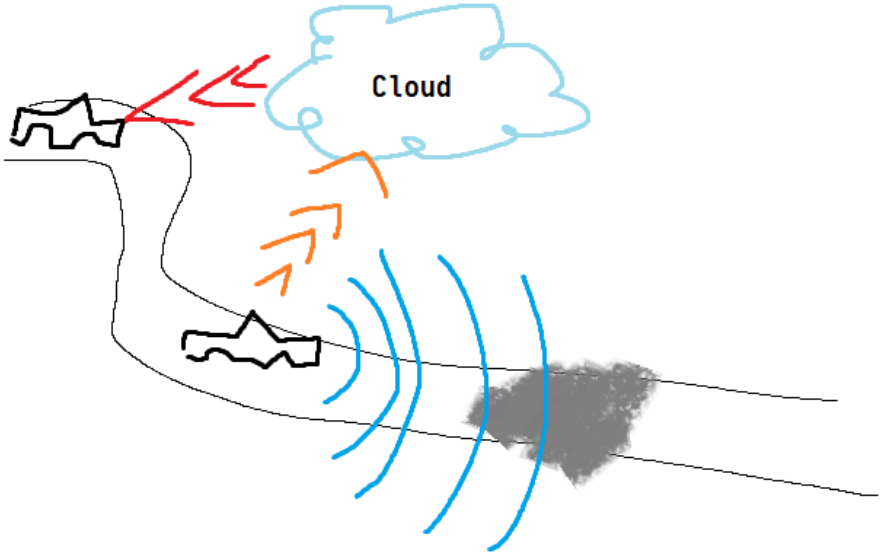
CHALMERS



Konstruktion och protokolljämförelse av V2C varningssystem med MQTT och REST

Examensarbete inom Data- och Informationsteknik

LINUS BERGLUND
TIMMY TRUONG

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2020

Examensarbete

Konstruktion och protokolljämförelse av V2C varningssystem med MQTT och REST

LINUS BERGLUND
TIMMY TRUONG

CHALMERS

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola
Göteborgs Universitet
Göteborg, Sverige 2020

Konstruktion och protokolljämförelse av V2C varningssystem med MQTT och REST
LINUS BERGLUND
TIMMY TRUONG

Handledare: Sakib Sistek, Institutionen för Data- och Informationsteknik
Examinator: Jonas Duregård, Institutionen för Data- och Informationsteknik

Examensarbete
Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet
SE-412 96 Göteborg
Telephone +46 31 772 1000

Göteborg, Sverige 2020

Konstruktion och protokolljämförelse av V2C varningssystem med MQTT och REST
LINUS BERGLUND
TIMMY TRUONG
Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola
Göteborgs Universitet

Sammanfattning

Halt väglag förorsakar många fordonsolyckor. Förare behöver redan ta hänsyn till faktorer som kringliggande fordon, trafikregler, samt balansering av tung last vid svängar. Blir föraren dessutom överraskad av is på vägen kan detta medföra olyckor med ödesdigra utfall. Främst skulle denna typ av olycka kunna undvikas om föraren i förväg meddelades om väglagets status. Examensarbetet undersökte kommunikationsteknologin Vehicle-to-Cloud, specifikt tillämpningen att varna närbelägna förare om hala och riskfulla vägar i deras rutt. I denna undersökning utvecklades även två applikationer baserade på protokollet MQTT respektive REST. Detta arbete utfördes på Vector Sweden (Vector Informatik GmbH) vid Lindholmen, Göteborg. Applikationerna överlämnades i demonstrationssyfte samt som prestandajämförelse av V2C-kommunikationen åt företaget. MQTT-lösningen hade `2,235` gånger lägre responstid än REST-lösningen under benchmarks. I genomsnitt uppvisade MQTT- och REST-lösningen en svarstid på `0,2146` ms, respektive `0,4797` ms. I en säkerhetskritisk miljö är minimering av svarstid nödvändigt, varav MQTT-protokollet uppvisade lägst i detta arbete.

Nyckelord: Protokoll, halt väglag, fordonsolyckor, fordonskommunikation, Vehicle-to-Cloud, MQTT, REST, benchmarks, responstid, säkerhetskritisk miljö.

Abstract

Slippery roads cause many vehicle accidents. Drivers need to carefully consider factors such as surrounding vehicles, traffic regulations, and balancing cargo during swerves. Accidents with fatal outcomes can occur if the driver is unaware of the ice on the road. This type of accident could be avoided if the driver was notified in advance of the state of the road. This degree project surveyed the Vehicle-to-Cloud communication technology, specifically the application of warning nearby drivers about slippery roads en route. In this study, two applications were developed based on the protocol MQTT and REST. The project was performed at Vector Sweden (Vector Informatik GmbH) at Lindholmen, Gothenburg. These applications were designed for demonstration purposes, to be delivered to Vector, as well as a performance comparison of the V2C communication. The MQTT-based solution featured 2.235 times lower reponse time than the REST solution during benchmarks. On average, the MQTT and REST solution exhibited a response time of 0.2146 ms and 0.4797 ms respectively. In a safety-critical environment, minimization of the response time is vital, where the MQTT protocol exhibits a lower average.

Keywords: Protocol, slippery roads, vehicle accidents, vehicle communication, Vehicle-to-Cloud, MQTT, REST, benchmarks, response time, safety-critical environment.

Förord

Detta examensarbete utfördes av Linus Berglund & Timmy Truong i samarbete med *Vector Sweden* och *Chalmers Tekniska Högskola*.

Vector har framförallt tillhandahållit projektförslaget och bistått med enorm fackkunkap inom området, samt en positiv miljö där man blir uppmuntrad att testa nya tekniker. Chalmers har bistått med rådgivning och projektplaneringsupplägg från tidigare erfarenheter både inom akademiska såväl industriella projekt.

Följande individer har varit fundamentalt projektkritiska för arbetets lyckade genomförande.

Från *Vector*

- Åsa Björnemark - Product Manager, Manager Test Solutions
- Samuel Winther - Test Solutions Engineer
- Joakim Robakowski - Test Solutions Engineer

Från *Chalmers*

- Sakib Sistek - Handledare & Forskningsingenjör

Slutligen vill vi tacka vänner och familj, som har stöttat oss under projektets gång.

Linus Berglund, Timmy Truong, Göteborg, Sverige, 2020

Innehållsförteckning

Figurer

Tabeller

Kodlistningar

Ordlista

benchmark Prestandajämförelse. iii, 3, 23

hosting Vara värd för. 17

JSON JavaScript Object Notation är ett öppen filformatsstandard som i detta projekt används som serialiseringsstruktur vid datatransport mellan de olika komponenterna. 14

message broker Motsvarar server i den traditionella client/server-modellen. Får ej förväxlas med MQTT backend, som är en MQTT Client. 12, 17

message queue Message queue är ett alternativt sätt att kommunicera mellan tjänster, till skillnad från exempelvis en klient mot en server. 8

overhead Godtycklig kombination av överflödig beräkningstid, minne, bandbredd, eller andra nödvändiga resurser för att utföra ett ändamål. 24, 42

p99 Inom 99% av totalet (99th percentile), ungefär tre standardavvikelser (99.7%). 33

PUBACK Ett PUBACK (publish acknowledgement) paket är godkännandet av ett PUBLISH paket med QoS 1. 9

publish-subscribe pattern Ett kommunikationssätt, 'syskon' till message queueing. 40

round-trip time Tur- och returtid. x, 24, 27, 31, 32, 35, 36, 38, 39

throughput Datagenomströmningskapacitet; ett mått på mängden data som färdas genom en kommunikationslänk. 1

topic En UTF-8 sträng som MQTT brokern använder för att filtrera meddelandena från varje ansluten klient. *För mer info läs om topic i Bakgrund*. 40

user story Informell beskrivning i naturligt språk på en eller flera mjukvarufunktioner i ett system. 10, 42

V2C Kommunikationsteknologin Vehicle-to-Cloud. 3, 40

1
Inledning

Följande kapitel beskriver arbetets problembeskrivning, dess bakgrund, syfte och mål. Vidare beskriver kapitlet även avgränsningar på vad examensarbetet inte täcker.

1.1 Bakgrund

Vector Informatik GmbH (*Vector*) arbetar med intern kommunikation inuti fordon, därbland verktyg, metoder, och protokoll för att facilitera effektiv kommunikation mellan olika Electronic Control Units (*ECU*). Historiskt sett har bussprotokollen CAN, LIN, FlexRay m.fl. använts för detta ändamål. I framkant ligger nu Automotive Ethernet, en sidospecifikation av Ethernetprotokollet som är anpassad för fordonskommunikation. Automotive Ethernet besitter mycket högre throughput än de traditionellt använda protokollen.

Vi kommer under projektets gång arbeta med verktyg utvecklade av *Vector*, därbland huvudsakligen *Vector CANoe* samt tillhörande hårdvara.

> *CANoe is the comprehensive software tool for development, test and analysis of individual ECUs and entire ECU networks. It supports network designers, development and test engineers throughout the entire development process – from planning to system-level test [2].*

Projektet gäller att skapa en konceptvalidering av Vehicle-to-Cloud (V2C) för att därpå kunna genomföra prestandajämförelser mellan topologiskt skilda arkitekturer. V2C är en teknologi som används när funktioner inte kräver alltför strikta realtidskrav som t.ex. via mobilapp låsa/låsa upp fordon (Lock/Unlock) eller som detta arbete eftertraktar; varningsnotifikationer till förare om halt väglag, som även benämns som *Road Ice Warning*.

1.1.1 Connected car

Connected car är en term som används för att beskriva fordon som är uppkopplade mot ett dubbelriktat nätverk utanför fordonet. Genom att fordonet är uppkopplat, så medför detta att fordonet innehar internetåtkomst och kan utbyta data mellan fordon till andra enheter, både internt och externt. Connected car innefattar även

specifika teknologier som i sin tur beskriver kommunikationen mellan *Vechicle-to-...* (*Fordon-till...*). Bland dessa teknologier finns fem större kommunikationsvägar mellan fordon till enhet [5]:

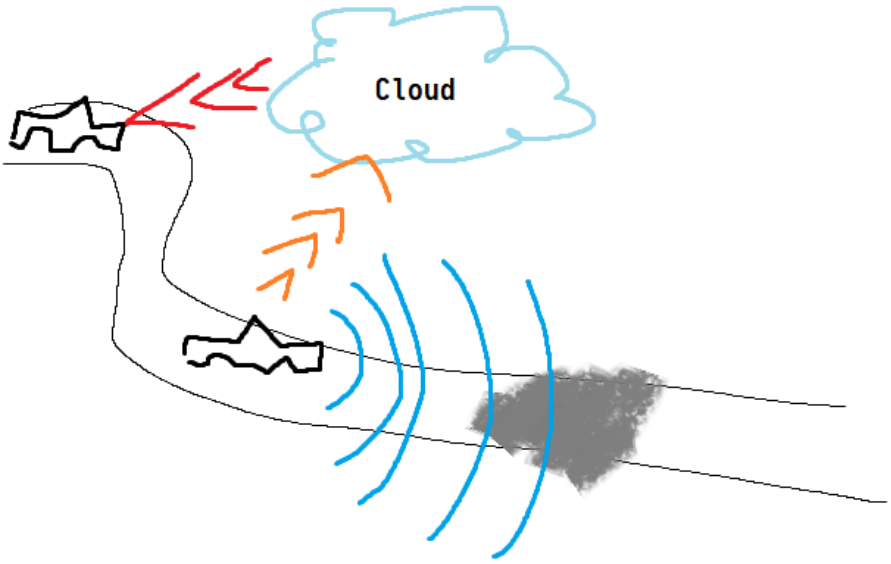
1. Vehicle-to-Infrastructure (V2I)
2. Vehicle-to-Vehicle (V2V)
3. Vehicle-to-Cloud (V2C)
4. Vehicle-to-Pedestrian (V2P)
5. Vehicle-to-Everything (V2X)

Detta arbete tillämpar teknologin **V2C**.

Figur 1.1: V2X: Hur sammankopplingen mellan fordon och enheter kan se ut. (Figur av Hans-J. Brehm, hämtad från Wikimedia Commons [4].)

1.1.2 Vehicle-to-Cloud

Vehicle-to-Cloud (V2C) är en teknologi som innebär en kommunikationslänk mellan fordon och molnet. Detta i syfte att erbjuda fordonen med aggregerad information från en mängd sammankopplade molntjänster [6]. I detta arbete tillämpas ett varningssystem via V2C.



Figur 1.2: En överblick på hur V2C kan tillämpas.

1.2 Mål

Målet är att konstruera två varningstjänster med vardera protokollimplementering, samt länka dessa tjänster med fordon, där fordonen i projektvalideringssyfte är simulerade i en simuleringsmiljö skapat i verktyget *Vector CANoe*.

1.2.1 Delmål

- Konstruera systemen på ett modulärt, tydligt, samt väldokumenterat sätt.
- Utred eventuella latensskillnader mellan nyttjade protokoll och ramverk.
- Utred hur stor påverkan varningsfunktionaliteter har på latens.

1.3 Syfte

Syftet med projektet är att utforska samt benchmarka två olika kommunikationsprotokoll (REST och MQTT) med avseende på V2C-kommunikation via konstruktion av en varningstjänst.

Det är även praxis att bygga lösningar på ett sätt som möjliggör friktionsfri vidareutveckling. Därför ska systemen konstrueras på ett modulärt, tydligt, samt väldokumenterat sätt. Nyckelegenskapen låg latens är även säkerhetskritisk. Projektet måste därför utreda eventuella latensskillnader mellan nyttjade protokoll och ramverk. För att vidare mäta latenspåverkan av funktionalitet behöver samtliga lösningar ha snarlika implementationer.

1.4 Precisering av frågeställningar

Frågeställningar som skall undersökas och besvaras i detta projekt:

- Vad är projektrelevanta för- och nackdelar med MQTT kontra REST?
- Hur behöver lösningsarkitekturerna för respektive lösning anpassas för att ta hänsyn till V2C?
- Vilken genomsnittlig responstid kan lösningarna bibehålla över en stabil kommunikationslänk?

1.4.1 Hypoteser

Hypoteser som hålls innan utfört arbete:

- MQTT-lösningen kommer ha genomsnittligt lägre responstid än REST-lösningen.
- REST-lösningen kommer gå snabbare att implementera, tack vare tidigare professionella erfarenheter inom området.

1.5 Avgränsningar

I projektet kommer vi inte arbeta med konstruktion av hårdvara, t.ex. ingen kretskortsdesign eller liknande; däremot kommer *koppling* mot befintlig hårdvara ingå som moment. ECU:erna kommer vara simulerade, inte reella, så ingen ECU kommer behöva byggas och integreras till komplett stadie. Slutligen kommer bara en ECU per lösning behöva programmeras och testas, inte flera, i.e. lösningen blir mer ad-hoc än generell.

För att förtydliga, följande punkter kommer ej utföras i projektet:

- Konstruktion av ny hårdvara.
- Användning av verkliga fordon/ECU:er.
- Lokalisering och utilisering av reella isfläckar.
- Klientmiljöer som ej är kompatibla med *CANoe*.

2

Teori & Teknisk bakgrund

2.1 Vector CANoe

Vector CANoe är ett mjuvaruverktyg för utveckling, analys och testning av ECU:er. CANoe-konfigurationsfiler har filformatet `.cfg` och sparas som konfigurationer (inställningar) för att användaren ska kunna spara och återuppta sessioner. I nätverkssektionen tillhandahåller verktyget användaren att skapa noder, som i sin tur kan öppnas i diverse textredigerare för att skriva kod. CANoe har en integrerad utvecklingsmiljö som kallas för *CAPL Browser* [3].

2.2 CAPL

Communication Access Programming Language (CAPL) är ett C-liknande programmeringsspråk. Språket används för att programmera nätverksnod-moduler (network node modules) och/eller specifika evalueringsprogram för individuella applikationer. CAPL är ett eventbaserat programmeringsspråk och inte driven av avbrott (interrupts). Användaren har direkt tillgång till ECU:ns interna kommuniktionssignaler, systemvariabler och tillståndsbedömande parametrar. Programmeringsspråket har även stöd för sammanlänkning av användardefinierade Dynamic-link libraries (DLLs). [3].

2.3 REST

Representational state transfer (REST) är ett arkitekturbegrepp som kan appliceras vid arkitektur och konstruktion av webbtjänster (då kallade *RESTful Web Services*). Kortfattat är det en samling regler man förhåller sig till vid utveckling av sådana tjänster. Roy Fielding beskriver sex fundamentala krav för att en applikation skall vara *RESTful* i sin doktorsavhandling om REST [8], som vardera beskrivs i följande delkapitel. REST-lösningen i detta projekt använder sig av HTTP som underliggande protokoll.

2.3.1 Client-server

Innebär separation av lagring av data på servern kontra gränssnitt hos klienten. I webbsammanhang ligger största styrkan då i att respektive komponenter kan utvecklas oberoende av varandra. Innebär även att relationen mellan antalet klienter

och servrar inte är bestämd.

2.3.2 Stateless

Med stateless kommunikation behöver varje förfrågning från klienten innehålla all nödvändig information som servern kräver, då servern inte lagrar klienternas tillstånd (cookies, variabler, temporär data etc.) mellan olika förfrågningar.

2.3.3 Cacheability

Serverns svar till klienterna måste beskriva sig som cachningsbara eller ej, i syfte att undvika onödig data i klientförfrågningar som t.ex. utdaterad information.

2.3.4 Uniform interface

Nyckelegenskapen som skiljer REST från resterande nätverksbaserade arkitekturbegrepp. Genom att generalisera och standardisera integration mellan nätverkskomponenter förenklas hela arkitekturen, och utbyggnad kan ske mycket effektivare.

2.3.5 Layered system

För enkel utbyggnad behöver inte klienter och servrar ha kunskap om de kommunicerar med slutdestinationen eller inte. Exempelvis kan en lastbalanserare enkelt implementeras mellan dessa nätverkslager utan nödvändiga integrationsförändringar hos vardera part.

2.3.6 Code-on-demand

Innebär att servern kan förstärka klienten med ytterligare funktionalitet genom att skicka exekverbar kod. I många fall behövs ingen dynamisk expandering av klientfunktionalitet, och därav är detta krav endast nödvändigt vid behov.

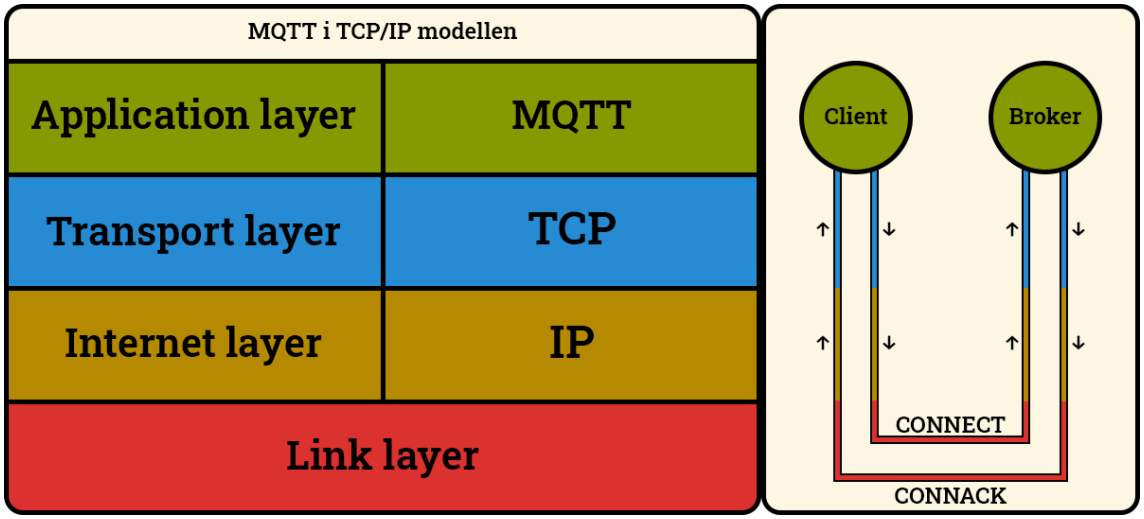
2.4 MQTT

MQ Telemetry Transport (MQTT) är ett nätverksprotokoll för transport av data mellan enheter via *publish/subscribe*-mönster. Protokollet uppfanns av Dr Andy Stanford-Clark och Arlen Nipper, designat för att vara lättviktat och simpelt. Dess primära användningsområde är huvudsakligen Machine-to-Machine (M2M) och IoT (Internet of Things) där begränsade enheter med låg bandbredd, hög latens och/eller instabilt nätverk kan förekomma. [15]

2.4.1 Publish–subscribe pattern

Publish-subcribe mönstret som även benämns *pub/sub* är ett alternativt meddelandemönster till den traditionella ovannämnda client-server. Pub/sub-mönstret frånkopplar klient och backend, utan överlåter själva hanteringen till en broker. Pre-

numeranter har alltså ingen direkt sammankoppling [9][18]. Ett exempel är t.ex. brevbärare, brevlådor och mottagare. Brevbäraren (broker) lägger brev (messages) i brevlådor (topics), men träffar inte mottagaren (client) personligen. En MQTT-anslutning använder publish-subscribe mönstret.



Figur 2.1: MQTTs arkitektur baserad på TCP/IP-stacken.

2.4.2 MQTT client

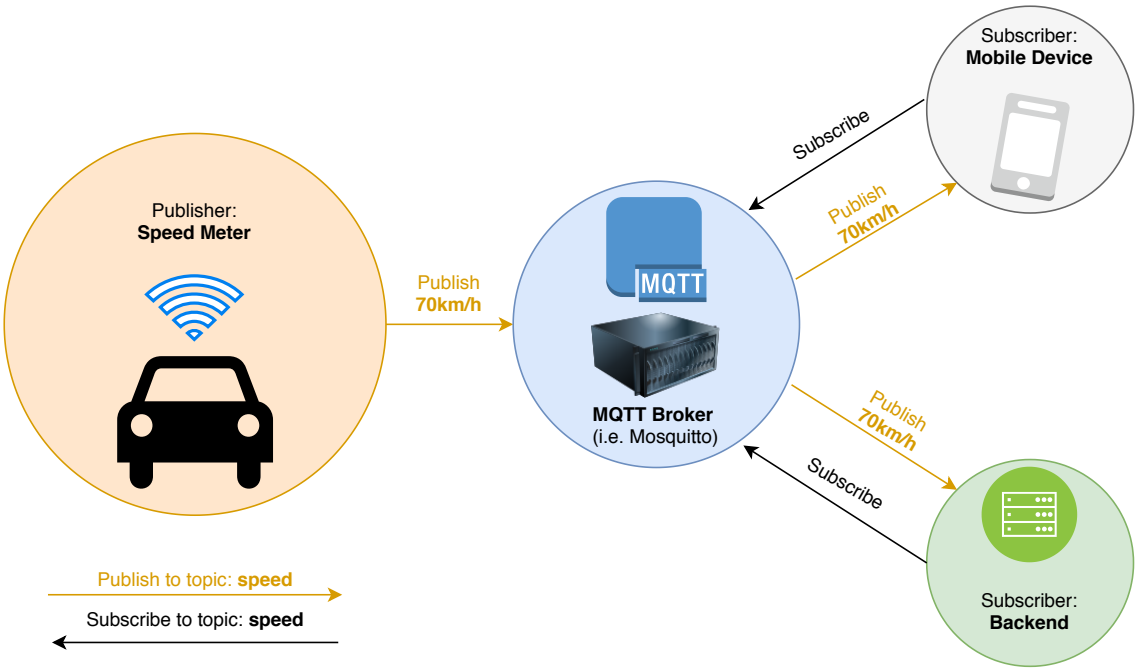
Ett program eller enhet som använder MQTT kallas för *MQTT Client*. En klient kan:

- Öppna nätverksanslutning till Server.
- Publicera (publish) applikationsmeddelanden (application messages) som andra klienter kan lyssna på.
- Prenumerera (subscribe) på förfrågningar av appliktationsmeddelanden som klienten är intresserad av.
- Avprenumerera (unsubscribe) på förfrågningar av applikationsmeddelanden.
- Stänga nätverksanslutning till Server.

2.4.3 MQTT broker

Motparten till en MQTT Client, dvs MQTT Server, kallas även för *MQTT Broker*. En broker kan:

- Acceptera nätverksanslutningar från klienter.
- Acceptera applikationsmeddelanden publicerade av klienter.
- Behandla prenumerationsförfrågningar (subscribe requests) och avprenumerationsförfrågningar (unsubscribe requests) från klienter.
- Vidarebefordra applikationsmeddelanden som matchar klientprenumerationer.
- Stänga nätverksanslutningen från klienten.



Figur 2.2: En generell avbildning på hur meddelandena utbyts i MQTTs kommunikationskedja baserad på publish/subscribe. I bilden är alla subscribers klienter. En MQTT-anslutning är alltid mellan klient och broker (server). Klienter är aldrig direkt anslutna till varandra.

2.4.4 Topic

Ordet *topic* (”ämne” på svenska) hänvisar inom MQTT-området en UTF-8 sträng som brokern använder för att filtrera meddelandena från varje ansluten klient. Alla topics (plural) innehåller en eller flera *topic-nivåer*. Varje nivå separeras genom ett snedstreck `/`, även kallad *topic level separator*. MQTT topics är, till skillnad från message queues, lättviktade. Klienten behöver inte skapa den efterfrågade topic:en innan det går att prenumerera eller publicera på den. Brokern accepterar varje topic utan någon tidigare initiering. Följande exempel på topics är acceptabla.

```
myhome/groundfloor/livingroom/temperature
USA/California/San Francisco/Silicon Valley
5ff4a2ce-e485-40f4-826c-b1a5d81be9b6/status
Germany/Bavaria/car/2382340923453/latitude
```

2.4.5 QoS

MQTT har en utmärkande egenskap som benämns *Quality of Service*. QoS i MQTT har tre nivåer, där nivån indikerar på typen av överenskommelse mellan sändare av ett meddelande respektive mottagare. Mer specifikt styr QoS-nivån hur pålitligt ett typiskt meddelande skall levereras. Nivåerna, ökande i pålitlighetsgrad, är följande:

- `At most once (0)` - Minst pålitlig, ingen garanti på att meddelandet levereras. Benämns vanligen ”fire and forget”.
- `At least once (1)` - Pålitlig, sändaren sparar meddelandet tills den får ett PUBACK paket från mottagaren. Meddelandet kan skickas eller tas emot fler än en gång.
- `Exactly once (2)` - Mest pålitlig, garanterar att meddelandet levereras exakt en gång. Detta uppnås genom en såkallad en ”four-way-handshake”, två förfrågningar och två svar, mellan sändare och mottagare.

3
Metod

Hela projektarbetet kommer att delas upp i små mindre moduler, som sträcker sig över definerad tidsram. Modulerna kommer utvecklas med Agila arbetsmetoder [20], därbland veckovisa *sprints*, där möten med handledare från *Vector* sker varannan vecka för återkoppling.

Huvudsakligen kommer vi bygga en simulerad ECU i *CANoe* samt en serverapplikation vars uppgift är att kommunicera med enheten. CANoe har ett mycket brett användningsområde, samt tämligen komplicerad inlärningskurva. Därav för att kunna erfara en grundläggande förståelse om CANoe och dess användning, därför krävs det att närvara en kurs som detaljerat går igenom teorin och praktiska exempel som programmet kan tillämpa.

Efter upprättning av funktionsduglig server samt ECU ska härnäst en klient utvecklas i demonstrationssyfte åt *Vector*. Klienten kommer skrivas i C# med *Windows Forms* som gränssnittsramverk, och är tanken att agera som en visuell *Man-in-the-Middle* för kommunikation mellan ECU och server.

För att testa lösningen övervägs flera lösningsmetoder, *exhaustive testing* [12], *integrationstestning* (end-to-end), *enhetstestning*, samt *Monte Carlo-testning* [17]. Exhaustive testing anses noggrann och precis, men tar orimligt lång tid att genomföra. Integrationstestning kräver en komplett produkt, vilket projektet inte täcker, då det efterfrågas ett koncept av lösningen. Enhetstestning lär nog inte vara lämplig då projektet kan ofta skifta avsevärt, vilket leder till att testerna blir förlegade och måste skrivas om.

Monte Carlo-metoden bygger främst på slumpmässig testning, vilket lämpar sig för detta projekt som genomsyras av nätverkskommunikation, som typiskt beror på externa variabler. Vidare kommer standardavvikelser användas för insamlad data vid presentation av grafer.

Rent administrativt ska flera plattformar för versions- samt user story hantering jämföras, bl.a. GitLab, GitHub, Trello.

4
Genomförande

4.1 Kravspecifikation

Valet av programmeringverktyg såsom versionshantering, språk och design av GUI-applikation var i stor omfattning fritt. Däremot önskade Vector att lösningarna implementerades i företagets självutvecklade verktyg, CANoe. Genom att använda CANoe var det möjligt att simulera ECU:er och skulle därför användas i projektet.

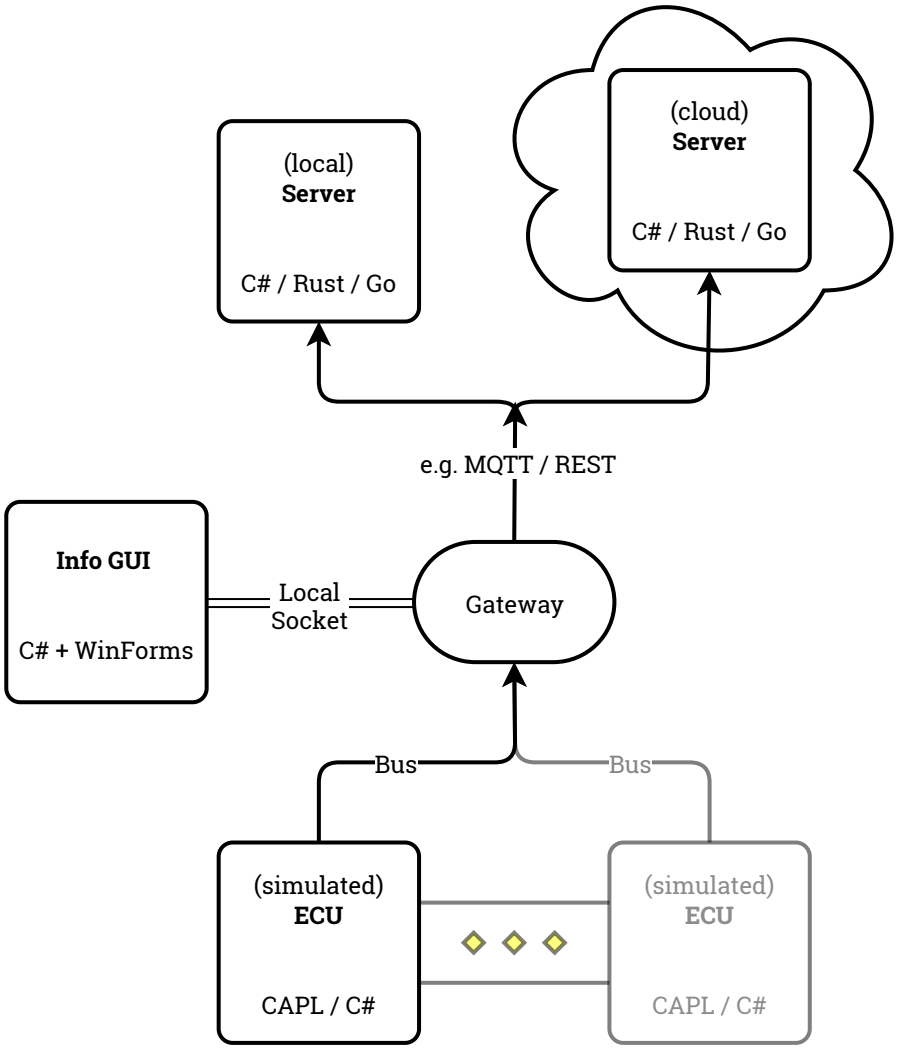
Trots att CANoe främst stöder CAPL, ett egenutvecklat C-likt programmeringsspråk, rekommenderades det av företaget att skriva lösningarna i C#. Detta för att möjliggöra användning av allmänt tillgängliga funktionsbibliotek för MQTT och REST.

Utöver att funktionaliteten för lösningarna kunde utvecklas i CANoe, så har verktyget även stöd för testning och design av användargränssnitt. Eftersom CANoe är för närvarande endast släppt för Windows med *.NET Framework* (inte *.NET Core*), ödslades ingen tid på utveckling av lösningar för andra plattformar.

Lösningarna är ämnade som demonstrationsprojekt att visa upp på t.ex. mässor eller liknande.

4.2 Systemöversikt

Vectors projektidé var inte exakt specifierad, utan beskrev önskvärd funktionalitet varpå frihet överlämnades att efter egen tolkning skissa arkitekturen utifrån beskrivna mål.
Lösningarna som utvecklades hade följande mjukvarukomponenter (Se Figur 4.1). Figuren visar en skiss på systemarkitekturen som designades för att spegla Vectors beskrivning om hur komponenterna är kopplade, samt hur de kommunicerar med varandra. Första skissen var godtagbar, men förhöll sig till funktionalitet som sedan visade sig mindre viktig och därpå avprioriterades. Exempelvis visade sig **Info GUI** vara överflödig, och togs därför bort.
Notera att grafen visar att flera simulerade ECU:er kan skapas och drivas parallellt, i syfte att skala lösningen horisontellt (med andra ord, flera bilar).



Figur 4.1: Översikt över den *inledningsvis skissade* systemarkiktekturen

4.3 Initiell protokollutvärdering

I projektet gavs MQTT och REST som huvudsakliga fokuspunkter, utöver dessa utforskades flera alternativ. Inte enbart i syfte att få en bredare förståelse och uppfattning om området, utan också för att kunna påvisa att MQTT och REST är de två bäst lämpade kandidaterna för arbeten likt detta.

4.3.1 Utvärdering av RabbitMQ

RabbitMQ är huvudsakligen en populär message broker, där organisationen implementerar även flertalet klientbibliotek i olika protokoll och programmeringsspråk. För detta projekt utvärderades RabbitMQs klientbibliotek för C#.

4.3.2 Utvärdering av AMQP

Advanced Message Queuing Protocol (AMQP) är en alternativ standard för meddelandehantering. Det upptäcktes att RabbitMQ C# klientbibliotek enbart stödjer AMQP, och inte MQTT. På grund av detta valdes RabbitMQ bort som alternativ för MQTT klienten, då MQTT och REST var protokollen som efterfrågades av Vector.

4.4 Val av ramverk och bibliotek

4.4.1 HTTP

REST kan implementeras på olika sätt beroende på valt underliggande protokoll. För REST-lösningen valdes HTTP.

För arkitekturbegreppet REST valdes två separata redan befintliga bibliotek för C# klienten respektive Go servern. Projektet avser REST implementerat med HTTP, och biblioteken reflekterar detta. Då tidigare erfarenheter applicerades här gick besluten samt konstruktionen mycket fortare.

4.4.1.1 Klient

Klienterna implementerades som simulerade ECU:er i Vector CANoe. Dessa skrevs i C# med .NET Framework 4.7.2, och använde sig därför av `HttpClient` klassen från Microsofts `System.Net.Http` bibliotek. Detta bibliotek valdes då det finns inbyggt i .NET samt verkade stödja alla funktioner som behövs för att konstruera en önskvärd klient.

4.4.1.2 Server

För Go servern fanns även där ett HTTP bibliotek inbyggt, nämligen `net/http`. Då önskvärd funktionalitet finns även där genomfördes ingen granskning av ytterligare bibliotek.

4.4.2 M2MQTT

M2MQTT är ett öppet bibliotek som tillämpades vid utveckling av MQTT-lösningen. Biblioteket innehaver en MQTT Client som tillåter anslutning till godtycklig MQTT Broker. M2MQTT är skriven i programmeringsspråket C# [11].

Vid inledande av processen för förstudie om MQTT testades onlinebrokern HiveMQs *MQTT Browser Client* [14]. Nyttjandet och igångsättning av en MQTT-anslutning i denna webbaserade klientsida ansågs vara relativ lätt. Vidare upptäcktes att HiveMQ även publicerat vägledning på integrationen mellan flertalet programmeringsspråk och MQTT-protokollet, där deras guide för C# utnyttjar M2MQTT [13]. Fördelarna med att HiveMQ, som tidigare nämnt, skrivit en detaljerad likväl lätt-

begriplig guide som tidigt möjliggjorde en potentiell lösning för en av byggstenarna till hela projektet.

4.5 Hårdvarulösning

Initiellt utfördes flertalet övningar utformade av Vector med Vectors hårdvara, bland annat interfacet VN5610A [21]. I själva verket medverkade hårdvarulösningen inte till slutresulatet, utan var mer av en förstudie. Samtidigt ansågs dessa övningar nyttiga och bidrog till att ge förkunskapen om hur kopplingen mellan två enheter som tillämpar CAN-protokollet fungerar.
Vidare uppvisades riktiga ECU:er för fordon som projektet eventuellt skulle kunna använda i framtiden. Dessa användes dock inte under lösningens utveckling.

4.6 Klientrepresentativ protokollklass

Samtliga lösningar behöver ett protokoll för att kunna skicka data till sina respektive servrar. Helst är detta protokoll gemensamt för att underlätta både vid konstruktion av serverlogik samt hantering av klienter. Med detta i avseende skapades den gemensamma protokollklassen som representerar ett fordon. Klassen har lämpliga variabler definierade utifrån önskvärd funktionalitet. Instanser av denna klass serialiserades till JSON innan transmission till respektive komponent, varpå den avserialiserades till objektform igen.

Notera att denna protokollklass är skilt från resterande protokolldiskussioner i projektet, där protokolldiskussionerna främst gäller MQTT kontra REST.

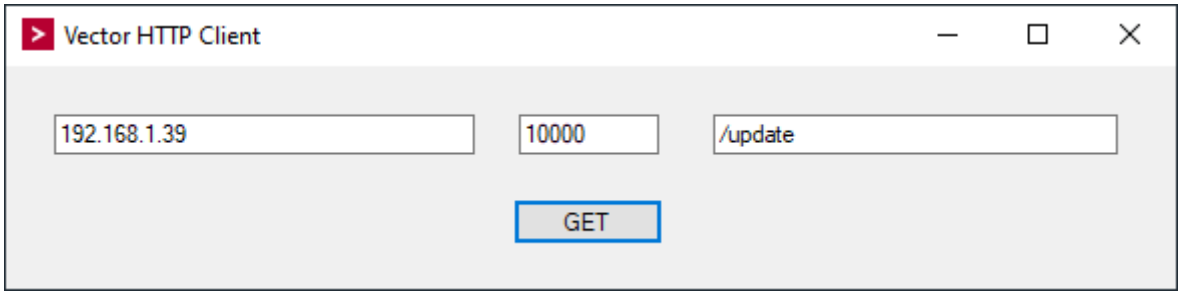
```
using System.Runtime.Serialization;

[DataContract]
internal class Car
{
    [DataMember]
    internal uint id;

    [DataMember]
    internal bool ice;

    [DataMember]
    internal GPSInterface gps;
}

[DataContract]
internal class GPSInterface
{
    [DataMember]
    internal double latitude;

    [DataMember]
    internal double longitude;
}
```

Kodlistning 1: `Car.cs` - Lösningsgemensamt protokoll för hantering av klienter

4.7 REST

Kapitlet beskriver planeringen samt konstruktionen av REST-lösningen.

4.7.1 Vector GUI client



Figur 4.2: Vector HTTP Client: InfoGUI från 4.1

Detta är den första representationen av REST-lösningens HTTP anslutningar. Den exponerar kontroll till ett underliggande HTTP objekt och kan skicka requests med knappen. Meningen med GUI'n var att i demosyfte visa vad som händer vid varje request. Senare versioner döljer detta och har enbart CANoe paneler.

4.7.2 Utvecklingsskede

Konstruktionen av REST-lösningen skedde i delmoment som varierade i storlek och utvecklingstid. Lösningen nyttjade till stor grad tidigare kompetenser inom området för att effektivt färdigställas.

4.8 MQTT

Detta och påföljande delkapitel beskriver genomförandet och processen för MQTT-lösningen till att bli slutresultatet baserat på protokollet MQTT. Lösningen genomgick ett par utvecklingsfaser innan den kom till att bli slutresultatet som presenteras i nästa kapitel. Dessa faser hade syftet med att antingen släppa ny funktionalitet eller förbättra tidigare versioner.

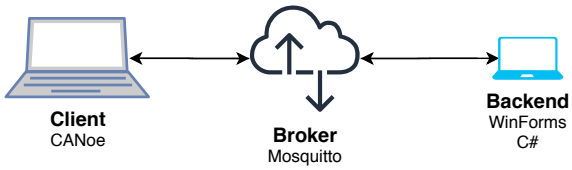
4.9 MQTT utvecklingsfas 1

Första fasen inkluderade grundläggande funktionsanrop, samt uppstart av kommunikation mellan enheter. Målet i denna fas var att upprätthålla simpel dataöverföring, och endast lägga tid på design för användargränssnitt i mån av tid.

4.9.1 Tidigaste kommunikationskedjan

Ursprungligen för att utveckla och testa tidigare nämnda funktionaliteten **Road Ice Warning**, las stora delar av fokuset på att upprätthålla en kommunikationslänk.

Där kommunikationslänken är mer realistisk, nämligen klient och backend separerade från varandra. Därav var den tidigaste kommunikationskedjan mellan klient, backend och broker hårdvarumässigt mellan två datorer (Se Figur 4.3). Meddelandehantering mellan topics och MQTT-klienter sköttes via Mosquitto[7], en open source message broker som möjliggör hosting av topics. Den uppdaterade versionen blev därefter reducerad till en dator, avsedd för testning som finns i kapitlet Resultat.



Figur 4.3: Första kommunikationskedjan där laptop #1 är klient, laptop #2 är backend, slutligen lokal broker Mosquitto installerad på båda datorerna

4.9.2 Klient 0.1 - temperaturförändringar

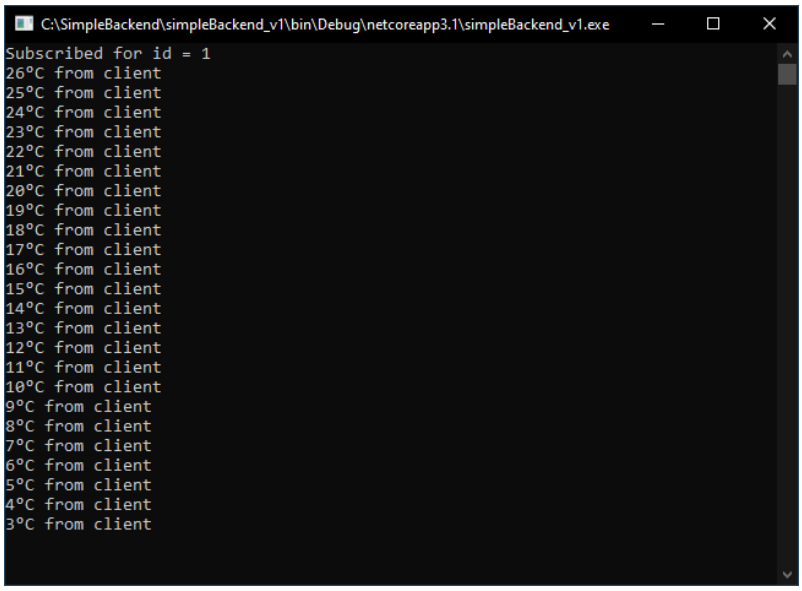
Det första klientdemot hade ingen GUI. Det var skapat för att testa en simpel MQTT-anslutning, samt för att se till att utskick av meddelanden fungerade utan defekter. I demot importerades biblioteket M2MQTT, som har inbyggda funktioner för att publicera i samma topic som korresponderande backend prenumererar på. Klienten får genom CANoes systemvariabel och signalgenerator ett invervall av temperaturvärden som uppdateras var 100:e millisekund. I koden (Se Kodlistning 2) skickar klienten ett nytt temperaturmeddelande till topic `ECU1/temp` varje gång systemvariabeln `ECU1.temp` uppdateras.

```
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

public class simpleClient : MeasurementScript
{
    MqttClient client;
    byte code;

    public override void Initialize()
    {
        //IP address of Laptop #2 - VSGD830NBH
        client = new MqttClient("[IP address]");

        //Connect to broker residing on Laptop #2 with a new unique client ID
        client.Connect(Guid.NewGuid().ToString());
    }

    ...

    [OnChange(typeof(ECU1.temp))]
    public void publishTemp() // will be called whenever "ECU1.temp" changes
    {
        String temp = ECU1.temp.Value.ToString();
        client.Publish("ECU1/temp", // topic
           Encoding.UTF8.GetBytes(temp), // message body
           MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE, // QoS level
           false); // retained
    }
}
```

Kodlistning 2: `simpleClient.cs` publicerar på `ECU1/temp` .

4.9.3 Backend 0.1 - utskrift utan beräkningar

Likaså hade första demot av backend inte någon GUI, utan det blev utskrifter direkt via terminalen. Utskrifterna visade förändringarna i temperatur från klientsidan, och gav en ny utskrift för varje uppdatering.



Figur 4.4: Skärmdump på värden som skickades till simpleBackend via topic ECU1/ från simpleClient.

Observera att de mottagna meddelandena var alla i sträng-format, och endast anpassade för utskrift. Med andra ord var dessa värden ännu ej lämpade för logikberäkning av variablerna på den klientrepresentativa protokollklassen. Med hjälp av följande kodstycke kunde ovannämnda temperaturförändringar från klientsidan erhållas (Se Kodlistning 3).

```
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

namespace simpleBackend
{
    static void Main()
    {
        client.MqttMsgSubscribed += client_MqttMsgSubscribed;
        client.MqttMsgPublishReceived += client_MqttMsgPublishReceived;

        // Prenumerationen på en topic
        client.Subscribe(new string[] { "ECU1/temp" },
        new byte[] { MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE });
    }

    void client_MqttMsgSubscribed(object sender, MqttMsgSubscribedEventArgs e)
    {
        Console.WriteLine("Subscribed for id = " + e.MessageId);
    }

    void client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e)
    {
        String clientMessage = Encoding.UTF8.GetString(e.Message);
        Console.WriteLine(clientMessage + "° degrees from client");
    }
}
```

Kodlistning 3: `simpleBackend` prenumererar på `ECU1/temp` och printar till terminalen för varje ny temperaturförändring.

Temperaturförändringar var dock endast en början på Road Ice Warning-lösningen, därav var det nödvändigt att fortsätta vidareutveckla simpleBackend för att få ett resultat som var mätbart för prestandajämförelse.

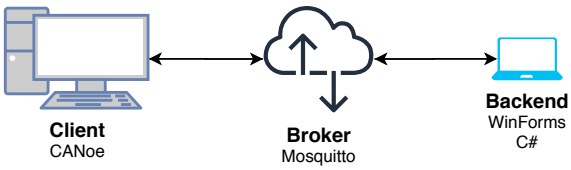
4.10 MQTT utvecklingsfas 2

Simpel kommunikation mellan klient och backend var nu möjlig, nästa steg blev att lägga till ett fönster som kunde inlärningsvänligt visa när det var tillräcklig låg temperatur för att indikera halt väglag. Under utvecklingen uppfattades prestandasvårigheter med en av bärbara datorerna försedda av företaget, som Vector senare

smidigt assisterade med att byta ut.

4.10.1 Uppdaterad kommunikationskedja

Tidigare nämnt var processen av lösningarnas utveckling lite av en flaskhals, då en av datorerna var näst intill oförmögen att starta, kompilera kod och köra CANoe eller Visual Studio 2019. Därpå försåg företaget med en ny och högpresterande stationär dator, som kom till att få hela utvecklingsprocessen att flyta på bättre.

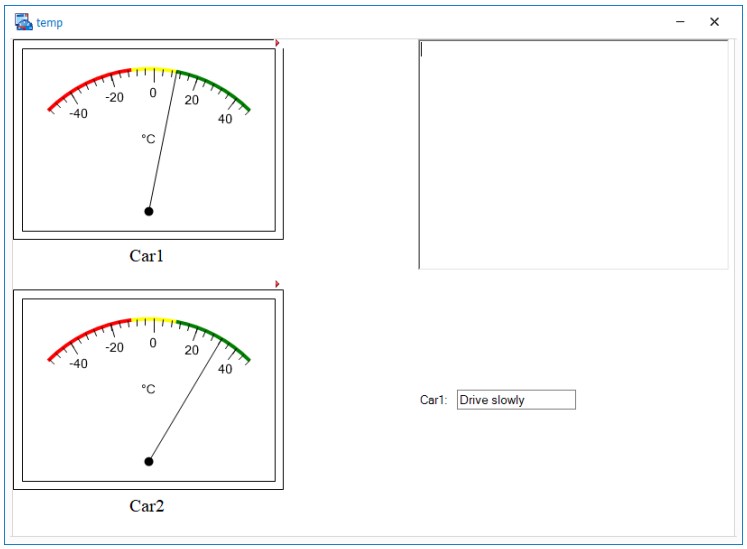


Figur 4.5: Stationär dator som ersatte laptop #1 med bättre prestanda. Samma laptop #2 förblev backend.

Det planerades att låta den stationära datorn med högre prestanda att vara backend och göra de tyngre beräkningarna. När en MQTT-klient försökte instansieras mot den stationära datorns IP-address uppstod felmeddelandet *'Connection is broken'*. Extra tid spenderades inte på vidare felsökning, och samma laptop förblev backend.

4.10.2 Klient 0.2 - varningsindikationer

Senare demot som blev en vidareuteckling på funktionaliteten, fick även ett användargränssnitt för att påbörja en form av varningsnotifikation för användaren. Värdena på komponenterna indikerade temperatur för två simulerade bilar och även en osynlig simulerad isfläck. Där värdena på det röda fältet blev indikationen på att bilarna har närmat sig riskabel temperatur.



Figur 4.6: Skärmdump på första användargränssnittet skapad på CANoe Panels.

4.10.3 Backend 0.2 - avgöra temperaturskillnader

Backend fick nu med hjälp av temperaturvärdena avgöra om en utetemperatur var grupperad som riskfylld eller ej. På ovanbeskrivna skärmdumpen med de respektive färgerna för intervallen symboliserades följande klassificering:

- **temperatur** $\leq$ -10°C : Röd, varningsmeddelandet 'Stanna fordonet'
- **temperatur** $\leq$ 10°C : Gul, varningsmeddelandet 'Kör långsamt'
- **temperatur** $>$ 10°C : Grön, meddelandet 'Allt väl'

Värdena valdes arbiträrt för demosyfte.

4.11 MQTT utvecklingsfas 3

Genom att definiera de föregående utvecklingsstadier var det nu möjligt att sätta ihop dessa idéer till ett mer fullständigt demo.

4.11.1 Klient 0.3 - varningsnotifikation på endimensionell sträcka

Tredje demot fick simulera ett mer avancerat problem än de tidigare demona. Följande scenario designades:

1. Två simulerade bilar med olika starttider på endimensionella sträckor.
2. Osynlig is utlagd på distansenhet 50.
3. Bil 1 åker tidigare än bil 2, och hittar is på distansenhet 50. Skickar därefter info till backend.
4. Backend kalkylerar och sätter en säkerhetsmarginal på 20 distansenheter framför isen som upptäckts.
5. Bil 2 passerar distansenheten 30, och får information från backend att själv vidta försiktighetsåtgärder.

Figur 4.7: Tredje demot.

På detta demo bekräftades att en simpel användning av V2C-kommunikation via CANoe och C# backend med WinForms var möjlig att åstadkomma. Vidare uppmanade företaget att vidareutveckla de endimensionella sträckorna till tvådimensionella, som ansågs vara ett mer verklighetstroget scenario. Därav byttes den singulära koordinaten på nästa demo (slutresultat) till variabler som kunde lagra positioner med två värden.

4.12 Testning

När lösningarna fullgjorts återstod testning och benchmark av protokollen. Båda applikationerna testades genom att låta CANoe-klienter och server placeras lokalt på en och samma enhet. Det finns stöd för många typer av testning i CANoe, men just dessa testfall är konstruerade som *CANoe Nodes*, och skrivna i C#.

Testningsproceduren innefattade följande steg:

1. Planera, diskutera, och fastställa möjliga testfall.
2. Avgränsa testfallen i avseende på relevans.
3. Avgränsa testfallen i avseende på tidsbrist.
4. Implementera testfall i lösningarna, inklusive refaktoreringar.
5. Utför testfallen m.h.a. CANoe och respektive server.
6. Logga resultaten till lämpligt format.
7. Analysera resultaten och plotta dem till grafer.

4.12.1 Testningshårdvara

På grund av Monte Carlo-testingsmetoden finns inga garantier för reproducerbarhet, men i öppenhetens intresse följer här specifikationer för testningshårdvaran.

```
Computer:    MSI MS-7751
CPU:         Intel Core i7-3770K (Ivy Bridge-DT, E1)
             3500 MHz (35.00x100.0) @ 1600 MHz (16.00x100.0)
Motherboard: MSI Z77A-GD65 (MS-7751)
BIOS:        V10.7, 10/22/2012
Chipset:     Intel Z77 (Panther Point DH)
Memory:      16384 MBytes @ 800 MHz, 9-9-9-24
             - 4096 MB PC12800 DDR3 SDRAM - Corsair CML16GX3M4A1600C9
             - 4096 MB PC12800 DDR3 SDRAM - Corsair CML16GX3M4A1600C9
             - 4096 MB PC12800 DDR3 SDRAM - Corsair CML16GX3M4A1600C9
             - 4096 MB PC12800 DDR3 SDRAM - Corsair CML16GX3M4A1600C9
Graphics:    MSI N660 Twin Frozr III OC (MS-V287)
             NVIDIA GeForce GTX 660, 2048 MB GDDR5 SDRAM
Drive:       Samsung SSD 840 Series, 244.2 GB, Serial ATA 6Gb/s @ 6Gb/s
Drive:       WDC WD20EARX-00ZUDB0, 1953.5 GB, Serial ATA 6Gb/s @ 6Gb/s
Drive:       TSSTcorp CDDVDW SH-224BB, DVD+R DL
Sound:       Intel Panther Point PCH - High Definition Audio Controller [C1]
Sound:       NVIDIA GK106 - High Definition Audio Controller
Network:     Intel 82579V (Lewisville) Gigabit Network Connection
Network:     Linksys Wireless-G USB Network Adapter with Wi-Fi Finder ver 1.1
OS:          Microsoft Windows 7 Professional (x64) Build 7601
```

Kodlistning 4: Vectors stationära dator - Första testriggen.

Den andra testningsdatorn har följande specifikation.

```
CPU:         AMD Ryzen Threadripper 2950X (ThreadRipper2/Colfax, PiR-B2)
             3500 MHz (35.00x100.0) @ 2195 MHz (22.00x99.8)
Motherboard: ASRock X399 Taichi
BIOS:        P3.50, 12/24/2018
Chipset:     AMD X399 (Promontory)
Memory:      65536 MBytes @ 1197 MHz, 17-17-17-39
             - 16384 MB PC19200 DDR4 SDRAM - Samsung M391A2K43BB1-CRC
             - 16384 MB PC19200 DDR4 SDRAM - Samsung M391A2K43BB1-CRC
             - 16384 MB PC19200 DDR4 SDRAM - Samsung M391A2K43BB1-CRC
             - 16384 MB PC19200 DDR4 SDRAM - Samsung M391A2K43BB1-CRC
Graphics:    MSI GTX 970 Gaming (MS-V316)
             NVIDIA GeForce GTX 970, 4096 MB GDDR5 SDRAM
Drive:       WDC WD80EZAZ-11TDBA0, 7814.0 GB, Serial ATA 6Gb/s @ 6Gb/s
Drive:       WDC WD80EZAZ-11TDBA0, 7814.0 GB, Serial ATA 6Gb/s @ 6Gb/s
Drive:       WDC WD80EZAZ-11TDBA0, 7814.0 GB, Serial ATA 6Gb/s @ 6Gb/s
Drive:       WDC WD80EZAZ-11TDBA0, 7814.0 GB, Serial ATA 6Gb/s @ 6Gb/s
Drive:       WDC WD80EZAZ-11TDBA0, 7814.0 GB, Serial ATA 6Gb/s @ 6Gb/s
Drive:       WDC WD80EZAZ-11TDBA0, 7814.0 GB, Serial ATA 6Gb/s @ 6Gb/s
Drive:       Samsung SSD 970 EVO Plus 1TB, 976.8 GB, NVMe
Sound:       NVIDIA GM204 - High Definition Audio Controller
Network:     Intel I211AT Copper (Pearsonville) Network Adapter
Network:     Intel Dual Band Wireless-AC 3168
Network:     Intel I211AT Copper (Pearsonville) Network Adapter
OS:          Microsoft Windows 10 Professional (x64) Build 18363.693
             - (1909/November 2019 Update)
```

Kodlistning 5: Linus' serverdator - Andra testriggen.

4.12.2 Testfall

Nedan är testfallen som utfördes innan projektet avslutades. Notera specifikt att testfall som innefattade round-trip time vid nyttolast var planerade, men valdes bort då den faktiska implementeringen av problemlogiken skiljer sig markant mellan de två lösningarna. Någon prestandamässig slutsats hade inte varit möjlig utifrån den hypotetiskt resulterande mätdatan.

Alla tidsbaserade tester uppmättes med `System.Diagnostics.Stopwatch`, en .NET Framework klass, och mäts i `Stopwatch.ElapsedTicks` snarare än sekunder/millisekunder för att inte tappa precision till avrundning. En liknande funktion i CAPL övervägdes, men den uppmätta overheaden ansågs inte relevant för dessa lösningar, som är tänkt att evaluera protokollen i sig. Rimligtvis bör CAPL mätningar lämpligen användas i integrationstest för konstruktion av en produktionsredo prototyp. Testfallen använder sig även av ett Vector bibliotek för att åberopa en funktion med ett fast intervall: `[OnTimer(1000)]`.

Diagrammen nedan är skapade med filtrerad data. Främst används p99 latens, i.e. 99% av latenser är lägre än dessa, då det vanligt att ta bort en viss del av utstickande data.

4.12.2.1 Round-trip time, en klient

I första testfallet startas en server och en klient. Klienten uppdaterar därpå servern med ett intervall på `1000 ms`. Detta fortskrider tills CANoe mätningen avslutas, och koden kan ses nedan.

```
1 Stopwatch rtt = new Stopwatch();
2
3 ...
4
5 rtt.Start();
6 response = await client.PostAsync(this.uri, sc);
7 rtt.Stop();
```

Kodlistning 6: Async/Await i `RestClient.cs` för sändning till server.

4.12.2.2 Round-trip time, flera klienter

I andra testfallet startas en server men flera klienter. Klienterna startas i grupper av 2^n, och fortsätter samma testmetod som en klient med lite hjälpfunktioner.

```
private List<RestClient> GenClients(uint n) {
    List<RestClient> newClients = new List<RestClient>();

    string groupDir = logDir + n + @"\";
    Directory.CreateDirectory(groupDir);

    for (uint i = 0; i < n; i++) {
        newClients.Add(new RestClient(i, groupDir + i + ".csv"));
    }

    return newClients;
}

private void StartClients(List<RestClient> clients) {
    foreach (var c in clients) {
        c.Start();
    }
}

private void StopClients(List<RestClient> clients) {
    foreach (var c in clients) {
        c.Stop();
    }
}
```

Kodlistning 7: Hjälpfunktioner för multipla klienter i REST.

Dessa två testfall kan benämnas som följande:
* Test 1: Simple Latency - Round-trip time from client to server to client
* Test 2: Load Latency - Round-trip time with an array of clients to one server

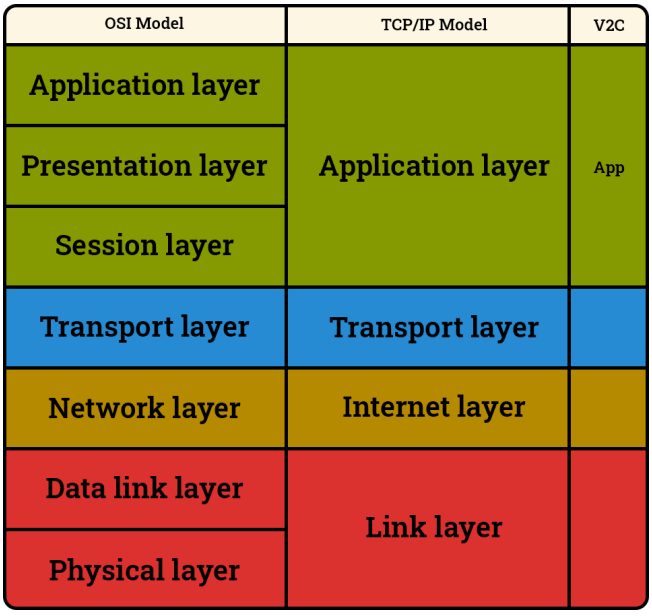
5
Resultat

Detta kapitel presenterar resultaten som uppnåtts, samt planerade resultat som avgränsades i mån av omprioriteringar. Därtill de viktigaste ramverk, bibliotek, och program som användes vid produktion av resultaten. Projektet involverade att utveckla två olika lösningar som använder samma fordonskommunikationsteknologi, V2C. Dessa lösningar beskrivs individuellt.

Målet var att konstruera två varningstjänster med varsin protokollimplementering, samt länka dessa tjänster med fordon, där fordonen i projektvalideringssyfte är simulerade i CANoe. Med de två lösningarna vi har utvecklat idag, så har målet uppfyllts, samt delmålen. Utöver att konstruera dessa varningstjänster fick utvecklades även tester för lösningarna. Dessa tester säger hur stor påverkan varningslösningarna har på latens när man kollar round-trip time.

5.1 Mjukvarustack

Lösningarna i form av applikationer för MQTT respektive REST hamnade i översta lagret, applikationslagret 5.1. För både MQTT-lösningens bibliotek och REST-lösningens ramverk, så hamnar även de i applikationslagret.



Figur 5.1: Applikationen i mjukvarustacken kopplat till OSI- och TCP/IP-modellen

5.2 REST-lösningen

5.2.1 Klient

Lösningens utveckling resulterade framförallt i en klientklass `RestClient.cs`, som kan instansieras i godtyckligt CANoe projekt och agera som en simulerad bil för projektändamålsenliga syften. Varje `RestClient` instans har även ett `Car.cs` objekt, den projektgemensamma protokollklassen (Se 4.6).

```
1  public async void Update(object sender, ElapsedEventArgs e)
2  {
3      sc = new StringContent(JsonConvert.SerializeObject(carClient));
4
5      try
6      {
7          response = await client.PostAsync(this.uri, sc);
8
9          if ((int)response.StatusCode == 409)
10         {
11             // 409 == Conflict, i.e. Ice nearby
12         }
13     }
14     catch (HttpRequestException err)
15     {
16         Output.WriteLine("\nException Caught!");
17         Output.WriteLine("Message :{0} ", err.Message);
18     }
19 }
```

Kodlistning 8: Async funktionen `Update` för sändning till server.

Funktionen `Update` (Se 8) åberopas kontinuerligt varje sekund. Notera att klienten använder sig utav en *async/await* anropningsparadigm.

5.2.2 Server

REST-lösningen innefattar också en monofilig backend skriven i programmeringsspråket Go.

```go
func main() {
  http.HandleFunc("/", root)
  http.HandleFunc("/update", update)
  http.HandleFunc("/fetch", fetch)

  // Default case, i.e. `.\server.exe`
  if len(os.Args) == 1 {
    fmt.Println("Listening on port 10000")
    log.Fatal(http.ListenAndServe(":10000", nil))
  }

  // Alternative case using parameter as port, e.g. `.\server.exe 1234`
  fmt.Println("Listening on port " + os.Args[1])
  log.Fatal(http.ListenAndServe(":"+os.Args[1], nil))
}
```

Kodlistning 9: Kärnan i REST-lösningens server som sammankopplar HTTP routes till respektive funktion.

Lite hjälpfunktioner skrevs i servern för att underlätta Test 2 som behöver flertalet klienter.

```go
// Helper function to update server representation of clients (cars)
//
// Uses globals: clients
// Returns:
//  true  - existing client is updated
//  false  - the new client is added to the global slice
func updateClient(currentClient Car) bool {
  for i := range clients {
    if clients[i].ID == currentClient.ID {
      clients[i] = currentClient
      return true
    }
  }

  clients = append(clients, currentClient)
  return false
}
```

Kodlistning 10: Hjälpfunktionen `updateClient` för Test 2.

```
// Helper function that determines if ice location is already
// known or not. Solution is O(n) in worst case because each
// element has to be checked.
//
// Uses globals: iceLocations
// Returns:
//  true  - ice location already known
//  false - the new ice location is added to the global slice
func updateIce(gps GPS) bool {
  for i := range iceLocations {
    if iceLocations[i] == gps {
      return true
    }
  }

  iceLocations = append(iceLocations, gps)
  return false
}
```

Kodlistning 11: Hjälpfunktionen `updateIce` för Test 2.

```
// Helper function that determines if a gps is close to any
// known ice location.
//
// Uses globals: safeIceLat, safeIceLong, iceLocations
// Returns:
//  true, coord  - gps is within distance to ice at `coord`
//  false, coord - gps is not within distance to ice
func withinIceDistance(gps GPS) (bool, GPS) {
  for i := range iceLocations {
    if math.Abs(iceLocations[i].Latitude-gps.Latitude) <= safeIceLat {
      if math.Abs(iceLocations[i].Longitude-gps.Longitude) <= safeIceLong {
        return true, iceLocations[i]
      }
    }
  }

  return false, GPS{Latitude: 0.0, Longitude: 0.0}
}
```

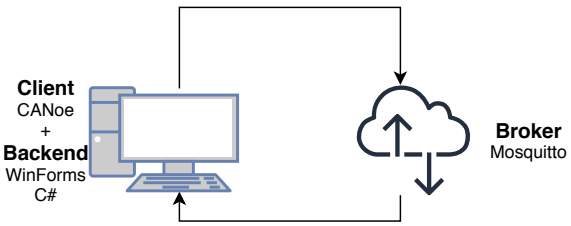
Kodlistning 12: Hjälpfunktionen `withinIceDistance` för Test 2.

5.3 MQTT-lösningen

Den sista demo-versionen simpleClient 0.4 blev slutresultatet av hela delen för MQTT-lösningen. MQTT-lösningen täckte den ena hälften av målet, samt uppfyllde de resterande delmålen som sattes. Slutligen utvecklades två tester för vardera protokoll för att styrka undersökningen med arbetet.

5.3.1 Kommunikationskedjan

Efter att ha diskuterat upplägget av hur testerna skulle genomföras, så beslutades att allting skulle köras på en och samma dator. Detta för att minimera externa variabler som kan påverka mätdata vid testning.

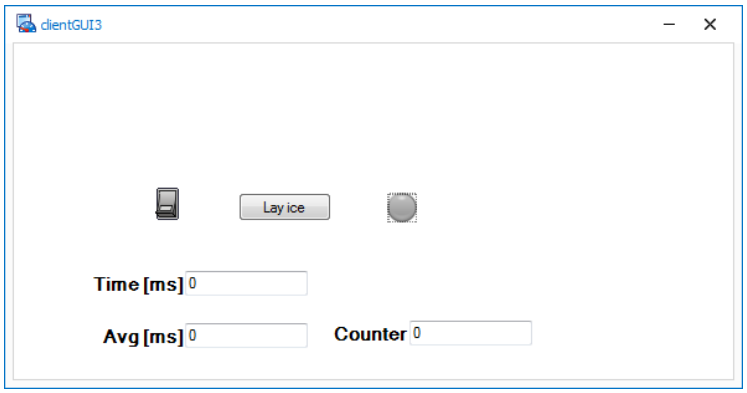


Figur 5.2: Slutresultatet blev interaktion mellan klient, backend och broker i en och samma dator.

Att uppdatera kommunikationskedjan (Se Figur 5.2) så att enheterna körde på localhost gav ett idealt förhållande för round-trip time-testerna, då den effektiva latensen på localhost är obefintlig.

5.3.2 Klientapplikation

Fjärde demot av lösningen blev slutresultatet för klientapplikationen. Det innehöll funktioner från tidigare demon, men var även utrustad för testning (Se Figur 5.3). Mot finalen las primära fokuset på att förbättra tidmätningarna av dataskickandet. Med den slutgiltiga kommunikationskedjan 5.2 kunde klientapplikationen börja mäta optimala round-trip time-värden.



Figur 5.3: Slutresultat av klientapplikationen, ämnad för testning.

Klientapplikationen (Se Figur 5.3) hade knappen `Lay ice` för att kunna starta en funktion som simulerar att en isfläck nyligen påträffats av fordonet. Detta för att starta och mäta hur lång tid det tar för ett MQTT-meddelande att färdas, processeras av backend, och slutligen mottaga ett varningsmeddelande i klientapplikationen (föraren). I samband med knappen fanns tre `Output Box` som visar tid och genomsnittlig tid för meddelandets round-trip time mätt i millisekunder. Sista rutan visar antalet manuella meddelandeskick, dvs antalet klickningar på knappen som gjorts.

Sammanfattningsvis hade klientapplikationen stöd för knapptryck vid manuell testning, men dessa kunde snabbt bytas ut mot de automatiska `Stopwatch`-testerna som genomfördes. Följande testkod 13 användes.

```
[OnChange(typeof(Panels.switchButton))]
public void switchValues()
{

    ...

    sw.Start(); //Start stopwatch measurement
    client.Publish("Cars/status", // topic
       Encoding.UTF8.GetBytes(json), // message body
       MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE, // QoS level
       false); // retained
}

public void client_MqttMsgPublishReceived(
    object sender,
    MqttMsgPublishEventArgs e)
{
    sw.Stop(); //Stop stopwatch measurement
    File.AppendAllText(logFile, responseID
        + "," + sw.ElapsedTicks + "\n");
    responseID++;
    sw.Reset();
}
```

Kodlistning 13: Integration med GUI knapp.

Testkoden
`sw.Start() ... sw.Stop()`
genererade en genomsnittlig tid på `0.18` millisekunder (p99) för första testfallet. Mer detaljerad info beskrivs nedan för respektive plottad graf.

5.3.3 Backend

I slutresultatet för backend sköttes primärt två arbetsuppgifter. Prenumerera på `Cars/*`, samt publicera varningsnotifkationer på `Cars/danger` (Se Figur 14).

```
public SimpleBackend()
{
    client = new MqttClient("localhost"); //local Mosquitto
    client.Connect(Guid.NewGuid().ToString()); //bind
    client.Subscribe(new string[] { "Cars/*" },
        new byte[] { MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE });

    ...
}

    ...

void client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e)
{
    ushort msgId = client.Publish("Cars/danger", // topic
     Encoding.UTF8.GetBytes("Danger message"), // message body
     MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE, // QoS level
     false); // retained
}
```

Kodlistning 14: SimpleBackend lyssnar på `Cars/status` samt publicerar på `Cars/danger` vid fara.

I ett försök att simplifiera backenden avgränsades funktionaliteten till ismarginal på scenarion med endimensionella positioner. Någon konstruktion av flerdimensionella positioner implementerades då ej.

5.4 Implementationsskillnader & komplexitet

Under projektets genomförande skedde utveckling på de två lösningarna parallellt. Särskilt implementeringsdelen är specifikt intressant då bristen på detaljerad kravspecifikation ledde till att lösningarna tog olika tillvägagångssätt på själva varningstjänsten.

5.4.1 REST backendlogik

1. Ta emot klientinformation: position + isdata.
2. *Om* klienten hittat is, lägg till klientpositionen i en lista med kända isfläckar, samt **avsluta requesten** med en HTTP error code.
3. *Annars*, iterera genom listan över kända isfläckar.
4. *Om* klienten är nära en känd isfläck i listan, **avsluta requesten** med en HTTP error code.

5. *Annars*, då inga närliggande isfläckar hittats, **avsluta requesten** med en HTTP OK.

Man kan snabbt se att ju fler isfläckar denna lösning lagrar, desto längre tid krävs för itereringen, och därmed ökas responstiden linjärt. Detta är lösningarnas huvudskillnad, och anledningen till att vidare tester som t.ex. skulle mäta round-trip time inte utfördes.

Framförallt berörs komplexitetsskillnaden då REST-lösningen tar hänsyn till alla tidigare isfläckar. Det genomsnittliga fallet för linjärsökning är $\theta(\frac{n}{2})$, vilket då blir $\theta(n)$. Linjärsökningen är också $\mathcal{O}(n)$ i värsta fallet.

5.4.2 MQTT backendlogik

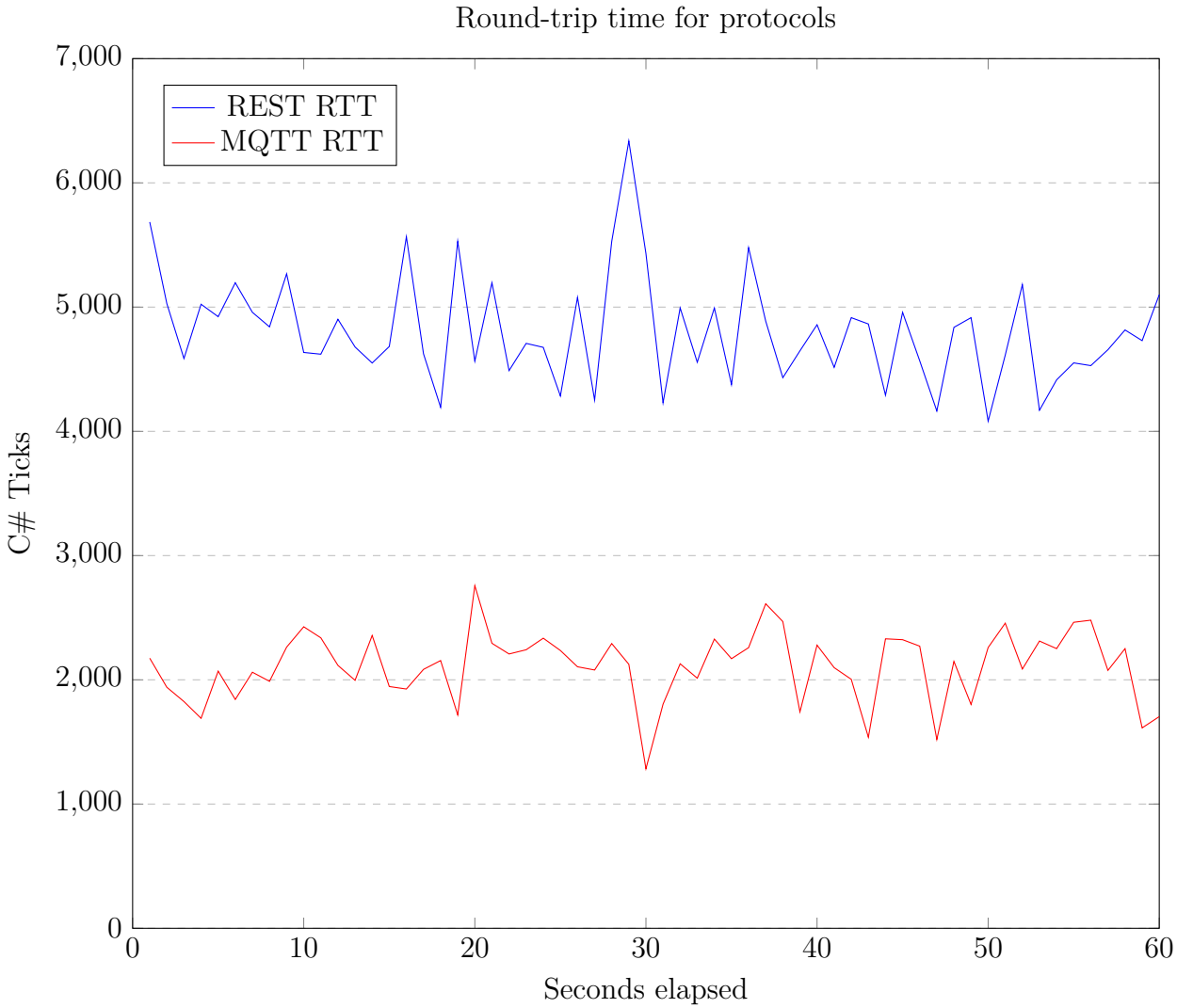
1. Ta emot klientinformation: position + isdata.
2. *Om* klient hittat is, spara positionen som känd isfläck, samt skicka varning till klienten.
3. *Annars*, kolla om klienten är nära den kända isfläcken.
4. *Om* klient inom varningsintervall, skicka varning till klienten.

Denna backendlösning implementerad via MQTT tar endast hänsyn till den senast lagrade isfläcken, och ersätter denna vid ny upptäckt isfläck. Med andra ord kan lösningen endast hålla koll på en isfläck, tills den blir överskriven av en nyare.

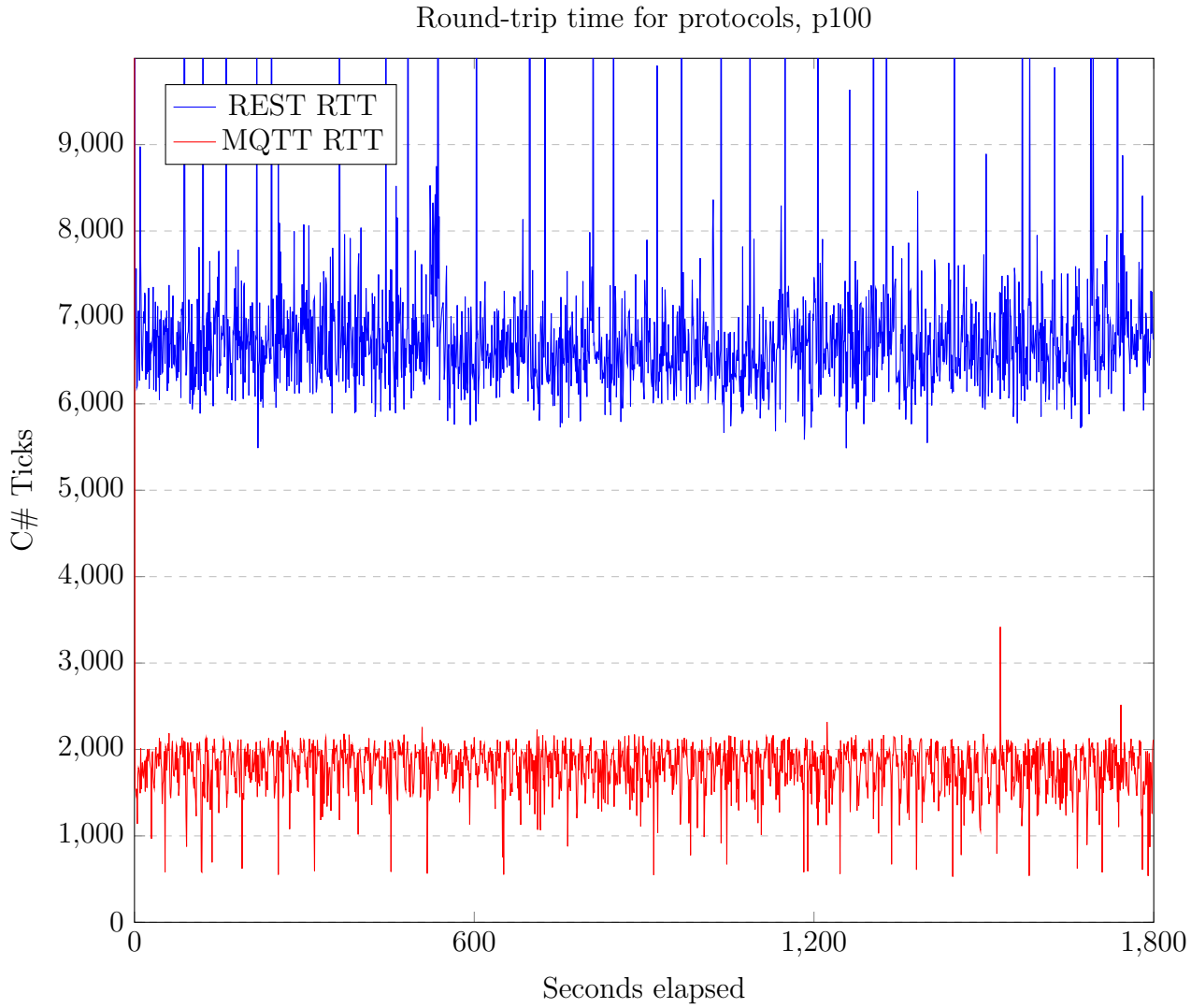
Eftersom det bara finns ett värde sker ingen sökning, och det uppfattas då tydligt att denna lösning är $\theta(1)$ i genomsnittliga fallet, men även $\mathcal{O}(1)$ i värsta fallet.

5.5 Testningsgrafer

Testresultaten mätt i round-trip time kan plottas som grafer. Nedan följer utvalda grafer som ansågs vara de mest väsentliga resultaten.

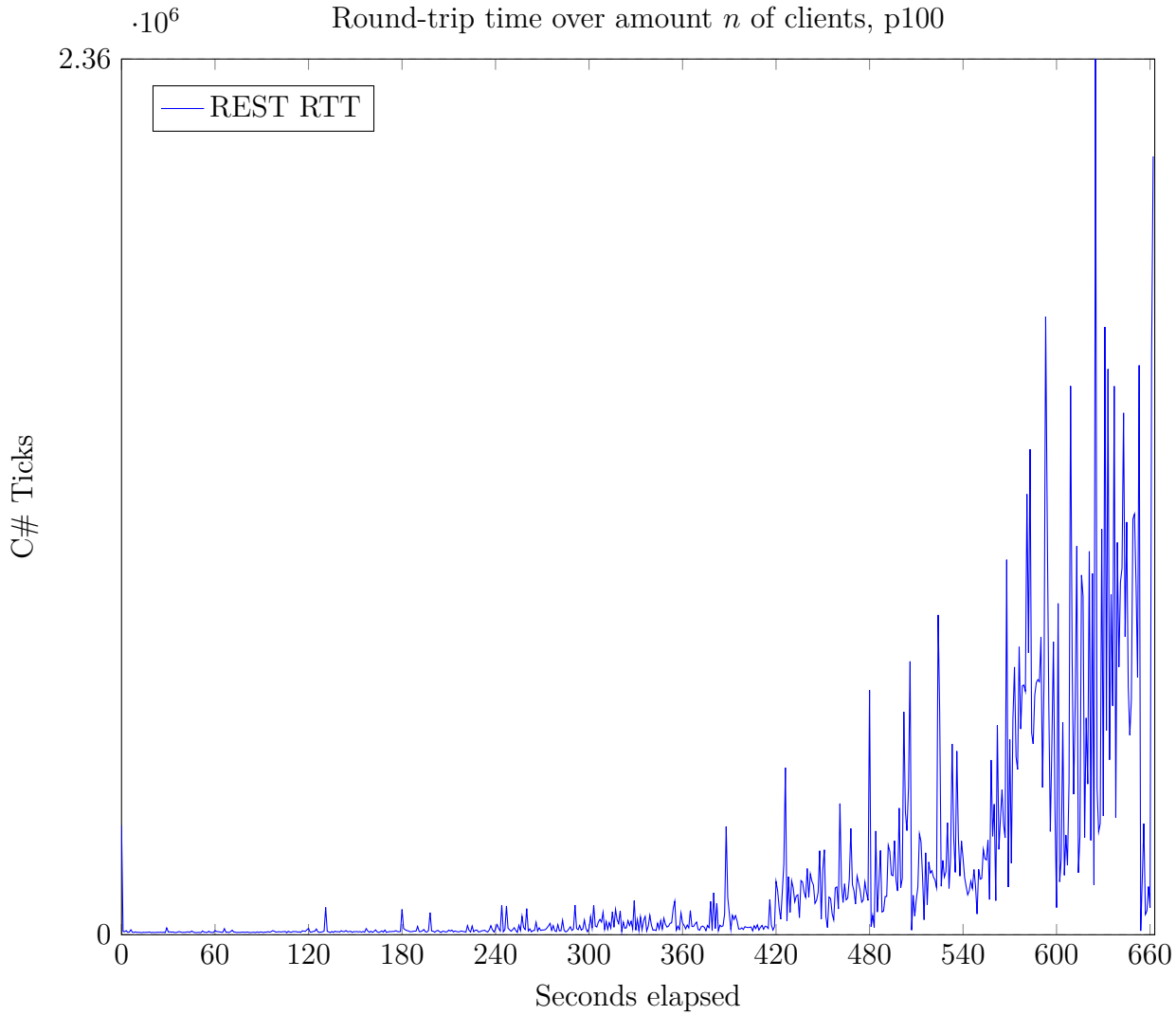


Första resultatet var definierad enligt första testfallet, dvs en klient round-trip time, även kallat `Test 1: Simple Latency`. Testerna som tidigare nämnt, baseras på Monte Carlo-metoden, där första testet för båda lösningarna exekverades under 60 sekunder. Både REST- och MQTT-lösningen testades på första testriggen (4). Graferna visar tydligt att MQTT har en lägre genomsnittlig round-trip time än REST-lösningen på den specifika testriggen, där REST har ungefär dubbla MQTTs round-trip time. Vidare antyder graferna att MQTT-lösningen har lägre svängningsdelta från mätpunkt till mätpunkt, med andra ord stabilare än REST-lösningen.



Figur 5.4: Test 1 - 30 min (p100)

Som man kan se på grafen så visar REST-lösningen tydligt intermittenta latensspikar då och då. Detta beror troligtvis på backenden, då den är skriven i Go som har inbyggd garbage collection. Bl.a. företaget Discord skrev en artikel där de beskrev problemet och hur de löste det (i deras fall bytte de språk till Rust) [10]. I senare versioner av Go finns nu fler sätt att hantera garbage collection, men för detta projekt är det mer troligt att generell backendoptimering är ett större problem.



Figur 5.5: Test 2 för REST-lösningen

I grafen 5.5 skildras den iterativt ökande belastningen med antal klienter (bilar) som tvåpotenser. Var 60:e sekund förändras antalet klienter från 2^x till 2^{x+1}. Här kraschar CANoes testnod efter 3 sekunder med 2048 klienter.

REST-lösningen hade — enligt round-trip time testen på första testriggen — en bibehållen genomsnittlig responstid på `0,4797 ms`, medan MQTT-lösningen under samma förhållanden hade `0,2146 ms`. Som även kan överskådas i figuren 5.4 uppvisade MQTT-lösningen lägre latens med en ungefär faktor `2,235`.

Antal klienter...	...i tvåpotensform	REST-lösning	MQTT-lösning
1	2^0	✓	✓
2	2^1	✓	✓
4	2^2	✓	✓
8	2^3	✓	✓
16	2^4	✓	✓
32	2^5	✓	✓
64	2^6	✓	✓
128	2^7	✓	✓
256	2^8	✓	✓
512	2^9	✗	✓
1024	2^{10}	✗	✓
2048	2^{11}	✗	✓
4096	2^{12}	✗	✓

Tabell 5.1: Test 2 (round-trip time flera klienter) på första testriggen

När en lösning kan hantera ett visst antal klienter, så gäller det över en period på 20 minuter. Tabellen 5.1 visar att första testriggen, dvs Vectors stationära dator, klarade upp till 256 klienter. Intressant kan vara att se att datorn kraschade på 512 klienter, men detta efter 4 sekunder.

Antal klienter...	...i tvåpotensform	REST-lösning	MQTT-lösning
1	2^0	✓	✓
2	2^1	✓	✓
4	2^2	✓	✓
8	2^3	✓	✓
16	2^4	✓	✓
32	2^5	✓	✓
64	2^6	✓	✓
128	2^7	✓	✓
256	2^8	✓	✓
512	2^9	✓	✓
1024	2^{10}	✗	✓
2048	2^{11}	✗	✓
4096	2^{12}	✗	✓

Tabell 5.2: Test 2 (round-trip time flera klienter) på andra testriggen

Ett eventuellt mönster upptäcktes där CANoe noden krashar med 2^x klienter, för att sedan nästa gång klara det och istället krasha på 2^{x+1} klienter. Detta gäller över båda testriggar.

6
Slutsats och Diskussion

Slutprodukterna av MQTT- och REST-lösningarna indikerar att det är möjligt att tillämpa V2C-teknologi via dessa implementerade protokoll. Resultaten påvisar vår hypotes om att MQTT-lösningen skulle vara snabbare än REST-lösningen, för denna begränsade utvärdering. Projektet bekräftade även den andra hypotesen, att REST-lösningen skulle gå snabbare att implementera.

Då MQTTs och RESTs kommunikationsparadigmer skiljer sig från grunden, så var det naturligt att även deras slutgiltiga lösningsarkitekturer som designades gjorde likadant. MQTT använder sig utav publish-subscribe pattern, medan lösningar som designas utefter REST tillämpar vanligtvis client-server-modellen. MQTT använder topics vid kommunikation mellan broker och klienter. Observera att MQTT-lösningens backend är även den en MQTT-klient, vilket betyder att även om MQTT-lösningen hade efterliknat REST-lösningens funktionalitet, som är mer komplex, för tillämpning av -logik, så hade detta inte påverkat brokern med en ökad belastning. Eftersom handlar om en generell teknologi inom fordonskommunikation som är öppen för alla, så hade det varit möjligt att utveckla utanför CANoe. Men fördelen med att ha utvecklat lösningen i CANoe är att om det funkar i simulerad miljö, så kan detta koncept deployas smidigt till en riktig bil. En annan fördel är att man kan nyttja integrationstester direkt i lösningar som skrivs i CANoe-projekt, t.ex. GPS-datan som togs genom en verklig bilfärd.

Dock medför en formad anpassning att man missar eventuell optimering i val av programmeringsspråk för klienter. Då klienterna implementerades som CANoe noder kan de endast implementeras i CAPL, alternativt C# med .NET Framework. Bland annat kan inte .NET Core användas för att nyttja plattformsagnostisk kod, även om det inte var nödvändigt för detta projekt.

Backend för respektive lösning fungerar fristående och är inte bunden till CANoes CAPL- eller C#-begränsningar. Det är inte nödvändigtvis sämre att klienterna var utvecklade i CANoe — som är låst till CAPL eller C# — men om det t.ex. hade uppdagats att det finns uppenbara bättre eller alternativa bibliotek för andra språk, så hade klienterna inte varit modulära nog att byta språk för att nyttja dem. Istället hade det krävts 'bindings' eller 'interop' till det biblioteket från C#.

MQTT-lösningen hade en lägre latens med en faktor på c.a. `2,235`.

Projektets kodmässiga struktur samt dokumentation håller enligt oss en god standard. Under arbetets process har tid lagts på att rigoröst säkerställa att de viktigaste principer för en ren kodbas har upprätthållits.

I avgränsningar tas det upp att projektet inte kommer hantera användning av verkliga fordon; däremot definieras en av de motiverade delmålen i syfte om att utreda latenskillnaderna i lösningarna. Detta i syfte att avgöra om latenserna är tillräckligt låga för en produktionsredo prototyp.

En jämförelse mellan protokollen var oerhört komplex, då skillnaderna i arkitektur, kommunikationsparadigm, samt metod för implementering av kommunikationsprotokoll är markant. Det stämmer att MQTT-lösningen har en lägre genomsnittlig latens än REST-lösningen. Däremot kommer faktorerna i en verklig miljö vara avsevärt annorlunda. Framförallt med avseende på GSM-nät, fördröjningar, täckningsproblem, och hur protokollen kan mitigera liknande bekymmer.

Att CANoe anslutningen till servern bröts efter 5 sekunder med 512 anslutningar under REST testfall 2 kan innefatta, men är inte begränsat till, följande faktorer.

1. Okända begränsningar i CANoe.
2. TCP begränsningar från operativsystemet (Windows 7 SP1).
3. Prestandabegränsningar från hårdvaruresurser (främst CPU/RAM).
4. Begränsningar i serverns programspråk (Go).
5. Begränsningar i serverns logik (ooptimerad kod).
6. Begränsningar i WiFi chippet om trafiken routades där trots localhost.

Då testerna utförda på andra testriggen klarade betydligt fler REST klienter är 2 och 3 starkt troliga faktorer. 1, 4, samt 5 är mindre troliga då de är statiska variabler över testriggarna. 6 är med stor sannolikhet övegripande irrelevant.

För testfall 2 (latenstest med flera klienter) kraschade REST-lösningen på andra testriggen i genomsnitt inom 10 sekunder med 2048 klienter, där klienterna är ECU:er (simulerade bilar). Dock är det simulerade scenariot inte nödvändigtvis verklighetstroget. En produktionsmiljö hade med stor sannolikhet haft mitigerande tekniker för att förebygga liknande krascher, därbland främst lastbalanserare mellan olika backends. Vidare mitigation kan innebära dynamisk uppdateringshastigheter hos klienterna. Just detta testfall konstruerades även ur ett pragmatiskt ad-hoc perspektiv. Antagelser likt att alla klienter skulle anslutas till samma backend och samtidigt skicka ut uppdateringar med liknande intervall är orimliga för en verklig produktionsmiljö.

6.1 Testfallen

Utifrån graferna hade REST-lösningen en genomsnittlig långsammare responstid. Detta beror med största sannolikhet på olika optimeringsgrader på serversidan, som sköter själva meddelandehanteringen. Där Mosquitto (MQTT-lösningens broker) har

funnits sen 2009, och har mer än 1880 commits över 84 unika projektdeltagare [7] så är REST-lösningens server utvecklad av oss under projektets gång i programmeringsspråket Go, som vi inte tidigare varit bekanta med. Det är rimligt att anta att REST-lösningens server inte är optimerad för sitt ändamål.

6.2 Kritisk diskussion

Fördelningen av delmål och milstolpar över löpande sprints var olikvärdigt fördelade, vilket medförde att vissa arbetsuppgifter kunde ändras med tiden, som gjorde det svårt att bedöma om något var fullbordat eller inte. Framförallt förverkligades inte den godkända tidplanen, utan projektet löpte långt mycket längre än väntat. Detta projektarbete jämför två protokoll som är olika i sin natur, vilket gjorde det svårt att rättvist jämföra.

Troligtvis var den inledningsvis noggranna user story hanteringen för mycket overhead för ett tvåmanna exjobb med dessa avgränsningar. Detta resulterade i att hela systemet för 'issues' försakades efter några veckor. En bättre metod hade varit mer lättöverskådlig och hanterbar user story planering.

Ett alternativt utvecklingsflöde hade involverat parvis design och konstruktion av lösningarna, så att de fungerade på samma sätt (Se 5.4).

6.3 Miljöperspektiv & etik

En deployad V2C varningstjänst hade möjligtvis resulterat i mer förberedda förare inför isfläckar, och då troligtvis avsevärt minimerat dagens bromssträckor förknippade med dessa. Förare som inte blir överraskade och panikbromsar hade kunnat bespara bensin, slitage, och skador i både hälsa och fordon. Därav hade tjänsten även kunnat bidra till Trafikverkets Nollvisionen [16].

Om tjänsten under fortsatt utveckling konstruerats som en öppen standard, som fritt kunde implementerats, så hade en betydligt större del av befolkningen kunnat dra nytta av säkrare vägar.

6.4 Förbättringar för vidareutveckling

Protokollmässigt verkar MQTT bäst lämpad för projektet. Däremot borde RabbitMQ med AMQP utvärderas, då dess message queueing erbjuder en annan typ av stabilitet som kan vara önskvärd. Funktionalitetsmässigt är implementationerna väldigt lättviktiga. En förare hade definitivt önskat mer granularitet över isvarningar än bara *säkert* och *fara*. För att kunna tillämpa varningssystemet i verklig miljö så krävs en närmare koppling till reell hårdvara, vilket var avgränsat från detta projekt.

Litteratur

[1] V. Borges. *Terminology Explained: P10, P50 and P90*. DNV GL. 2016. URL: https://blogs.dnvgl.com/software/2016/12/p10-p50-and-p90/ (hämtad 2020-03-13).

[2] *CANoe | ECU & Network Testing on Highest Level*. Vector Informatik. URL: https://www.vector.com/int/en/products/products-a-z/software/canoe/ (hämtad 2020-01-20).

[3] *CAPL Scripting Quickstart*. Vector Informatik. URL: https://assets.vector.com/cms/content/products/VectorCAST/Events/TechNights/CAPLQuickstart_Generic_2018_Final.pdf (hämtad 2020-02-22).

[4] *Car2x communication.jpg*. 2020. URL: https://commons.wikimedia.org/wiki/File:Car2x_communication.jpg (hämtad 2020-03-15).

[5] *Connected and Automated Vehicles*. Center for Advanced Automotive Technology. URL: http://autocaat.org/Technologies/Connected_and_Automated_Vehicles/ (hämtad 2020-01-09).

[6] *Connected Vechicle Cloud Platforms*. ABIresearch. URL: https://www.abiresearch.com/market-research/product/1022093-connected-vehicle-cloud-platforms/ (hämtad 2020-01-09).

[7] *Eclipse Mosquitto. An open source MQTT broker https://mosquitto.org*. Eclipse Foundation. URL: https://github.com/eclipse/mosquitto (hämtad 2020-03-12).

[8] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures". Diss. University of California, Irvine, 2000. Kap. 5.1, s. 76–85. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (hämtad 2020-02-15).

[9] G. C. Hillar. *MQTT Essentials - A Lightweight IoT Protocol*. 1st Edition. Packt Publishing, 2017. ISBN: 9781787285149.

[10] J. Howarth. *Why Discord is switching from Go to Rust*. Discord Inc. URL: https://blog.discordapp.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f (hämtad 2020-02-04).

[11] *M2Mqtt & GnatMQ. MQTT Client Library & Broker for .Net platform*. M2MQTT. URL: https://m2mqtt.wordpress.com/ (hämtad 2020-02-28).

[12] *M2Mqtt & GnatMQ. MQTT Client Library & Broker for .Net platform*. M2MQTT. URL: https://m2mqtt.wordpress.com/ (hämtad 2020-02-28).

[13] *MQTT Client Library Encyclopedia*. HiveMQ. 2020. URL: https://www.hivemq.com/mqtt-client-library-encyclopedia/ (hämtad 2020-02-15).

[14] *MQTT Websocket Client*. HiveMQ. 2020. URL: http://www.hivemq.com/demos/websocket-client/ (hämtad 2020-02-15).

[15] *MQTT Version 5.0. OASIS Standard*. Organization for the Advancement of Structured Information Standards. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf (hämtad 2020-03-05).

[16] *Nollvisionen*. Transportstyrelsen. 2020. URL: https://www.transportstyrelsen.se/sv/vagtrafik/statistik/olycksstatistik/statistik-over-vagtrafikolyckor/nollvisionen/ (hämtad 2020-03-15).

[17] C. Pease. *An Overview of Monte Carlo Methods. OASIS Standard*. 2018. URL: https://towardsdatascience.com/an-overview-of-monte-carlo-methods-675384eb1694 (hämtad 2020-03-13).

[18] *Publish & Subscribe - MQTT Essentials: Part 2*. HiveMQ. URL: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/ (hämtad 2020-02-28).

[19] *Stack Overflow Developer Survey 2019*. Stack Exchange, Inc. URL: https://insights.stackoverflow.com/survey/2019/ (hämtad 2020-03-03).

[20] *What is Agile?* Agile Alliance. 2020. URL: https://www.agilealliance.org/agile101/ (hämtad 2020-03-15).

[21] *VN5610A/VN5640. Powerful and Multifunctional USB Network Interfaces for Automotive Ethernet and CAN FD*. Vector Informatik. URL: https://www.vector.com/int/en/products/products-a-z/hardware/network-interfaces/vn56xx/ (hämtad 2020-03-11).