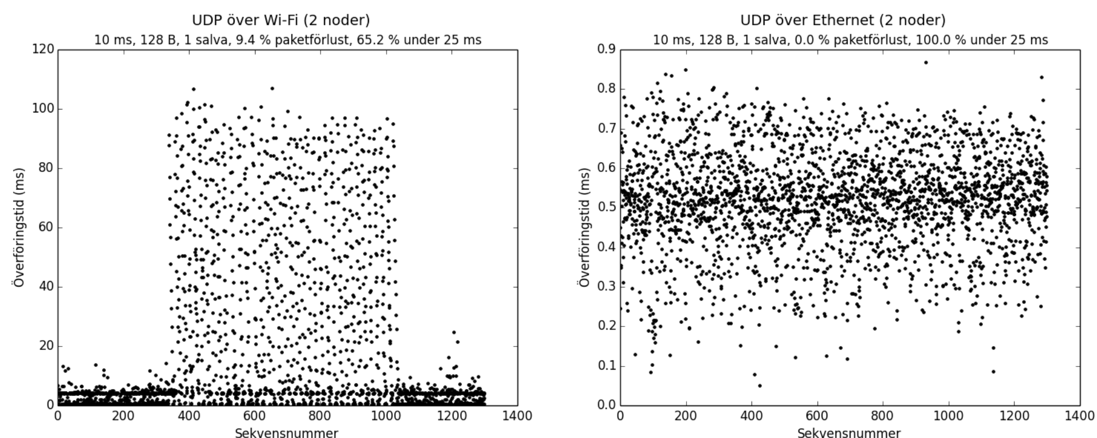
Figur 4.8: Jämförelse mellan olika loop-fördröjningar över Wi-Fi (2,4 GHz).

Konsekvensen av att använda sig av en högre loop-fördröjning i praktiken är att Gulliverbilen får tillgång till information från de övriga bilarna mer sällan och den hinner

således färdas en längre sträcka än vid en lägre loop-fördröjning. Denna sträcka är i detta fall endast marginell, vilket behandlas i avsnitt 5.4.2.

En annan parameter som utvärderades var nyttolasten. Denna parameter varierades under alla tester och det framkom tydligt att dess påverkan är den väntade, det vill säga att prestandan ökar för mindre nyttolaststorlek. I figur 4.13 i avsnitt 4.4.2 visas ett exempel på hur nyttolasten påverkar prestandan för Paxos, och detsamma gäller för UDP Broadcast.

Under de tester som utfördes uppstod ofta fenomenet med störningar över det trådlösa nätverket. Figur 4.9a visar tydligt förekomsten av störningar, vilket exempelvis skulle kunna vara en mikrovågsugn som introducerar störningar på samma frekvensband. Det syns ett stort hopp i leveranstider mellan paket 400 och 1000, men bortsett från denna störning håller de flesta paketen sig under en konstant låg nivå när det gäller överföringstid. I figur 4.9b ses att användandet av Ethernet som transportmedium eliminerar all form av paketförlust samtidigt som överföringstiden håller sig på en konstant låg nivå. Transportmediets resistans mot störningar visar att spridningen av överföringstiderna håller sig på en jämnare nivå än över ett trådlöst nätverk.



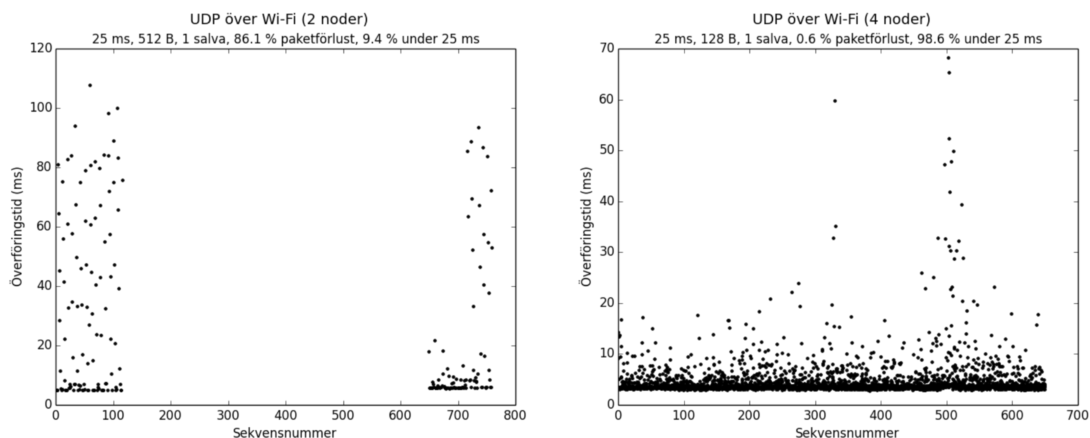
(a) Mellan sekvensnummer 400 och 1000 förekommer en störning som fördröjer paketens leverans.

(b) Med Ethernet är överföringstiderna konstant under 1 millisekund.

Figur 4.9: Överföringstider för Wi-Fi (2,4 GHz) och Ethernet med 128 bytes nyttolast och 10 millisekunders fördröjning.

Det finns också tillfällen då den trådlösa kommunikationen helt kan avbrytas på grund av dåliga förhållanden i det trådlösa nätverket. Ett sådant tillfälle är testet i figur 4.10a, där det under en lång tid i mitten av testet är ett totalt avbrott. Trots detta kan paketströmmen återupptas när avbrottet upphört, och överföringstiderna håller en liknande nivå som före avbrottet. Ett vanligt förekommande resultat vid testning av UDP Broadcast över Wi-Fi utan större störningar visas i figur 4.10b. Den allra största delen av paketen håller sin deadline på 25 millisekunder, men det finns paket som blir försenade,

troligen på grund av mindre störningar i den trådlösa nätverksmiljön.

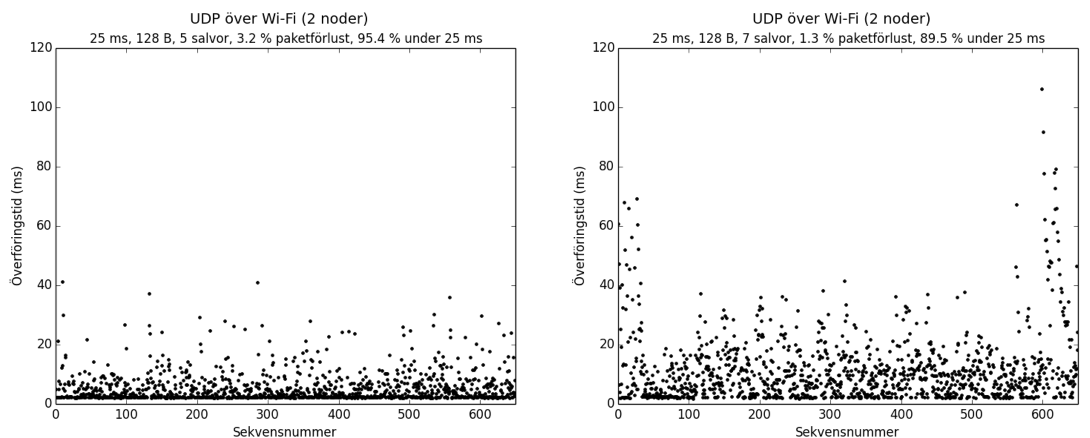


(a) Testkörning över Wi-Fi (2,4 GHz) där en störning helt avbryter paketströmmen.

(b) Testfall över Wi-Fi (2,4 GHz) med 128 bytes nyttolast där små störningar förekommer.

Figur 4.10: Två olika typer av störningar över Wi-Fi (2,4 GHz).

För att visa på hur man skulle kunna sänka paketförlusten vid användning av UDP Broadcast varierades salvstorleken. Då salvstorleken ökar, ökar även sannolikheten för att paketen ska komma fram. Detta påverkar dock endast marginellt vilket kan ses i figur 4.11 och möjliga anledningar till detta diskuteras i avsnitt 5.4.3. Att ändra salvstorleken har visat sig ha större påverkan vid testning över 2,4 än 5 GHz. Däremot har det under testerna noterats att detta skett på bekostnad av överföringstiden, vilken ökar med antalet paket över nätverket.



(a) Salvstorlek 5 visar en viss paketförlust.

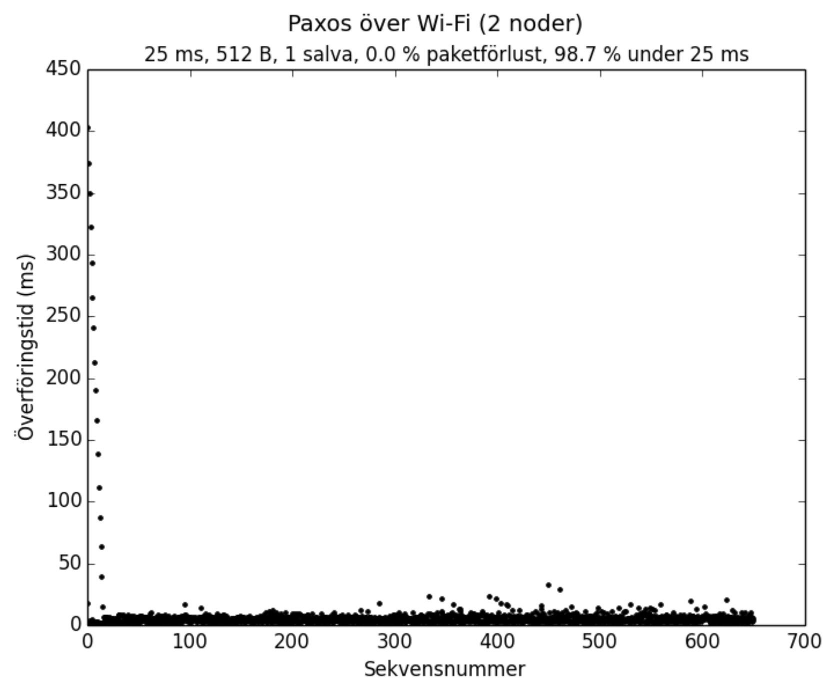
(b) Salvstorlek 7 visar på en sänkning av paketförlusten.

Figur 4.11: Testfall med Wi-Fi (2,4 GHz) där salvstorleken ändras.

4.4.2 Paxos

Under testerna med Paxos märktes liknande tendenser som för testningen med UDP Broadcast då i stort sett samma parametrar varierades. Undantaget var salvstorleken som i fallet med Paxos är ointressant då algoritmen garanterar leverans av meddelanden. De förhållanden som beskrivits i introduktionen till avsnitt 4.4.1 gäller även här.

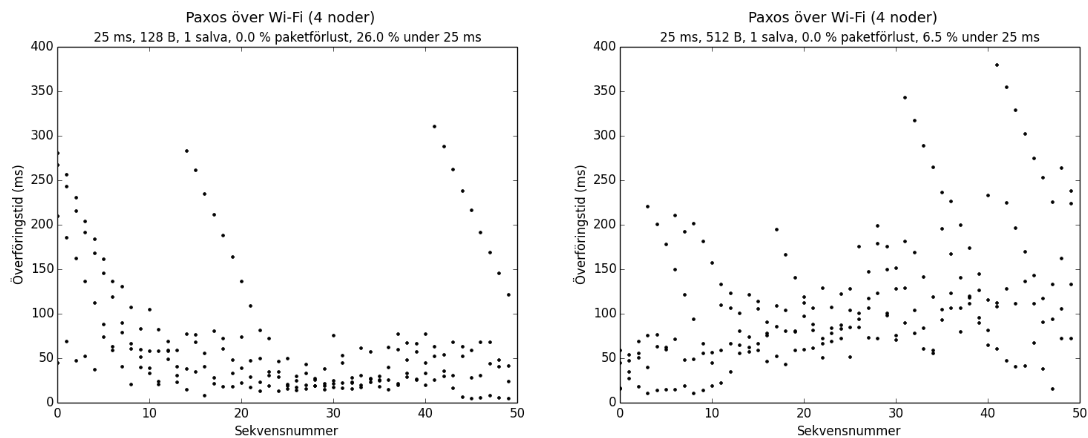
Ett återkommande fenomen för testerna av algoritmen, som var att en av noderna tog mycket lång tid på sig för att få iväg sina paket, visas i figur 4.12. Inledningsvis verkade det som att detta hade att göra med buffring av paket i routern, men allt eftersom testningen fortskred tycktes det mer sannolikt att det underliggande protokollet TCP var orsaken. Detta diskuteras utförligare i avsnitt 5.4.4.



Figur 4.12: Testfall där fenomenet med en hög initial överföringstid för en av noderna uppkommer.

Genom att variera nyttolaststorleken under tester på såväl 2,4 som 5 GHz, kunde dess påverkan på prestandan undersökas. I figur 4.13 ses att en fyrdubbling av nyttolaststorleken medför att många av paketen missar den önskade tidsgränsen på 25 millisekunder. I figur 4.13b kan man även ana en tendens med en ökande överföringstid ju längre testet pågår. Man kan också se hur en ökning av nyttolaststorleken ökar spridningen av paketen och minskar stabiliteten. Testet körs över 2,4 GHz vilket ger ett stort negativt utslag på överföringstiden.

Då samma test utförs över 5 GHz syns i figur 4.14 en tydlig förbättring. En fyrdubbling av nyttolasten vid en körning hanteras klart bättre på frekvensen 5 GHz då betydligt

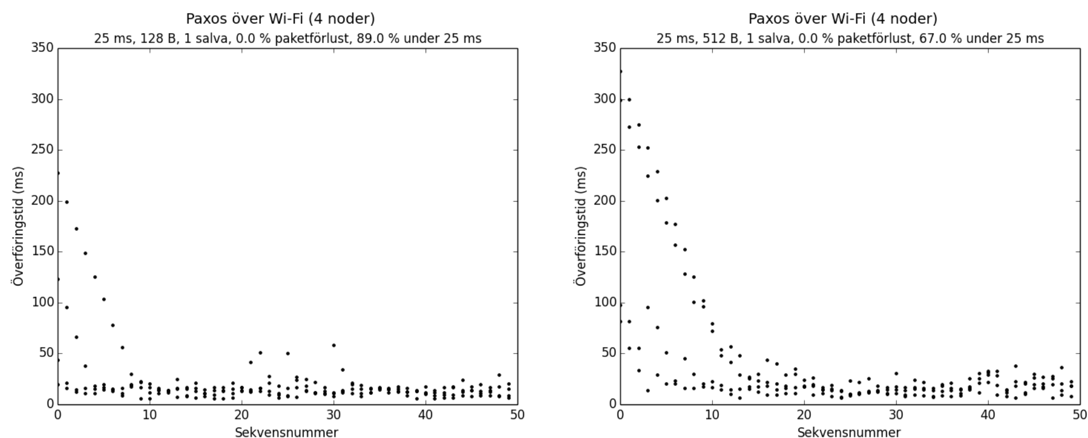


(a) Testfall där det redan vid en nyttolast på 128 bytes visar på höga överföringstider.

(b) Testfall med en fyrdubbling av nyttolasten jämfört med (a).

Figur 4.13: Jämförelse av olika nyttolaststorlekar över Wi-Fi (2,4 GHz).

fler av paketen har en överföringstid under 25 millisekunder, samtidigt som det inte finns några tendenser som talar för en försämring av prestandan över tid. Dessa testfall visar även på något som uppmärksammats under testningen – att körningar över 5 GHz har en konsekvent bättre prestanda.

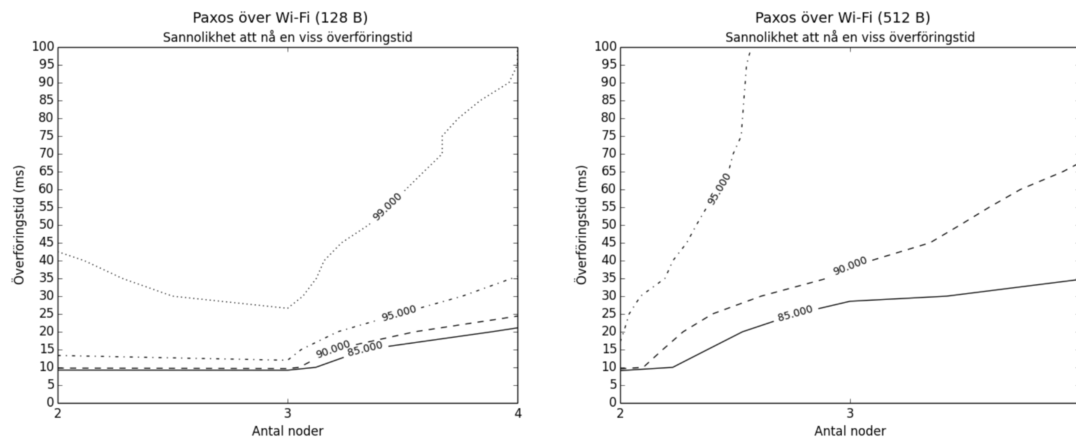


(a) Testfall med en nyttolast på 128 bytes visar på acceptabla överföringstider.

(b) Testfall med en fyrdubbling av nyttolasten jämfört med (a).

Figur 4.14: Jämförelse av olika nyttolaststorlekar över Wi-Fi (5 GHz).

För att sammanfatta hur olika nyttolaststorlekar påverkar prestandan vid en körning av Paxosalgoritmen visas i figur 4.15 en konturgraf där tester med nyttolaststorlekarna 128 och 512 bytes körts med 2, 3 eller 4 noder över Wi-Fi (5 GHz). Genom att följa de olika linjerna ges vald sannolikhet att paketen ska hålla sig under den avlästa överföringstiden vid ett specifikt antal noder.



(a) Sannolikheten att nå en viss överföringstid med en nyttolast på 128 bytes över 2, 3 och 4 noder.

(b) Sannolikheten att nå en viss överföringstid med en nyttolast på 512 bytes över 2, 3 och 4 noder.

Figur 4.15: Jämförelse av sannolikheten att nå en viss överföringstid med en nyttolast på 128 respektive 512 bytes över Wi-Fi (5 GHz).

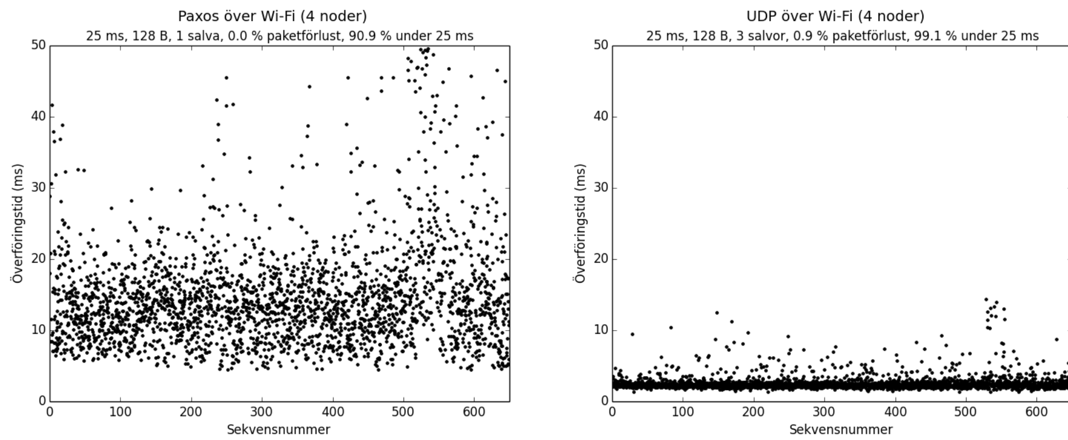
4.4.3 Jämförelse av UDP Broadcast och Paxos

Det stora antalet tester som genomförts har visat på att UDP Broadcast har bättre rå prestanda än Paxos vid likvärdiga inställningar, men att man i princip alltid kan räkna med en viss paketförlust. Ett typiskt exempel som visar detta ses i figur 4.16. Den vänstra figuren är inzoomad för att läsaren tydligare ska kunna göra en jämförelse med figuren till höger (2,46 % av paketen har en överföringstid över 50 millisekunder och syns därför inte i figuren). Överlag har båda algoritmerna visat goda prestanda på Wi-Fi (5 GHz) och vilken som är att föredra diskuteras i avsnitt 5.4.5.

4.4.4 Nätverk utan störningar

För att visa hur stor påverkan de yttre störningarna hade på prestandan visas i detta avsnitt grafer över de resultat som uppnåddes vid körningar över ett nätverk utan störningar.

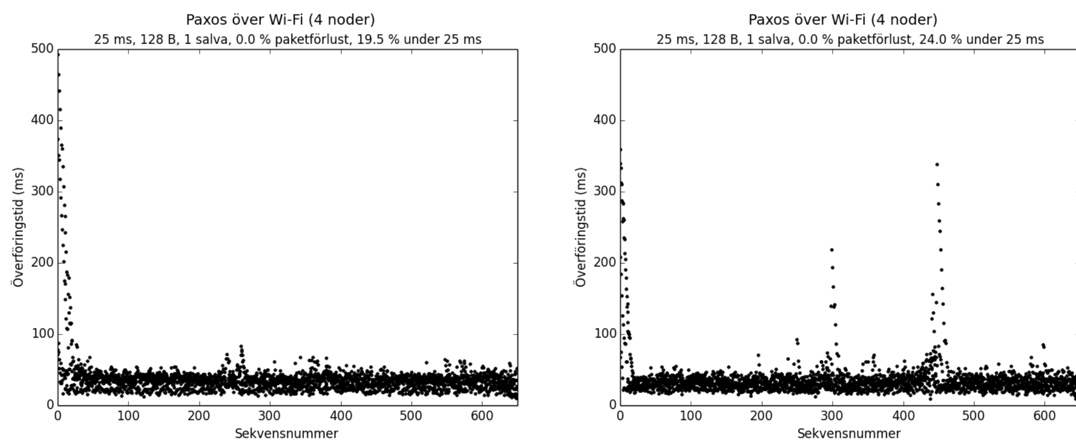
I figur 4.17 visas resultatet från två testfall med identiska parametrar där det ena testfallet genomförts i en miljö med flera kringliggande trådlösa nätverk medan det andra genomförts i en isolerad utomhusmiljö. Figuren visar dock att prestandaskillnaden är försumbar och det är svårt att dra någon slutsats om detta beror på variationer eller andra faktorer. Observera att då en mobiltelefon användes som AP under båda testerna så är prestandan inte jämförbar med övriga grafer.



(a) Testkörning där majoriteten av paketen har en överföringstid runt 15 millisekunder.

(b) Testkörning där majoriteten av paketen har en överföringstid runt 3 millisekunder.

Figur 4.16: Jämförelse av överföringstiden för Paxos gentemot UDP över Wi-Fi (5 GHz).



(a) Testkörning över Wi-Fi med kringliggande trådlösa nätverk.

(b) Testkörning över Wi-Fi utan kringliggande trådlösa nätverk.

Figur 4.17: Jämförelse av prestandan över Wi-Fi med respektive utan kringliggande trådlösa nätverk.

5 Diskussion och slutsats

Detta kapitel beskriver några av de tankar som uppkommit under arbetets gång. Dessutom kommenteras resultat och tillvägagångsätt som varit av speciellt intresse samt de användningsområden delprojektens resultat kan ha i den framtida utvecklingen av Gulliverprojektet. Därefter diskuteras framtida möjligheter och eventuella problem som kan finnas med autonoma bilar. Då autonoma bilar fortfarande är ett nytt fenomen finns det många aspekter inom området som bör beaktas, både vad gäller tekniken som används och de moraliska frågor som uppstår då bilar styrs av datorer. Avslutningsvis sammanfattas de slutsatser arbetet lett fram till.

5.1 Utvärdering av arbetsgång

Arbetet med att förbättra Gulliver har mestadels gått som förväntat även om gruppen har stött på en del hinder på vägen. Många av de problem vi stött på hade kanske kunnat lösas på ett bättre sätt om gruppen disponerat arbetstiden jämnt mellan början och slutet av projektets tidsspann. Ungefär två tredjedelar av de timmar gruppen arbetat har skett under andra halvan av projektet. När arbetsbördan legat på den senare delen i projektet har problem som egentligen borde ha kommit upp tidigare under projektets gång dykt upp senare, vilket har lett till tuffare arbetsveckor och onödiga påfrestningar.

Samtidigt som det varit en trygghet att arbeta med ett etablerat projekt som Gulliver har även vissa svårigheter uppstått, då vi under arbetets gång varit beroende av Gulliverprojektet och behövt arbeta inom dess ramar utan att ha full frihet att komma på helt egna lösningar. Något som har varit ett hinder under arbetets gång har varit den begränsade tillgången till bilarna vilket har minskat möjligheten till testning i en verklig miljö.

En annan aspekt är att de tre olika delprojekten skiljde sig åt i svårighetsgrad och arbetsbörda. Kameralokaliseringen och förbättringarna i kartklienten var uppgifter som liknade sådana gruppen var förhållandevis vana vid vilket ledde till att de gick relativt smärtfritt att utföra. Delprojektet Atomic Broadcast var mer tungt teoretiskt sett och saknade en tydlig lösning, vilket ställde högre krav på vår egna förmåga att ta oss an ett – för oss – outforskat område för att nå vårt mål.

5.2 Kameralokalisering

Projektet med kameralokalisering fungerade som en introduktion till Gulliverprojektet där gruppmedlemmarna lärde sig hur systemet fungerar, hur mjukvaran är uppbyggd samt hur den används. Under utvecklingen av kartklientens integrering med kameralokaliseringen har funktionaliteten validerats både genom kodutvärdering och tester. Under kodutvärderingen påpekades att som en följd av implementationen av möjligheten att ta bort bilar hade en av funktionerna i koden tidskomplexitet $O(n^2)$ på grund av den underliggande strukturen i vilken bilarna lagras i koden. Om denna struktur skrivits om hade funktionen kunnat uppnå tidskomplexitet $O(n)$, men det hade inneburit en stor omskrivning av koden som hade haft effekter utanför delprojektets ramar. Då det endast finns åtta bilar i Gulliverprojektet ansågs detta ändå vara acceptabelt. Med facit i hand hade det kanske ändå varit bra att lägga ner tiden på att göra de mer grundläggande ändringarna för att få ett mer framtidssäkert system.

Tack vare det Python-skript som implementerades tidigt under arbetets gång kunde mycket tid sparas genom tidig testning. Detta var särskilt viktigt då det projekt där kamerans mjukvara utvecklades inte slutfördes förrän en bra bit in i arbetet.

Vid de tester av systemet som utfördes syntes tydligt att transportmediet som används mellan kameran och kartklienten hade stor inverkan på prestandan för lokaliseringen i kartklienten. Eftersom lokaliseringsdatan bara är relevant så länge den är ny är det viktigt att datan kan hämtas från kameran på så kort tid som möjligt. Helst bör inte datan vara mer än ett par millisekunder gammal för att ge en rättvis position. Det är därför rekommenderat att kameran antingen är kopplad direkt till samma dator som kartklienten körs på eller med Ethernet för att säkerställa ett bra flyt i uppritningen och undvika onödig belastning av det trådlösa nätverket där bilarnas kommunikation sker.

5.3 Förbättring av kartklienten

När gruppen började arbeta med detta projekt var medlemmarna inte särskilt insatta i kodbasen som Gullivers mjukvara utgör. Eftersom koden var odokumenterad gick mycket tid åt till att studera koden. Utöver det hade gruppmedlemmarna inte någon tidigare erfarenhet när det gällde programmeringsspråket C++ och Qt vilket innebar en viss inlärningsperiod. Detta har dock lett till att vi har lärt oss mycket nytt och värdefullt om dessa tekniker. Vi har också förstått vikten av att dokumentera koden man skriver och hur det kan effektivisera framtida arbeten avsevärt, speciellt i större projekt som detta.

5.4 Atomic Broadcast

I jämförelsen mellan Paxos och UDP Broadcast har vi bara utfört tester på våra egna datorer som alla har befunnit sig på ett nära avstånd från AP:n. Genom att utföra ett fåtal tester på varierande avstånd upp till 8 meter såg vi att prestandan höll sig på samma nivå, vilket gjorde att vi valde att fortsätta testningen på korta avstånd av praktiska skäl. Med andra avstånd och andra datorer är det möjligt att våra resultat kunnat bli annorlunda. Vi har dock inte fokuserat på dessa problem då vi tyckt att våra testfall har täckt den viktigaste delen av testen för att kunna få en bra helhetsbild.

I syfte att efterlikna användandet av ett dedikerat frekvensband genomfördes en del tester långt bort från andra trådlösa nätverk. Dessa tester utfördes på en mobiltelefon som AP med betydligt lägre prestanda än den AP som användes i den vanliga testmiljön i Chalmers lokaler. Då syftet med testet var just att simulera en störningsfri miljö samt att testerna jämfördes med referenstester utförda med samma typ av AP så tyckte vi att det var en acceptabel lösning i brist på möjlighet att utföra testerna med den AP som annars använts. Det hade dock varit intressant att se tester utomhus även med den vanliga AP:n.

5.4.1 Nätverkets begränsningar

Under de UDP Broadcast-tester som körts har vi maximalt uppnått en nätverksbelastning på ungefär 30 Mbps. Tester har även visat att den största påfrestningen på nätverket kommer från hur många paket som skickas, de paketkollisioner som uppstår samt störningar på nätverket. Med Paxos är nätverksbelastningen ännu högre på grund av den extra synkronisering som krävs. Då 2,4 GHz har ett teoretiskt maximum på 54 Mbps är det därför inte förvånande att det inte klarar av denna typ av belastning. När det gäller 5 GHz borde dock inte dess teoretiska maximum på 867 Mbps vara ett hinder, men testresultat har visat att även denna frekvens har problem att hantera stora mängder paket vilket vi tror beror på störningar samt de paketkollisioner som uppstår.

5.4.2 Loop-fördröjningen

Ökningen av loop-fördröjningen innebar stabilare tester med mer konsekventa resultat, då nätverket inte överbelastades lika lätt, speciellt när 2,4 GHz användes. Detta skedde dock på bekostnad av upplösningen av datan för bilarna. Då tiden ökas från 10 till 25 millisekunder sänks antalet mottagna paket per sekund paket från 100 till 40 stycken vilket innebär äldre data. I Gulliverbilarnas fall är detta dock acceptabelt, då bilen inte kommer mer än ett par centimeter på tiden mellan paketens leverans när den kör i en normal hastighet på runt 1 m/s. Därför lades fokus på att testa algoritmerna främst med en fördröjning på 25 millisekunder.

5.4.3 Salvstorlekens påverkan

Då storleken på salvorna ändrades i testerna av UDP Broadcast var tanken att paketförlusten skulle minska. Det visade sig att detta stämde, men bara med förhållandevis låg effekt, vilket visas i avsnitt 4.4.1. Detta skulle kunna bero på det faktum att alla paket i en salva skickas direkt efter varandra, vilket gör det sannolikt att om ett av paketen i salvan förloras på grund av störningar eller överbelastning, kommer även de närmast efterföljande göra det. Vad gäller överföringstiden kan den till och med komma att påverkas negativt då antalet paket skickade över nätverket ökar, vilket leder till större belastning på nätverket.

5.4.4 Problem med Paxos

Under implementationen av vår testkod för LibPaxos uppkom flera problem. Mycket tid lades på att få LibPaxos att fungera stabilt då många tidiga testsessioner kantades av krascher och instabila resultat. Efter en noggrann genomgång av vår egen del av koden kontaktades skaparen av biblioteket, vilket nämns i avsnitt 3.3.1. I efterhand kan vi se att detta borde gjorts mycket tidigare då testningen först efter detta kunde komma igång på riktigt.

Nackdelen med användningen av endast en förslagsställare är att det inte är lika representativt för den miljö Gulliverbilarna testas i, då en specifik nod alltid måste vara i ett fungerande tillstånd. För slutlig användning av Paxos behöver man antingen hitta en lösning för att välja vilken nod som ska agera förslagsställare alternativt använda sig av en implementation som stöder flera förslagsställare för att få alla de garantier som Paxos levererar.

Ett annat problem var fenomenet med att paketen från en av noderna hade väldigt långa överföringstider i början (se figur 4.12 i avsnitt 4.4.2) som aldrig fick någon riktig förklaring. Det skulle kunna bero på att någon del i systemet startar långsamt och sedan kommer igång och håller en acceptabel hastighet så att en mängd paket levereras nästan samtidigt. Detta skulle förklara den linjära tendens de första paketen för noden visar. Då hårdvaran i andra fall klarar av liknande belastning verkar en mer rimlig anledning vara det faktum att Paxos använder sig av TCP, vilket har funktioner för att begränsa takten i vilken paket skickas för att inte överbelasta nätverket. Innan en slutlig integrering av en Paxos-implementation skulle ske hade man behövt undersöka detta ytterligare och säkerställa orsaken till problemet.

5.4.5 Val av algoritm

För att välja vilken algoritm som bör användas i Gulliverprojektet måste en mängd aspekter tas hänsyn till. Algoritmen UDP Broadcast är enkel att förstå och implementera i sin grundform men dess nackdel är avsaknaden av garantier. I ett säkerhetskritiskt

system så som en förarlös bil bör säkerhet stå i fokus. Genom att skicka samma paket flera gånger kunde antalet förlorade paket reduceras, men aldrig elimineras helt och hållet. Det visade sig också att användningen av salvor kunde försämra överföringstiderna.

Gemensamt för de båda algoritmerna är att de påverkas kraftigt av antalet noder, och om man önskar öka detta kan vissa parameterval behöva ses över. Med fyra noder kan en nyttolast på upp till 256 bytes vara realiserbar med Paxos, medan en storlek på 1024 bytes kan hanteras i UDP Broadcast. Med ännu fler noder kan troligen en mindre paketstorlek hanteras.

När störningar uppstår på ett trådlöst nätverk blir fördröjningen alldeles för hög hos båda algoritmerna. På grund av att Paxos garanterar att ett paket når fram belastas nätverket onödigt mycket vid en störning. När en störning uppstår och paketet blir försenat kommer Paxos ändå att försöka nå fram. Då ett försenat paket många gånger kan likställas med skräp kan denna belastning på nätverket förhindras med UDP Broadcast. Den sistnämnda algoritmen kan alltså återhämta sig snabbare från en störning än Paxos.

Enligt resultatet i figur 4.16a i avsnitt 4.4.3 har cirka 10 % av paketen en överföringstid över 25 millisekunder. Motsvarande test för UDP Broadcast visar cirka 1 % paketförlust (se figur 4.16b). Då algoritmerna drabbas av försenade respektive förlorade paket krävs försiktighetsåtgärder oavsett vilken algoritm som skulle väljas. En fördel med Paxos är att man vet när paketen mottagits och då kan bestämma hur man ska reagera. I fallet med UDP Broadcast finns inte dessa möjligheter utan där måste mottagaren bekräfta att ett paket nått fram. Detta är dock något som inte tagits hänsyn till i vår testning. Paxos garanterar även att paketen kommer fram i rätt ordning, vilket i ett säkerhetskritiskt system kan vara mycket viktigt.

Valet av algoritm landar slutligen i de tidskrav systemet har. De resultat som vi har sett visar på att man skulle behöva tillåta en något högre överföringstid för att Paxos skulle vara ett lämpligt val i Gulliver. Detta skulle kunna vara en bra avvägning med tanke på de garantier som erbjuds men även på grund av den tidsvinst det innebär att använda sig av en existerande implementation som tillhandahåller säker kommunikation mellan bilarna.

Är tidskraven däremot striktare hade UDP Broadcast varit mer lämpligt då den råa prestandan är bättre, men nackdelen är att det kan krävas mycket extra arbete för att komma fram till en likvärdig lösning som tillhandahåller de garantier som Paxos gör. Detta skulle också kunna innebära att prestandan försämras då dessa garantier implementeras.

5.5 Framtida utmaningar för autonoma bilar

Samtidigt som man utvecklar tekniken för autonoma bilar är det viktigt att ta ett steg tillbaka och se till helhetsbilden av dess framtid och vad de kommer att ha för inverkan på samhället. Dessa bilar kommer säkerligen att ha både positiva och negativa implikationer på vår vardag. Förhoppningsvis kommer dock de positiva aspekterna väga upp de negativa.

5.5.1 Juridiska frågor

Många lagar och regler gällande transportfordon är skrivna innan de autonoma bilarnas uppkomst. Ett exempel på detta är UNECEs överenskommelse (Vienna Convention, 1968) då det beslutades att (fritt översatt från engelska) "Varje förare ska alltid kunna kontrollera sitt fordon...". Detta betyder, så länge man följer dessa stadgar, att bilar eller andra fordon alltid måste ha en mänsklig förare med sig som kan ta över kontrollen av fordonet. Det är alltså inte tillåtet för autonoma bilar att köra utan att en förare befinner sig i bilen.

En betydande anledning till att företagen tvivlar på autonoma bilar är att det ofta inte finns några lagar eller regler gällande ansvar vid olyckor. Det kan vara avskräckande för biltillverkarna om de är ansvariga för alla skador som sker på grund av fordonen när de framförs autonomt. I sådana fall skulle det med stor sannolikhet inte bli lönsamt för dessa företag att sälja autonoma bilar. Å andra sidan är konsumenter kanske inte särskilt benägna att köpa autonoma fordon om de skulle vara ansvariga för olyckor som sker när de inte kontrollerar fordonet själva (Pinto, 2012).

5.5.2 Etiska frågor

Det finns också många frågor man kan ställa sig när det gäller etiken i att låta autonoma bilar köra i trafiken tillsammans med andra biltrafikanter, fotgängare och cyklister. Vi vet att människor som vanligtvis kör i trafiken innehar ett mått av medkänsla och oftast kan avgöra vad som är rätt och fel och då väga fördelar och nackdelar i ett specifikt agerande (Lin, 2013). Att bryta mot trafiklagar är kanske olagligt men inte alltid omoraliskt. En människa skulle kunna avgöra när det är acceptabelt att bryta mot trafiklagar för att kunna undvika något mycket värre, till exempel om ett barn skulle springa ut på en väg där filbyte är förbjudet skulle en mänsklig förare kanske köra ut i filen bredvid för att väja för barnet och för att slippa en tvärbromsning som kanske riskerar en krock med bilar bakom.

En annan diskutabel aspekt som uppstår i och med att man låter datorer kontrollera våra bilar är risken för att bilen blir hackad och sedermera kontrollerad av hackaren i fråga. I teorin finns inga datorer som inte går att hacka så länge de kommunicerar eller

är beroende av källor utifrån. På så sätt skulle hackare kunna ta kontroll över bilarna och orsaka stora skador mot både människor i och utanför bilen. Biltillverkarna säger att de arbetar för att förhindra hackare men frågan är om det någonsin kommer att vara helt säkert (Krisher, 2013).

5.6 Slutsats

De förbättringar vi har bidragit med till Gulliversystemet kommer underlätta framtida testning med systemet. De nya funktionerna har lagt en grund för framtida utveckling av systemet genom att sänka förberedelsetiden för testning och systemets förkunskapskrav. Vårt utvärderingsarbete har dessutom visat att direktkommunikation mellan bilarna är en möjlighet i Gulliver, vilket motiverades i detalj i avsnitt 5.4.5.

Vi i projektgruppen tycker att konceptet att använda miniatyrbilar som mellansteg mellan simulering och testning av fullskaliga testsystem är en smart lösning. Desto verklighetstrognare man kan göra testningarna i ett tidigt stadie desto mer tjänar man på det i slutändan, både tidsmässigt och kostnadsmässigt.

Som nämnts i kapitel 1 kommer en allt mer trafikerad värld behöva nya tekniker för att öka säkerheten och minska utsläppen. Det kommer bli en kostsam process att utveckla och testa dessa nya tekniker men med hjälp av småskaliga testsystem kan man lösa detta på ett ekonomiskt sätt. Desto mer vi har arbetat och läst om dessa tekniker desto mer övertygade blir vi att autonoma bilar, framförda på allmänna vägar i stor skala, kommer att bli verklighet.

Referenser

Behrisch, M. et al. (2011). "SUMO - Simulation of Urban MObility: An Overview". I: *SIMUL 2011, The Third International Conference on Advances in System Simulation.* Barcelona, Spain.

Blanchette, J. och Summerfield, M. (2008). *C++ GUI programming with Qt 4.* Upper Saddle River, NJ: Prentice Hall/Trolltech.

Bray, T. et al. (1998). "Extensible markup language (XML)". I: *World Wide Web Consortium Recommendation REC-xml-19980210.*

Burns, L. D. (2013). "Sustainable mobility: A vision of our transport future". I: *Nature* vol. 497.nr 7448, s. 181–182.

Chandra, T. D., Griesemer, R. och Redstone, J. (2007). "Paxos Made Live - An Engineering Perspective". I: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, s. 398–407.

Dahlgren, E. et al. (2012). *Gulliver – en plattform för testning, utveckling och demonstration av transportsystem.* Göteborg: Chalmers tekniska högskola. (kandidatarbete inom Institutionen för Data och Informationsteknik).

De Prisco, R., Lampson, B. och Lynch, N. (1997). "Revisiting the Paxos algorithm". I: *Distributed Algorithms.* Utg. av M. Mavronicolas och P. Tsigas. Vol. 1320. Lecture Notes in Computer Science. Springer Berlin Heidelberg, s. 111–125.

Défago, X., Schiper, A. och Urbán, P. (2004). "Total order broadcast and multicast algorithms: Taxonomy and survey". I: *ACM Computing Surveys (CSUR)* vol. 36.nr 4, s. 372–421.

Ellis, M. A. och Stroustrup, B. (1990). *The Annotated C++ Reference Manual.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Fiala, M. (2004). "Vision guided control of multiple robots". I: *Computer and Robot Vision, 2004. Proceedings. First Canadian Conference on Computer and Robot Vision*, s. 241–246.

Forward, S. (2008). "Driving Violations: Investigating Forms of Irrational Rationality". Diss. Uppsala Universitet.

Gezici, S. et al. (2005). "Localization via ultra-wideband radios: a look at positioning aspects for future sensor networks". I: *Signal Processing Magazine, IEEE* vol. 22.nr 4, s. 70–84.

Hangal, S och Söderberg-Rivkin, A (2014). *GulliView: A Vision Based Localization System for Autonomous Vehicles.* Göteborg: Chalmers tekniska högskola.

Kernighan, B. W. och Ritchie, D. M. (1978). *The C Programming Language.*

Krisher, T. (2013). *Hackers find ways to hijack car computers and take control.* URL: `http://business.financialpost.com/2013/09/03/hackers-find-ways-to-hijack-car-computers-and-take-control/?__lsa=ce2f-5871` (hämtad 2014-05-15).

Lamport, L. (1998). "The part-time parliament". I: *ACM Transaction on Computer on Systems* vol. 16.nr 2, s. 133–169.

Lamport, L. (2001). "Paxos made simple". I: *ACM SIGACT News (Distributed Computing Column)* vol. 32.nr 4, s. 51–58.

LibPaxos (2013). *LibPaxos: open-source paxos.* URL: `http://libpaxos.sourceforge.net` (hämtad 2014-02-07).

Lin, P. (2013). *The Ethics of Autonomous Cars.* URL: `http://www.theatlantic.com/technology/archive/2013/10/the-ethics-of-autonomous-cars/280360/` (hämtad 2014-05-15).

Loeliger, J. (2009). *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development.* 1st. O'Reilly Media, Inc.

MacDonald, A. (2012). *Paxos by example.* URL: `http://angusmacdonald.me/writing/paxos-by-example/` (hämtad 2014-02-07).

Mills, D. (1991). "Internet time synchronization: the network time protocol". I: *Communications, IEEE Transactions on* vol. 39.nr 10, s. 1482–1493.

Naturvårdsverket (2013). *Transportsektorns miljöpåverkan.* URL: `http://www.naturvardsverket.se` (hämtad 2014-02-06).

Olson, E. (2011). "AprilTag: A robust and flexible visual fiducial system". I: *Robotics and Automation (ICRA), 2011 IEEE International Conference on Robotics and Automation (ICRA 2011)*, s. 3400–3407.

Owens, M. och Allen, G. (2006). *The definitive guide to SQLite.* Vol. 1. Springer.

Pahlavan, M., Papatriantalou, M. och Schiller, E. M. (2011). "Gulliver: a test-bed for developing, demonstrating and prototyping vehicular systems." I: *Proceedings of the 9th ACM International Workshop on Mobility Management & Wireless Access, MOBIWAC 2011*, s. 1–8.

Pinto, C. (2012). "How autonomous vehicle policy in california and nevada addresses technological and non-technological liabilities". I: *Intersect: The Stanford Journal of Science, Technology and Society* vol. 5.

Postel, J. (1980). *User Datagram Protocol.* RFC 768. Internet Engineering Task Force.

Postel, J. (1981). *Transmission Control Protocol.* RFC 793. Internet Engineering Task Force.

Primi, M. (2008). *LibPaxos performance analysis.* University of Lugano.

Ronacher, A. (2014). *What does "micro" mean?* URL: `http://flask.pocoo.org/docs/foreword` (hämtad 2014-05-14).

Rossum, G. (1995). *Python Reference Manual.* Tekn. rapport. Amsterdam, The Netherlands.

Schwaber, K. (1997). "Scrum development process". I: *Business Object Design and Implementation.* Springer, s. 117–134.

Stojaspal, J. (2013). *Ann Arbor and the future of V2V/V2, part I.* URL: `http://analysis.telematicsupdate.com/print/35848` (hämtad 2014-05-09).

Trafikverket (2012). *Nollvisionen.* URL: `http://www.trafikverket.se` (hämtad 2014-02-06).

Transportstyrelsen (2012). *Olyckor efter svårhetsgrad.* URL: http://www.transportstyrelsen.se (hämtad 2014-02-06).

Urmson, C. (2012). *The self-driving car logs more miles on new wheels.* URL: http://googleblog.blogspot.se/2012/08/the-self-driving-car-logs-more-miles-on.html (hämtad 2014-02-06).

Vieira, G. M. D. och Buzato, L. E. (2009). "The Performance of Paxos and Fast Paxos". I: *Proc. of the 27th Brazilian Symposium on Computer Networks, Recife, Brazil*, s. 14.

Vienna Convention (1968). *United Nation Conference on Road Traffic, Convention on Road Signs and Signals, Vienna.*

Volvo Cars (2008). *Vision 2020.* URL: http://www.volvocars.com/intl/top/about_volvo/corporate/volvo-sustainability/safety/pages/vision-2020.aspx (hämtad 2014-02-07).

Volvo Cars (2012). *Volvo Personvagnar deltar i framgångsrika försök med fordonståg. SARTRE-projektet går in i slutfasen.* URL: http://www.volvocars.com/se/top/about/news-events/pages/default.aspx?itemid=283 (hämtad 2014-02-07).

Volvo Cars (2013). *Volvo Car Group initiates world unique Swedish pilot project with self-driving cars on public roads.* [Press release]. URL: https://www.media.volvocars.com/global/en-gb/media/pressreleases/136182/volvo-car-group-initiates-world-unique-swedish-pilot-project-with-self-driving-cars-on-public-roads (hämtad 2014-02-06).

A Källkod

A.1 Simulering av webbkamera

```
import sys
from socket import *
from ctypes import *
import struct
import math
import array
from random import randint
serverPort = 8484
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(("", serverPort))
clientSocket = socket(AF_INET, SOCK_DGRAM)
clientPort = 4242
nrOfCars = 8
speeds = array.array('i')
it = array.array('i')

x = 0
while x < nrOfCars :
    speeds.append(randint(20,100))
    it.append(randint(0, speeds[x]))
    x = x + 1
## Used to insert data that can be read in C++
#uint16
def insertu16(i) :
    array.extend(struct.pack('>H', i))
#uint32
def insertu32(i) :
    array.extend(struct.pack('>I', i))
#uint64
def insertu64(i) :
    array.extend(struct.pack('>Q', i))
#int32
def inserts32(i) :
    array.extend(struct.pack('>i', i))

```

```
36 print "PYTHON SERVER: Server started with " + str(nrOfCars) + "
       cars. Listening on port: " + str(serverPort)
37 while 1:
38     data, clientAddress = serverSocket.recvfrom(2048)
39     data = struct.unpack("!II", data)
40     array = bytearray()
41     if data == (1, 1): #positional data request
42
43         insertu32(2)              #type
44         insertu32(2)              #subtype
45         insertu32(45678)          #seq - not used
46         insertu64(123456789000) #time_seconds - not used
47         insertu64(123019)         #time_useconds - not used
48         insertu32(nrOfCars)       #length
49
50         i = 0
51         while i < nrOfCars:
52             rad = 1000 + i * 500
53             angl = 2 * math.pi / speeds[i] * it[i]
54
55             x_pos = math.cos(angl) * rad
56             y_pos = math.sin(angl) * rad
57
58             #every other car should go clockwise
59             if(i%2 == 1) :
60                 it[i] = (it[i] + 1) % speeds[i]
61                 heading = int((angl + math.pi/2) * 1000)
62             else :
63                 it[i] = (it[i] - 1) % speeds[i]
64                 heading = int((angl + 3 * math.pi / 2) * 1000)
65
66             insertu32(i) #vehicle id
67             inserts32(x_pos)
68             inserts32(y_pos)
69             insertu32(heading)
70
71             i = i + 1
72
73         clientSocket.sendto(array, ('127.0.0.1', 4242))
74         print "PYTHON SERVER: Sending positional data."
75     else:
76         print "PYTHON SERVER: Unknown request."
```

A.2 Uppritning av grafer

```
1  def render_sequence_time(db, session_id):
2      s = session.Session.find(db, session_id)
3
4      # Plot all sent messages for every node included in the
           session s
5      for node_id in range(0, s.nodes):
6          transmissions = s.transmissions_by_sender(node_id)
7          sequences = map(lambda (s, _): s.sequence,
               transmissions)
8          times = map(lambda (s, r): 1000 * (r.timestamp - s.
               timestamp), transmissions)
9          pyplot.plot(sequences, times, 'k.')
10
11     pyplot.xlabel(u'Sekvensnummer')
12     pyplot.ylabel(u'Överföringstid (ms)')
13
14     packet_loss = 100 * s.packet_loss()
15     if packet_loss < 0.1:
16         packet_loss = math.ceil(packet_loss) / 10
17
18     packet_loss_25 = 100 * s.packet_loss(deadline=0.025)
19     if packet_loss_25 < 0.1:
20         packet_loss_25 = math.ceil(packet_loss_25) / 10
21
22     pyplot.suptitle(u'Paxos över %s (%d noder)' % (s.
           transport_label, s.nodes),
23                     fontsize=14)
24     pyplot.title(u'%d ms, %d B, %d %s, %.1f %% paketförlust,
           %.1f %% under 25 ms' %
25                  (s.loop_delay, s.payload, s.burst, s.
                       burst_label,
26                   round(packet_loss, 1), round(packet_loss_25,
                        1)), fontsize=12)
27
28     f = StringIO.StringIO()
29     pyplot.savefig(f)
30     value = f.getvalue()
31
32     f.close()
33     pyplot.clf()
34
```

```
35      return value
36
37  def render_contour(db, session_ids):
38      x = range(2, 5)
39      y = range(0, 105, 5)
40      z = numpy.ones((len(y), len(x)))
41
42      transport_label = None
43      payload_label = None
44
45      for session_id in session_ids:
46          s = session.Session.find(db, session_id)
47          transport_label = s.transport_label
48          payload_label = s.payload
49          for i in range(0, len(y)):
50              z[i][s.nodes-2] = 100 * s.packet_loss(deadline=
                    float(i*5)/1000)
51
52      CS = pyplot.contour(x, y, z,levels=[85, 90, 95, 99],
53                          linestyles=['solid', 'dashed', 'dashdot
                              ', 'dotted'],
54                          colors='k')
55      pyplot.clabel(CS, inline=1, fontsize=10)
56      pyplot.suptitle(u'Paxos över %s (%d B)' % (transport_label,
            payload_label), fontsize=14)
57      pyplot.title(u'Sannolikhet att nä en viss överföringstid',
           fontsize=12)
58      pyplot.xlabel(u'Antal noder')
59      pyplot.ylabel(u'Överföringstid (ms)')
60      pyplot.xticks([2,3,4])
61      pyplot.yticks(range(0, 105, 5))
62
63      f = StringIO.StringIO()
64      pyplot.savefig(f)
65      value = f.getvalue()
66
67      f.close()
68      pyplot.clf()
69
70      return value
```

A.3 Sessionshantering för Atomic Broadcast

```
1  class Session:
2      def __init__(self, **kwargs):
3          self.id = kwargs['id']
4          self.nodes = kwargs['nodes']
5          self.loop_delay = kwargs['loop_delay']
6          self.payload = kwargs['payload']
7          self.transport = kwargs['transport']
8          self.running_time = kwargs['running_time']
9          self.burst = kwargs['burst']
10
11         self.db = kwargs['db']
12
13         self.transmissions_cache = []
14
15     @staticmethod
16     def find(db, id):
17         """
18         Get a session from the database and turn it in to a
               Session object.
19         """
20         db_cursor = db.cursor()
21         db_cursor.execute('SELECT * FROM sessions WHERE id = :
               id', { 'id': id })
22         arguments = db_cursor.fetchone()
23         return Session(id=arguments[0], nodes=arguments[1],
24                        loop_delay=arguments[2], payload=
                              arguments[3],
25                        transport=arguments[4], running_time=
                              arguments[5],
26                        burst=arguments[6], db=db)
27
28     @property
29     def burst_label(self):
30         if self.burst == 1:
31             return 'salva'
32         else:
33             return 'salvor'
34
35     @property
36     def transport_label(self):
37         if self.transport == 'eth':
38             return 'Ethernet'
39         else:
```

```
40                  return 'Wi-Fi'
41
42          @property
43          def transmissions(self):
44              """
45              Get all transmissions.
46              """
47
48              if len(self.transmissions_cache) > 0:
49                  return self.transmissions_cache
50
51              query = """
52              SELECT *
53              FROM transmissions
54              WHERE session_id = :session_id
55              ORDER BY sequence,sender,timestamp ASC, burst,receiver
56              """
57              rows = self.db.cursor().execute(query, { 'session_id':
                    self.id })
58              for row in rows:
59                  self.transmissions_cache.append(
60                      Transmission(sender=row[0], receiver=row[1],
                            sequence=row[2],
61                                   burst=row[3], timestamp=row[4],
                                        type=row[5]))
62              return self.transmissions_cache
63
64          def packet_loss(self, **kwargs):
65              """
66              Calculate packet loss.
67              """
68              c = 0
69              total = float(self._count_transmissions(type='SND')) /
                    self.burst
70
71              if 'deadline' in kwargs:
72                  for node_id in range(0, self.nodes):
73                      n = len(filter(lambda (s, r): (r.timestamp - s.
                            timestamp) <= kwargs['deadline'],
74                                      self.transmissions_by_sender(
                                            node_id)))
75                      c = c + n
76                  return c/total
```

```
77            else:
78                for node_id in range(0, self.nodes):
79                    c = c + len(self.transmissions_by_sender(
                          node_id))
80                return 1-c/total
81
82        def transmissions_by_sender(self, sender):
83            """
84            Get transmissions by sender ID.
85
86            The return value is a list of tuples with the sent
87            transmission as the first element and the received
88            transmission as the second element. Where the received
                  element
89            is the transmission with the highest transmission time
                  of all
90            receiving nodes.
91
92            If burst is in use, the lowest burst number available
                  will be
93            selected.
94
95            If a packet is lost it will not be included in the list
                  .
96            """
97            def filter_transmissions_by_sequence_and_sender(t):
98                return t.sequence == sequence and t.sender ==
                      sender and t.type == 'RCV'
99
100           xs = {}
101
102           """
103           Add a tuple (sent_transmission, None) for every
                  sequence
104           number to the dict xs.
105           """
106           for transmission in self.transmissions:
107               if (transmission.sender == sender and transmission.
                      receiver == -1
108                   and transmission.burst <= 1):
109                   if transmission.sequence not in xs:
110                       xs[transmission.sequence] = (transmission,
                              None)
```

```
111
112         for sequence in xs.keys():
113             current = xs[sequence]
114 
115             # All received transmissions sent by sender ID
116             received_transmissions = filter(
117                 filter_transmissions_by_sequence_and_sender,
                    self.transmissions)
118 
119             # A set containing receiver IDs
120             s = set()
121             for transmission in received_transmissions:
122                 if transmission.receiver in s:
123                     continue
124                 else:
125                     s.add(transmission.receiver)
126                     xs[sequence] = (current[0], transmission)
127 
128             # Filter out tranmission if incomplete
129             for i in range(1, self.nodes):
130                 if i != sender and i not in s:
131                     xs[sequence] = (current[0], None)
132 
133         return [ (s, r) for (s, r) in xs.values() if s != None
               and r != None ]
134 
135     def _count_transmissions(self, **kwargs):
136         """
137         Helper function for counting number of transmissions by
                type.
138         """
139         query = """
140         SELECT COUNT(*)
141         FROM transmissions
142         WHERE session_id = :session_id AND type = :type
143         """
144         db_cursor = self.db.cursor()
145         kwargs['session_id'] = self.id
146         db_cursor.execute(query, kwargs)
147         return db_cursor.fetchone()[0]
148 
149 
150 class Transmission:
```

```
151     def __init__(self, **kwargs):
152         self.sender = int(kwargs['sender'])
153         self.receiver = int(kwargs['receiver'])
154         self.sequence = int(kwargs['sequence'])
155         self.burst = int(kwargs['burst'])
156         self.timestamp = kwargs['timestamp']
157         self.type = kwargs['type']
```

A.4 Paxosnod

```
1  struct node {
2    int id;
3    struct event_base *base;
4    struct evproposer *p;
5    struct evacceptor *a;
6    struct evlearner *l;
7  };
8
9
10 /**
11  * Callback function, called when a resolution is done. Prints
       received message
12  * to stdout.
13  */
14 static void deliver(char *value, size_t size, iid_t iid,
      ballot_t b,
15                     int proposer, void *arg)
16 {
17   struct node *n = arg;
18   struct message *m = (struct message *)value;
19
20   struct timeval tv;
21   gettimeofday(&tv, NULL);
22
23   if (n->id != m->node_id) {
24     // LOG_FORMAT is a format string macro
25     printf(LOG_FORMAT,
26            m->node_id,
27            n->id,
28            m->sequence_number,
29            0,
30            tv.tv_sec,
31            tv.tv_usec,
```

```
32              "RCV");
33    }
34  }
35
36  /**
37   * Initialize a node by starting an acceptor and a learner. If
        node_id is 0 a
38   * proposer is also started.
39   */
40  struct node *node_init(int node_id, const char *config_file)
41  {
42    struct node *n = malloc(sizeof(struct node));
43
44    n->id = node_id;
45    n->base = event_base_new();
46
47    if (n->id == 0) {
48      // init_proposer is a wrapper for a libpaxos API function
49      n->p = init_proposer(n, config_file);
50    }
51
52    // init_acceptor is a wrapper for a libpaxos API function
53    n->a = init_acceptor(n, config_file);
54    // init_learner is a wrapper for a libpaxos API function
55    n->l = init_learner(n, config_file);
56
57    return n;
58  }
59
60  /**
61   * Enter the event loop.
62   */
63  void *node_loop(void *arg)
64  {
65    struct node *n = arg;
66    event_base_dispatch(n->base);
67    return NULL;
68  }
69
70  /**
71   * Free resources used by a node.
72   */
73  void node_free(struct node *n)
```

```
74 {
75   event_base_loopexit(n->base, NULL);
76
77   if (n->id == 0) {
78     evproposer_free(n->p);
79   }
80
81   evacceptor_free(n->a);
82   evlearner_free(n->l);
83   event_base_free(n->base);
84
85   free(n);
86 }
```

A.5 Paxosklient

```
1  struct client {
2    int node_id;
3    const char *config_file;
4    struct sockaddr_in proposer_address;
5    struct event_base* base;
6  };
7
8
9  /**
10  * Initialize a client.
11  */
12 struct client *client_init(int node_id, const char *config_file
     )
13 {
14   struct client* c = malloc(sizeof(struct client));
15   c->node_id = node_id;
16   c->config_file = config_file;
17
18   // Load proposer from config file
19   struct evpaxos_config *config = evpaxos_config_read(c->
       config_file);
20   c->proposer_address = evpaxos_proposer_address(config, c->
       node_id);
21   evpaxos_config_free(config);
22
23   return c;
24 }
```

```
25
26 /**
27  * Enter the send loop and connect to a proposer.
28  */
29 void *client_loop(void *arg)
30 {
31   struct client *c = arg;
32
33   struct event_base *base = event_base_new();
34   // Connect to proposer specified in the config file
35   // connect_to_proposer is a wrapper for a libpaxos API
       function
36   struct bufferevent *bev =
37     connect_to_proposer(base, (struct sockaddr *)&c->
         proposer_address);
38
39   c->base = base;
40
41   struct message m;
42   m.node_id = c->node_id;
43
44   struct timeval tv;
45   int delay;
46
47   // NUMBER_OF_MESSAGES is the set number of messages for test
48   for(int i = 0; i < NUMBER_OF_MESSAGES; i++) {
49     m.sequence_number = i;
50     gettimeofday(&tv, NULL);
51     paxos_submit(bev, (char *)&m, paxos_config.payload_size);
52     event_base_dispatch(base);
53     // LOG_FORMAT is a format string macro
54     printf(LOG_FORMAT,
55            m.node_id,
56            -1,
57            m.sequence_number,
58            0,
59            tv.tv_sec,
60            tv.tv_usec,
61            "SND");
62     // DISPATCH_BASE_DELAY is the set loop delay
63     delay = DISPATCH_BASE_DELAY + (rand() % 7 - 3);
64     time_t sdelay = (int) delay / 1000;
65     long longns = (delay % 1000) * 1000000;
```

```
66      struct timespec nsdelay;
67      nsdelay.tv_sec = sdelay;
68      nsdelay.tv_nsec = longns;
69      nanosleep(&nsdelay, NULL);
70    }
71
72    return NULL;
73  }
74
75  /**
76   * Free resources used by a client.
77   */
78  void client_free(struct client *c)
79  {
80    event_base_loopexit(c->base, NULL);
81
82    free(c);
83  }
```