UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

DATA MANAGEMENT FOR CLOUD SUPPORTED COOPERATIVE DRIVING

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

Miguel Ângelo Luís Lourenço

Dissertação orientada por:
Prof. Doutor António Casimiro Costa
Prof. Doutor Alysson Bessani

2020

Acknowledgements:

I am extremely grateful to the Fundação para a Ciência e Tecnologia (FCT) that supported this work through the projects AQUAMON PTDC/CCI-COM/30142/2017 and IR-CoC PTDC/EEI-SCR/6970/2014.

I would also like to extend my deepest gratitude to my supervisors who have always supported me and pointed me the right way: Prof. Alysson Bessani and Prof. António Casimiro Costa, even through these hard times. This thesis would not have been concluded without their guidance, knowledge and availability.

Many thanks to my family and friends who taught and supported me to this day, I am the person who I am now thanks to all of you.

I also had great pleasure of working with my colleagues and friends throughout these years in FCUL, from great conversations to group projects, you know who you are.

A special thanks to FCUL for taking me in through my bachelor's and master's and LASIGE for the amazing people and workspaces, you facilitated my progress and always strove for my success.

Resumo

O aumento de tecnologia inserida em veículos permitiu ao utilizador comum ter acesso a um leque crescente de funcionalidades para tornar a condução mais fácil, económica e segura. ABS, GPS, Bluetooth, computador de bordo são algumas das tecnologias associadas a um veículo recente. No caso de veículos mais experimentais, já são associadas tecnologias de deteção de obstáculos e de travagem e condução automáticas. Apesar de experimental, a condução automática tem sido bastante cobiçada e desenvolvida nos últimos anos, sendo a ligação e troca de dados entre veículos uma das vertentes dessa tecnologia. Este aspeto é essencial para o desenvolvimento de um sistema de veículos autónomos, transformando um veículo autónomo independente num nó de informação para um vasto sistema de veículos interligados. Este será o desafio que esta dissertação irá atacar: a possibilidade de criar um sistema fiável de gestão de dados de veículos autónomos baseado na nuvem. O armazenamento de dados na nuvem permite suportar um número varíavel e indefinido de veículos e facilita o acesso aos dados, delegando no provedor da nuvem o ónus da manutenção das infraestruturas físicas.

Este sistema vai ser desenhado para que consiga suportar um cenário em que vários veículos autónomos ligados ao sistema, incluídos numa determinada localização geográfica, consigam obter informação sobre o ambiente que os envolve, bem como as intenções dos outros veículos ligados. Dessa forma eles conseguem comunicar e decidir sobre que ações tomar. O objectivo principal deste sistema é conseguir fazer com que os veículos cooperem em manobras cordenadas como por exemplo seguir em pelotão, quebrar formação ou concordar com manobras de ultrapassagem. Todos estes exemplos requerem uma comunicação rápida e correta. Idealmente, estes veículos estarão ligados ao sistema através de redes wireless à internet para conseguirem ter acesso a este mesmo sistema. O desafio principal será conseguir suportar um número elevado de veículos enquanto processa e serve dados atualizados e válidos aos veículos, para que os mesmos possam atuar num curto período de tempo e em segurança. Para esses dados serem válidos, estes terão que ser consistentes. Para conseguir garantir consistência, é necessário do sistema utilizar tempo para provar a consistência dos mesmos. Com isso é preciso chegar a um meio termo onde latência e garantia de consistência cheguem a um compromisso. Tomando em conta este cenário, foi-se definido que dois segundos seria a latência máxima permitida por cliente no sistema. Isto é devido a análises de trânsito definirem que um veículo

realiza uma manobra entre 5 a 7 segundos em determinados ambientes. Desse tempo decidiu-se que 2 segundos seria o tempo máximo de latência para tomar essa decisão, sobrando o resto do tempo para realizar a manobra.

Para saber quais serviços poderiam ser utilizados para hospedar a base de dados do sistema, cumprindo com o requisito definido, avaliou-se três serviços oferecidos pela Amazon, através da sua infraestrutura *Amazon Web Services* (AWS): S3, EC2 e DynamoDB; os aspetos avaliados foram o preço, a velocidade de resposta, a facilidade de utilização/implementação e escalabilidade, pois estes são os aspetos essenciais na utilização e manutenção de um sistema que fornece informação a veículos em tempo real. Após comprovar-se a viabilidade destes serviços comparando com o requisito estipulado, decidiu-se utilizar o serviço S3 devido à sua compatibilidade com uma biblioteca existente que consegue emular uma memória partilhada na cloud utilizando este serviço.

Após ter-se decidido qual o serviço mais apropriado, avançou-se para a criação de dois modelos de clientes. A camada de acesso dos clientes para a base de dados foi baseada no algoritmo *Two-Step Full Replication* que utiliza os serviços de armazenamento de objetos através de *Key-Value Stores* de várias nuvens em simultâneo, simulando uma memória partilhada com registos *multi-writer*, *multi-reader*. Este algoritmo tolera falhas bizantinas através de técnicas de quorum bizantino e de medidas de verificação de integridade e autenticidade. Foram definidas e implementadas as necessárias alterações para que o algoritmo servisse melhor os clientes que acedem este sistema de gestão de dados.

O primeiro modelo, chamado "*Atomic Snapshot Client*", utiliza uma implementação do *Two-Step Full Replication* modificada com o algoritmo de *Atomic Snapshots* como interface do cliente. Este modelo garante que uma leitura completa do sistema (*snapshot*) é feita de forma atómica, ou seja, fornece um *snapshot* inadulterado por escritas concorrentes, a custo de tempo de resposta. O segundo modelo é uma versão mais rápida do primeiro com o objetivo de obter respostas mais rápidas sem ter que sacrificar muito na consistência dos dados. Os dados obtidos neste cliente têm garantia de consistência regular. Este cliente é designado "*Fast Snapshot Client*". As alterações principais foram diminuir as garantias dos registos atómicos para regulares na camada de acesso à base de dados, tornando as operações de leitura dos registos (*scan*) e atualização dos mesmos (*update*) muito mais simples e rápidas e a mudança da interface do cliente em que se removeu a utilização do algoritmo que fornece *snapshots* atómicos. Este algoritmo foi substituído por invocações básicas de escrita e leitura em quórum. Com a análise dos dados obtidos destes modelos foi possível observar uma relação entre o aumento no tempo de *scan* e o aumento do tempo total gasto na execução de várias operações de escrita e leitura numa aplicação constituída por vários clientes. Para resolver este problema implementou-se um *garbage collector* simples que limpa cada registo após este ultrapassar um determinado número de escritas antigas. Esta solução, apesar de simples, mostrou-se efectiva para baixar e estabilizar o tempo de execução de cada operação que envolve a chamada de um

scan. A resolução desse problema revelou a existência de outro, em que a latência do *scan*, apesar de estabilizada ao longo do tempo de vida do sistema, mantinha-se elevada em sistemas com bastante clientes. A solução encontrada foi atribuir um número de threads à *thread pool* dedicada a fazer *scans* do cliente igual ao número total de clientes que o sistema suporta. Desta forma conseguimos garantir que a latência obtida seja mais baixa que utilizando um número fixo de *threads*.

Por fim, implementou-se um sistema de gestão de dados baseado no serviço S3 da AWS, constituído por dois tipos de clientes baseados no modelo *Fast Snapshot Client*, denominados por *vehicular client* e *calculator client*. Os dois tipos de cliente trabalham em conjunto onde os *vehicular clients* (veículos) partilham informação com o *calculator client*. O *calculator client* faz *scan* aos valores dos registos dos *vehicle clients*, processa os dados e escreve nos seus registos informação relativa à análise desses mesmos valores. As funções dos *vehicle clients* são escrever dados relevantes obtidos dos seus sensores e ler a informação do registo do seu *calculator client* respectivo e agir de acordo com os dados lidos, caso sejam válidos.

Para obter dados sobre este sistema, cada cliente foi testado individualmente de forma a obter a latência de execução das suas funções. Cada tipo de cliente foi submetido a um teste com um tempo de execução de cinco minutos onde se testou a latência de execução das suas duas funções de comunicação com a memória partilhada baseada na cloud com *n* números de *vehicle clients* associados. Dos dois tipos de cliente, confirmou-se que a latência do *calculator client* está dependente do número total de clientes associados ao sistema e que um sistema deste tipo consegue abranger entre 0 a 750-1000 veículos.

Após esses testes confirmou-se a possibilidade deste sistema suportar um elevado número de veículos cumprindo um requisito de latência máxima de 2 segundos. Com esta conclusão também propôs-se possíveis melhorias no sistema para poder ser utilizado numa situação mais real e a possibilidade de implementação deste sistema baseado noutro tipo de serviço que oferece leituras com uma latência mais baixa que o serviço usado.

Palavras-chave: Gestão de dados, Memória Partilhada, Nuvem, Tempo Real, Condução Autónoma

Abstract

The increasing number of technologies inserted into vehicles, allowed the common user to have access to a broad number of utilities that allows driving to be easier, safer and more economical. ABS, GPS, Bluetooth and onboard computer are some of the technologies associated with a recent vehicle. On more experimental ones there is obstacle detection, automatic braking and self-driving technologies, which can be supported by a wireless network connection to further improve their capabilities. That connection allows the transformation of each independent vehicle into nodes in an ad-hoc network. The current challenge is to connect all those vehicles and be able to provide the data needed for their correct functioning in a timely manner. That is the challenge this dissertation will seek to analyse: the possibility to create a reliable vehicular information system for cooperative driving based on the cloud. Cloud-based storage can support an ever changing number of vehicles while still satisfying scalability requirements and maintaining ease of access without the need to maintain a physical infrastructure, as that responsibility is laid upon the provider. To understand which service is the best to host the vehicular information system it was analyzed three services from Amazon Web Services (AWS): S3, EC2 and DynamoDB. Ease of utility, latency, scalability and cost were the main requirements tested as they are the most important aspects for a real-time vehicular information system for autonomous vehicles.

After deciding which cloud service would be the most appropriate to implement the vehicular information system, two client models were created that fulfilled a set of requirements. They were based in an already existing algorithm named Two-Step Full Replication which utilizes a group of Key-Value Stores services from various clouds to simulate a shared-memory based on multi-writer, multi-reader (MWMR) registers. This algorithm tolerates Byzantine faults by using Byzantine quorum techniques and integrity and authenticity checks. It was defined and implemented the necessary changes on the algorithm to create usable a client for a vehicular information system.

The first model called ”Atomic Snapshot Client”, uses the modified Two-Step Full Replication interface with the Atomic Snapshot algorithm. This model guarantees that the read of the system (snapshot) is done atomically without being adulterated by concurrent writes, sacrificing execution latency. The second model is a faster version of the first one with the objective of obtaining faster responses from the system without overly sacrificing

data consistency, which is called "Fast Snapshot Client". The main change from the first one is the reduction of the guarantees of the atomic registers to regular ones making the reads (*scan*) and writes (*update*) simpler and faster, although removing the atomic snapshot feature. With the analysis of the data collected from experiments performed with this model it was possible to observe a relation between the increase of the scan latency time and the total time spent on the execution of the read and write operations on an application with various clients. To solve this problem a simple garbage collector was implemented, which cleans each register when the number of outdated writes that it contains goes over a specified threshold. This solution, although simple, proved to be effective to reduce each scan time.

Finally, a vehicular information system based on the AWS S3 service was implemented. It is composed by two types of clients based on the Fast Snapshot Client, named vehicular client and calculator client. The two types of client work together, where the vehicular clients trade information with the calculator. The calculator client scans the registers of the vehicle clients and writes on its registers the processed data for each vehicular client. The vehicle clients need to write all the relevant data they gather and read the register of their respective calculator client and act according to the data read. Each of the clients was tested separately and analysed in order to discuss the viability of this system in a real-world application as well as possible changes to further improve it.

Keywords: Data Management, Shared Memory, Cloud, Real-Time, Autonomous Driving

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Creating fully functional autonomous vehicles is a challenge that has originated around the 1920's [3]. Over the years of development, the autonomous vehicles have had an increase of sensors, either in number and in types, to improve their decision-making [4]. The objective is to give the decision-making entity the maximum possible information, so it makes better decisions and avoids accidents on the road.

Besides using their collected information, it is possible to exchange sensor data between vehicles sharing the same area. For that to work they need to be interconnected in a Peer-to-Peer (P2P) manner or connected to a wireless network with access to the internet. Using the latter, it is possible to create a database for storing and accessing all the vehicles' information [5].

There are various choices on how to host and implement a database. Cloud service providers such as Amazon Web Services (AWS), Microsoft Azure and Google Cloud offer various database and storage services to their clients, where the maintenance and scaling responsibility is laid upon the provider, leaving the client to define its requirements. This solution is elegant as it can up or down-scale automatically for any kind of throughput it is subjected to.

However, for the database to be useful, it needs to give timely responses with fresh and correct data. The work presented in this thesis aims to explore the possibility to create an interface layer for accessing the cloud-based database, which adds mechanisms for interacting with the database to allow the mentioned timeliness and correctness requirements to be achieved when multiple clients want to read and write concurrently on the database.

1.1 Motivation

In a world where almost every household has a private four-wheeled vehicle, the streets have become more and more crowded. That implies more accidents, traffic jams and more time on the road overall. Even with the increase of technology being added like sensors, drive assistance and even self-driving, there are still some adjustments and improvements

needed for them to be reliable. Those cars can act independently or can join a wireless vehicular network to exchange data between themselves and work together to improve the traffic flow, avoid accidents and do maneuvers that require knowledge about other surrounding vehicles and obstacles [6].

Since most common communication technologies used on vehicular networks such as DSRC and LTE are able to connect to the internet [7] [8], it is possible to connect those vehicles to the cloud without any extra equipment and will allow them to connect to a widely available cloud service provider such as AWS and Microsoft Azure.

With that capability, it creates the opportunity to use cloud-based databases for sharing and storing vehicular data. However, having such clients writing and reading freely on the database creates a coordination problem which can affect data coherence. On the other hand, a system with lots of clients using strict coordination rules to act on the database will affect the system's performance negatively. This is the challenge that this dissertation will investigate by checking for existing mechanisms or solutions to access the database that will give enough performance while coordinating a large number of vehicles.

1.2 Goals

The overall objective of this work is to design and implement a solution to manage and support the transfer of vehicle data between vehicles and a cloud-based database service. The main challenge addressed was performance, considering that large amounts of data are produced and consumed over time, whilst needing to remain valid. Elasticity of the solution, to adapt to varying workloads, is also a challenge, if it is not already taken care of by the cloud service.

The basic idea followed was to separate concerns between processing and data storage/management. This means that a system was designed and built around a database that only provides data storage and leaves the processing to the clients. To reach this goal, four milestones were defined:

1. Obtain the latency values of the response on various cloud services using a simple client, in order compare them between each other and check if the raw latency is low enough, such that the cloud service acting as a database can be used for the cloud-based vehicular information system.

2. Take the most appropriate cloud service and use it to implement a fault tolerant and secure database, by adding a layer in which some protocols for interacting with the database are implemented, such that the required correctness properties are achieved while providing a simple client interface to ensure its simplicity and separation of concerns.

3. Test different solutions for the implementation of the client layer to retrieve the data through snapshots with different guarantees, such as the atomic and regular snapshot [2]. The latency on executing the basic database requests (such as *scan* and *update*) will be one of the deciding factors on choosing which solution is the best.

4. Take the most efficient solution, latency-wise, for the client layer with the strongest possible snapshot guarantees and adapt it to serve in a working cloud-based vehicular information system. To finally check its viability in a real world application, we make some tests to verify its capabilities as well as weak points.

We believe that, by reaching these four milestones, we can achieve the goal defined for this dissertation.

1.3 Contributions

On this project we propose a cloud-based vehicular information system based on one of the current big cloud service providers. The main goal of the vehicular information system is to be able to quickly provide processed data for all the participating automated vehicle clients so they can act in a safe manner.

To be able to prove the feasibility of our solution, we designed, implemented and analysed two different clients that can access the AWS S3 service as a database. They provide basic data access operations such as *read*, *write* and *scan*. These are sufficient to evaluate if the clients ensure low latency data access and validity of data. Using the best client under the evaluation parameters, we implemented it in the proposed vehicular information service, analysed and discussed its viability.

Finally, we also discuss the possibility to use the DynamoDB service as the vehicular information service database as well as some possible improvements on the service functionalities.

1.4 Work Plan

The project was executed in accordance with the following plan:

- September 2019 – October 2019: Study of the existing platforms and services for cloud-based databases and for vehicle coordination. Study of the state of the art on vehicular and wireless network protocols, cloud architecture, shared memory abstractions and related projects.

- November 2019 – December 2019: Design and test simple clients on various Amazon Web Services (AWS) to obtain latency values and compare them between each other.

- January 2020 – February 2020: Develop two different clients to access the preferred service as a shared memory abstraction.
- March 2020: Response latency and performance evaluation of the previous clients.
- April 2020 - June 2020: Use the best client implementation to test its capability in a working cloud-based vehicular information system and discuss the results.
- July 2020 - October 2020: Thesis preparation.

1.5 Structure of the Document

This document is organized as follows:

- Chapter 2 introduces the background and related work of this project. It starts by explaining the basics of vehicular networks and wireless network technologies that are, or can possibly be used to connect vehicles to vehicular networks. Afterwards it explains what the cloud is, the most known providers, their service models and how they can be used on a vehicular network setting. Next it talks about shared memory abstractions and how to implement them on a cloud and obtain snapshots. Finally it talks about related work done in the area of data sharing for autonomous vehicles.
- Chapter 3 describes the idealized design of the vehicular information system, the requirements that we make for it and highlights the milestones to attain in order to implement it.
- Chapter 4 describes the details of three AWS services, analyses its associated prices and execution latency of simple single and multi-threaded clients on requests such as *read*, *write* and *scan*. It ends by comparing the average price of each service under a defined scenario and their overall qualities.
- Chapter 5 is dedicated to defining and testing two client implementations that emulate a shared memory on the S3 service. The first client provides atomic snapshots at the cost of latency and the second one sacrifices the snapshot consistency in order to have lower latency. This chapter also emphasizes the need of a garbage collector on these clients to maintain performance over time and the need to increase the previously default number of threads in the thread pool of the multi-threaded variants to decrease latency. Besides the client designs, this chapter also covers their implementation, testing and their viability analysis.
- Chapter 6 focuses on taking the most appropriate client implementation made on the previous chapter and using it to create the two specific clients for the idealized

vehicular information system. It also includes tests and their respective analysis to understand the viability of this implementation.

- Chapter 7 concludes the thesis and refers to some future work to further improve the performance and increase the service's functionalities.

Chapter 2

Background and Related Work

This chapter gives a brief summary of the technologies relevant for the development of this dissertation. It gives a brief definition of vehicular network and describes some of the most known wireless networks that can be applied in a vehicular network setting. It also defines what the cloud is, its possible applications and its respective advantages and some related and relevant work in other fields which are relevant for the data management service implementation done in this project.

2.1 Vehicular Networks

As the name implies, vehicular networks [9] [10] are networks made specifically for vehicles, taking advantage of wireless technologies and ad-hoc networks [11]. The peers that form these networks can be vehicles, Road Side Units (RSU) and pedestrians that carry a communication device. The technical name for these peers are Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructure (V2I) and Vehicle-to-Pedestrian (V2P). Although there has been a huge improvement in these kinds of networks throughout the years, the main challenges still remain as latency and throughput [12]. The objectives that vehicular networks aim to achieve are supporting distributed applications for increased road safety and traffic efficiency.

These networks are comprised by several layers and mechanisms [13]: the physical layer cooperation, MAC (Medium Access Control) protocols, routing mechanisms, forwarding mechanisms, link scheduling, performance analysis, power/resource allocation, cooperative group communication and secure cooperative communication. The following paragraphs will focus on the most relevant of them.

The physical layer focuses on using spatial diversity to improve the reliability in the communications by transmitting messages through different communication channels (different frequencies). The most common systems that use this technique are called MIMO (Multiple Input Multiple Output) Systems and are able to have a bigger throughput compared to systems that use a single channel connection. However, for better usage

of these channels there are existing algorithms that improve communications and energy-saving.

The MAC protocols take care of how the nodes connect and communicate with each other. The current protocols can be subdivided in three categories [13]: contention-based, contention-free and hybrid. The contention-free protocols use a scheduler to regulate communication by giving each participant a variable-sized time slot (depending on the data size) in which it can use the communication channel. The problem with these type of protocols is when the channel is filled with nodes that want to communicate a huge number of data; that will cause a bottleneck on the channel. Contention-based protocols do not have a scheduler, so the node has to contend with its neighbouring nodes in order to coordinate themselves. This avoids the problem associated with the scheduler but it creates a problem with packet collisions by concurrent transmissions. Hybrid protocols combine the advantages of the contention-free and contention-based ones and attenuate the problems associated with them. They reduce packet collisions between non-neighbour nodes and are still able to achieve high channel usage under high contention conditions. All the problems associated with the protocols described above can be attenuated, but not totally solved, by more complex protocols [13].

The Routing Protocols in Cooperative Vehicular Networks (CVNs) have the requirement of needing the best paths with the existing relays in the network. In a perfect network, the nodes would never disconnect unintentionally but there is no such thing in the real world, so these protocols have to verify, on every hop, if the relays are still available to use. With this requirement in mind, the routing protocols are the ones that take care of verifying the network for dead/misbehaving relays. For example, there is a basic protocol called two phased-based generous cooperative (GEC) routing [13] that uses a watchdog model to keep in check false alarms and increase detection probability. The GEC protocol has two phases: route discovery and route maintenance. The first phase does the neighbor discovery, learning the relay metric and cooperative relay selection. The second phase just keeps the relays on check and if a node receives a route error message, it initiates a route recovery process. If the link to a node fails, then the route is erased from the existing routing table. This solution removes the uncooperative vehicles from the route and reduces the end-to-end delay.

The main focus on the Forwarding Mechanisms Protocols [13] is to find an alternate node on each hop needed to transmit a package, this way there is redundancy in the network and the ability to assign new transmission routes in case one is being used. There is a proposed solution that uses the idea of master/worker topology [13] . The master node selects the neighbor nodes that will be its workers. The method uses the direction, stability and closeness to the master as selection parameters to choose which neighbor nodes will be its workers. To send a message, it gets encoded by each node it passes using linear network coding [14] with fixed coding vectors; linear network coding refers to a scheme

for improving the throughput, efficiency and scalability of a two-way communication by encoding a data stream between two points.

The Cooperative Link Scheduling [13] is the process of selecting secondary links between nodes in order to transmit information without interfering with the primary communication links. This way it is also possible to have redundancy in the network. Algorithm-wise there is a simple implementation where there is a leader election which chooses the main node of each vehicle and then assigns the other adjacent ones to a secondary link to be used.

2.1.1 Wi-Fi

Wi-fi, also known as IEEE standard 802.11 [15], it is a family of ever improving protocols of wireless local area networks (WLAN) introduced in 1997. They focus on standardizing the Medium Access Control (MAC) and Physical (PHY) layers specifications on WLAN. These kinds of networks prioritize mobility over transmission speed while maintaining all the wireless devices connected with the wired ones and to the internet. There are many variants of this standard where security, speed and stability are some of the key improvements/changes. The current variant being widely used on home routers is the 802.11ac, using the 5GHz frequency band, is also known as Wi-Fi 5, although in 2019 the 802.11ax (Wi-Fi 6) was adopted as the next protocol to use, as it increases transmission speed while maintaining backwards compatibility with the predecessor protocols.
For vehicular networks there is the 802.11p variant [16], which changes some of the MAC and PHY standards to better fit the nature of those networks. Those changes include better communication channels between fast-moving vehicles and between vehicles and RSUs. This standard is used in various vehicular ad-hoc networks (VANET) such as DSRC (described below).

2.1.2 UMTS

Universal Mobile Telecommunications System (UMTS) [17] is a third generation standard for mobile cellular systems based on its predecessor Global System For Mobile Communications (GSM), a 2G technology. Although it is an outdated technology, it was a big improvement compared to its predecessor as it increased the transmission speed from 14.4 kbps to 2 Mbps. Its frequency band is situated on the 2600 MHz mark. Like the newer generations, it is based on subscriber identity module (SIM) cards for authentication into the network. This technology is referenced here mainly for further comparison with the newer ones and to better understand the increase in quality and performance in recent years.

2.1.3 DSRC

Dedicated Short-Range Communications (DSRC) [18] [8] is a VANET technology that allows vehicles to communicate in a V2V or V2I manner. Especially dominant in the US, it works based on short to medium ranged connections between nodes and boosts high speed, secure and anonymous data exchange in a one or two way communications. This network requires the connecting vehicles to have computational power as it makes them broadcast some of its data with the surrounding vehicles (e.g. speed and location) and process it. They can be used to detect collisions, warn about incoming vehicles or simply exchange traffic data. Although the first use of this network was for road and driver support, the autonomous vehicle industry also saw the potential of this network to be used on their vehicle networks. Being a wireless network, DSRC typically uses 75MHz in the 5.9 GHz band in the spectrum and is able to keep a communication line between vehicles even at high speeds and in a one kilometer range paired with low latency responses and strong connection, even in adverse conditions.

2.1.4 LTE

Long Term Evolution, also known as LTE [7] was the successor to 3G networks while trying to catch up with the 4G requirements. Although it does not reach those requirements, it still is a significant improvement compared to 3G networks. There is a misconception between LTE being a 4G network mainly for marketing reasons. It is a wireless broadband technology standard for mobile devices later adopted for vehicular networks due to its high data rate, large coverage area, huge penetration rate and low-latency. It uses the 700-2690 MHz frequency in the spectrum to exchange messages and a node infrastructure (V2I) organized in the same way as a cellular network to transmit information which allows a wide coverage area while maintaining a high speed connection. Besides being low latency, with a round trip time varying from 10 to 100 ms, it also has a high downlink and uplink capacity which can go up to 300 and 75 Mbps. Like any other technology, LTE has disadvantages since communications must cross infrastructure nodes even in a simple V2V data exchange; this might cause, in dense traffic areas, infrastructure overload and affect message latency.

2.1.5 5G

Also known as the fifth generation of wireless network technology for cellular networks [19], it has a frequency spectrum that can be divided in three categories: millimeter waves, mid-band or low-band waves. The millimeter wave is the fastest of the three, being able of speeds of 2 Gb/s and has a frequency value around the 30 GHz mark. The mid-band category is the most used of the three with speeds that can reach up to 10 Gb/s, using frequencies around the 2.6 GHz mark. The last one, the low-band waves offers a connection

Wireless Technologies	Frequency band(s)	Max bit rate	Range
Wi-Fi (802.11ac)	5 GHz	1.73 Gbps	100 m
Wi-fi (802.11p)	5.9 GHz	27 Mbps	1 Km
UMTS	2600 MHz	2 Mbps	10 Km
LTE	700-2690 MHz	300 Mbps	30 Km
DSRC	5.9 GHz (Europe)	27 Mbps	1 Km
5G	1.8, 2.6, 30 GHz	10 Gbps	500 m

Table 2.1: Features of the wireless communication technologies mentioned in this chapter.

similar to the 4G counterpart. This technology divides the coverage area in small geographical areas and calls them cells. All the devices connected in the cell communicate via radio waves to a local receiver/transmitter that are connected to the telephone network and internet, being able to create connections between devices from other cells. Since the first rumours about 5G networks there has been talks about using the potential of this technology to create a vehicular network [20]. Its capabilities for fast speed, structureless architecture, connection power, low latency and many others shows a promising future to mission-critical communications and internet of things. Although this is a fairly new technology, it is still expected to have a huge potential to be implemented as a vehicular communication network.

2.1.6 Comparing Communication Technologies

To summarize the technologies mentioned before, Table 2.1 was made with some of their characteristics. The similarity in the characteristics between 802.11p and DSRC technologies are due to the fact that DSRC is based on the 802.11p protocol; inheriting its features from it. Also note that the 500 meter range of the 5G is on the millimeter wave situated on the 30 GHz frequency band. If the frequency band is the same as 4G (600-700 MHz) it is able to achieve communication distances of 15 Km.

Besides using a single wireless communication technology on vehicular networks, it is possible to create a hybrid solution [8] by using both a cellular network and DSRC. It adds the advantages of the cellular network on top of the DSRC network. There are three benefits worth mentioning: the capability to create a backup connection to exchange vehicular data in case the V2V connections are broken, having an access network to the internet and using it as control message dissemination in case the DSRC network is fragmented. This hybrid network, although still being experimental, might improve the quality of vehicular networks.

2.2 Cloud Computing

Cloud computing [21] [22] is a technology that provides computer resources like computing power, databases and storage over the network. It is a distributed system where lots of machines work as one to offer their resources to the end-user. Cloud services can be provided according to three different service models:

- Infrastructure as a Service (IaaS) is the model that offers computing resources like storage or data processing services.
- Platform as a Service (PaaS) is an intermediate model with offers pointed towards software developers which can write their applications on a particular platform without the need to worry about having an hardware infrastructure. They can upload the code they created to the platform and it manages the up or downscaling of the application, depending on the usage.
- Software as a Service (SaaS) is the model that has the provider's software applications and tools and allows the user to use them in a pay-per-use way. This is the most used model by the end-users.

There are several cloud service providers and the most known public providers are Amazon, Microsoft and Google. They provide various services like analytic tools, machine learning services and databases. To keep themselves competitive in the market, they all have their own implementation of popular services; taking NoSQL databases [23] as an example, Amazon Web Services (AWS) has DynamoDB [24], Microsoft Azure has Cosmos [25] and Google Cloud has BigTable [26]. Thanks to these similarities it is possible to use multiple providers to the same end and that is the basic principle of a multi-cloud service [27]. Those providers offer their services on a pay-as-you-use basis.

There are also private clouds and hybrid clouds [21]. Private clouds are clouds with their data centers being managed by the company or organization that owns it and where that company has full control over it. One of its advantages is that the information stored inside that cloud never leaves the organization's machines, which makes more difficult to be accessed by hackers. The hybrid cloud merges both public and private clouds and allows an organization to use either of them to run their applications depending on the requirements. This solution mixes the advantages of both clouds, where the critical data and applications stay inside of the organization's data centers and still be able to run some less critical applications in a scalable public cloud.

Besides these types of clouds there is also the multi-cloud or cloud-of-clouds model [27], which uses more than one public cloud provider to work as storage or service. Not to be confused with the hybrid cloud solution, which uses both private and public clouds. There are several benefits associated with this solution; comparing to a single-cloud solution there is redundancy by having the files replicated on both clouds, lowering the

chance of not having them available at a certain time, scalability by using a software or being automatically managed by the provider to scale the services to be able to answer to the increasing or diminishing number of clients, security by partitioning the files between clouds and pricing because most of the multi-cloud providers are able to choose which cloud to use depending on the client's needs and offer the cheapest solution. The biggest downside to it is that it might be needed to create an interface to setup and use all those clouds as one if there are no interfaces available for that mix of clouds wanted.

2.2.1 Cloud and Vehicular Networks

By having this immense array of tools and features, the cloud can be a versatile technology to implement services and create networks. *Yan Zhang et al.* [28] proposed an architecture consisting of three interconnected cloud infrastructures which could be used to share resources between its participating peers. From the most resource constrained to the least there are: the vehicular cloud, roadside cloud and central cloud. This architecture was proposed because vehicles have low-cost hardware systems with limited computational and storage capabilities on pair with increasingly powerful applications; this is a problem that can be solved by creating a network for resource-sharing using all the available infrastructures for it. Starting with the vehicular cloud, which is composed by vehicles, having the most limited resources but high mobility; they are connected through a VANET which forms local clouds and share information between each other. The high mobility factor is the most challenging feature in this cloud which can quickly change the network composition; for that it proposes two customization strategies, one through virtualization of their resources and their distribution through a scheduler and the other one is by applying to use the other available vehicles computational resources, such as processing power. The roadside cloud is divided in two parts: dedicated local servers and roadside units. The servers provide virtual resources and act as a local cloud and the RSUs work as a radio beacon for the vehicles to connect to the local cloud. And finally the third infrastructure, the central cloud. This is the one with more computational power and resources in the architecture and can be created on top of dedicated servers for vehicular networks or normal servers in the internet through public providers. This architecture can support various applications such as real-time navigation, video surveillance and checking real time monitoring of traffic conditions.

2.3 Shared Memory Abstractions

By definition a shared memory [29] is a type of memory that can be accessed by multiple programs/clients simultaneously. Each chunk of memory is a shared register which can be accessed by single/multiple writer or reader. The shared memory is an abstract concept that can be implemented either in a single computer for data sharing between processes or

in a distributed system environment [30] that shares its storage information to connected clients.

2.3.1 Emulating MWMR registers in the cloud

As said in the previous section, it is possible to create a shared memory on top of any interconnected system if it has any shared storage space. It can go from a simple computer sharing RAM registers or disk storage with multiple processes to a cloud storage system with multiple computers reading and writing in that memory. In *Oliveira et al.* [1], the authors were able to emulate a shared memory with multi-writer, multi-reader (MWMR) registers on a multi-cloud Byzantine fault tolerant [31] setting while guaranteeing integrity, security and authenticity of the data. This multi-cloud solution used is based on public cloud service providers such as Amazon [32], Microsoft [33] and Google [34] and takes advantage of their Key-Value Store (KVS) services such as Amazon's S3 service [35] to emulate a shared memory. Although it is possible to simulate a shared memory by using only one instance of a service, having multiple instances ensures availability of the stored data despite failures in the cloud[31]. To access those KVSs and data, the authors created three different client algorithms. However, we will focus only on the first, as it will be the one used on this dissertation. The first algorithm relies on full-replication in the cloud storages and the other two rely on object partitioning and the use of erasure-codes [36] to detect faults. The registers use at least four cloud service providers because that allows to detect and tolerate Byzantine faults, maintaining availability if one of the services goes down and keep the files information secure. Figure 2.1 shows the algorithm used to implement the Two-Step Full Replication [1] *write* and *read*.

The *write* function works by invoking a *listQuorum* function, which returns a quorum of base objects. Afterwards, it checks for the most recent metadata information with a valid signature of the desired register and uses that information to sign and identify (with a datakey) the new object that is about to be written. Finally it invokes the *writeQuorum* function which creates the new entry for the register in the quorum. The *read* function is to read one register and works by listing a quorum of base objects. Then, it enters in a loop until it reads a valid value from the list, which is the most recent register object in the quorum. That value is the one returned to the reader. Note that this implementation does not delete older register values.

2.3.2 Shared Memory and Snapshots

A shared memory is usually comprised by multiple registers. Sometimes, there is the need to be able to get a view of the content of all the shared memory registers. That view is called a snapshot and it can be better understood as a representation of the shared memory at a certain moment in time. In other words, a snapshot shows the values being held by

```
1  Procedure FR-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      new_key ← ⟨max.ts + 1, c, H(value)⟩;
5      data_key ← new_key + sign(new_key, K_r);
6      v[0..n − 1] ← value;
7      writeQuorum(data_key, v);
8  Procedure FR-read() begin
9      L ← listQuorum();
10     repeat
11         data_key ← maxValidVersion(L);
12         d[0..n − 1] ←⊥;
13         concurrently for 0 ≤ i ≤ n − 1 do
14             value_i ← get(data_key)_i;
15             if H(value_i) = data_key.hash then
16                 d[i] ← value_i;
17             else
18                 d[i] ← ERROR;
19         wait until (∃i : d[i] ≠⊥ ∧d[i] ≠ ERROR) ∨ (|{i : d[i] ≠⊥}| ≥ q);
20         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {data_key};
21     until ∃i : d[i] ≠⊥ ∧d[i] ≠ ERROR;
22     return d[i];
```

Figure 2.1: Two-Step Full Replication *write* and *read* functions algorithm [1]

all registers from the shared memory at a certain moment.

Due to the independent and ever-changing nature of the registers, creating a snapshot might prove difficult, especially if a wait-free algorithm [37] is to be used so it does not hinder the normal functioning of the other registers. A wait-free algorithm guarantees that every call is finished in a finite number of steps, preventing a block in simultaneous calls/processes. That is what is proposed in *Afek et al.* [2]. One of the proposed wait-free pseudo-code algorithm for single-writer, multi-reader (SWMR) registers can be seen in Figure 2.2. That snippet has two functions: *update* and *scan*. Starting with the simple one, the update, which simply changes the information stored in a register. Each register holds the snapshot of the shared memory at the moment of the update, the sequence number of the update and the arbitrary data given through the parameter *data*. All the parameters, besides the arbitrary data parameter, are stored the register to be able to maintain the atomic guarantees. The *scan* function basic idea is to read all registers in the shared memory and return the relevant data of all of those registers (snapshot). For the scans to be atomic and wait-free, the algorithm needs to be a bit more complex. It is based on two fundamental ideas:

- It is possible to obtain a snapshot by making two sequential reads of the entire memory. However, it is only declared a successful snapshot if every register sequence number is the same in the reads. That means the shared memory was not changed during those two reads and thus constitute a valid atomic snapshot.

- If a *scan* sees a register being updated twice, it means that the register completed

```
procedure scan_i
    begin
      0:  for j = 1 to n do moved[j] := 0 od;
      1:  while true do
      2:      a[1..n] := collect;                          /* (data, seq, view) triples. */
      3:      b[1..n] := collect;                          /* (data, seq, view) triples. */
      4:      if (∀j ∈ {1, ..., n}) (a[j].seq = b[j].seq) then
      5:        return (b[1].data, ..., b[n].data);              /* Nobody moved. */
      6:      else for j = 1 to n do
      7:           if a[j].seq ≠ b[j].seq then                   /* P_j moved. */
      8:             if moved[j] = 1 then                /* P_j moved once before! */
      9:               return (b[j].view);
      10:            else moved[j] := moved[j] + 1 ;
             od;
          od;
    end scan_i;

procedure update_i (data)
    begin
      1: s[1..n] := scan_i;                                    /* Embedded scan. */
      2: r_i := (data, r_i.seq+1, s[1..n]) ;
    end update_i;
```

Figure 2.2: The unbounded single-writer algorithm used to acquire atomic snapshots [2].

one update within the interval of the scan. If this applies, it is possible to return the snapshot contained in the last updated register. This is the reason why each register holds a snapshot of the scan done before updating, as it can be used as a snapshot in case this situation does happen.

The advantage of an atomic snapshot is that it guarantees that in a certain moment in time, the shared memory had the presented values. In other words, the atomic snapshot is linearizable [38]. Its formal definition says that linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre and post-conditions.

2.4 Related Work: Cloud-based Membership System For Vehicular Cooperation

The challenge of creating a cloud-based vehicular information system to connect autonomous vehicles has already been tackled before by *Casimiro et al.* [39]. In this work, the authors defined a membership service that provides each autonomous vehicle a view of the relevant vehicles nearby, with which they need to coordinate to safely perform maneuvers.

An implementation of the membership service proposed in [39] was described in [40],

using Zookeeper [41] for data storage and coordination. Zookeeper provides synchronization, fault tolerance and scalability for a hierarchical main memory database. This work takes advantage of those features to implement the data structures needed to maintain a membership service. On a high-level overview, they use the hierarchical tree structure to organize their service into three categories: agents, membership and segments. Each category will hold ”agents” as their leaf nodes. Each agent is a remote connection between a vehicle and the service. The agent connections are made under Zookeeper's ephemeral nodes. That means that as long as the client connection is maintained, the node will stay in the tree. If the client disconnects from the service, the node will be removed. This way, the service keeps itself updated only with the active vehicles.

The author did some tests of his implementation of the membership system running on a machine based in the cloud and concluded that the connection latency was one of the main factors that affected his solution's viability in a real world system.

2.5 Conclusion

This chapter started by introducing the basics of vehicular networks, their uses and challenges in various fields and some potential wireless networks that are or can be used to implement one. The final section provides a table with a comparison between those wireless networks as well as a reference for a potential hybrid solution, mixing the advantages of both short range communication protocols such as DSRC and cellular networks to make a more resilient network.

Afterwards we presented a brief introduction to cloud computing by talking about the service models that public providers offer, as well as some service examples from some known public cloud providers. This was followed by a description of the different types of clouds such as public and private clouds and what defines them. Finally, it references a structure proposal [28] in which the idea of using clouds on various levels in a vehicular setting to create a fully functioning vehicular network.

To end this chapter's background, the concept of shared memory was given to familiarize the reader to the topic and introduce to the work done by *Oliveira et al.* [1], where it is offered three different algorithms to emulate, on top of multiple KVSs services from multiple cloud providers, a shared memory with a wait-free MWMR registers that can be accessed by external clients, while guaranteeing Byzantine fault tolerance, availability and security. In order to obtain a true view of the system at a given time it is needed to acquire an atomic snapshot from the shared memory, and that is what is explored afterwards by analyzing the most basic wait-free atomic snapshot algorithm from SWMR registers.

Finally, it was referenced the related work of *Casimiro et al.* [39] where he proposes a cloud-based membership service for vehicular clients and an implementation from *Correia* [40] using Zookeeper as a database. Their implementation and conclusions will be

taken into account for the realization of this thesis.

Chapter 3

Vehicular Information System Architecture

In this chapter we will be providing an architecture solution to the problem we aim to solve on this dissertation. To do that, we will begin by recalling the problem, explaining the scenario it is applicable and the requirements needed to be met in order to successfully implement the said architecture.

3.1 Problem Definition

In order to have a cloud-based vehicular information system providing timely data and serving a high number of autonomous vehicles, it is required to have a certain degree of coordination between clients and the service. If there is little to no coordination, the communication latency will be small, but the data coherence will be jeopardized and will affect the decision making of the participating clients, facilitating the occurrence of mistakes. If the system is too strict, the opposite will happen, the communication latency will rise but the data will rarely be incoherent.

Let us imagine a scenario where a real-time vehicular system is used to exchange relevant data between autonomous vehicles to support them in real-time decision-making. We can not expect to be able to sacrifice access latency to have perfect data consistency or vice-versa. To solve this problem we will have to find a solution where it allows a huge number of clients to exchange data in due time, while maintaining some data coherence in the reads and storage. On this scenario, if the worst comes to happen and the client is not able to obtain the required data from the service, it will work independently, using its own data gathered through its sensors.

To maintain a timely exchange of data between vehicles it is required to separate responsibilities on the cloud-based vehicular information system. For that we will need to have at least two different clients in the vehicular system working together by sharing data through the cloud storage. One client will be the vehicle which shares his data on the cloud storage and the other one is a client which uses the data provided by the vehicles to

produce and return useful data for the vehicles. Let us call the first type of client ”vehicle client” and the latter ”calculator client”.

3.2 Requirements

For the proper functioning of the cloud-based vehicular information system, it is expected from the clients the following requirement:

- The latency spent on communication can not be superior to two seconds, otherwise the client will declare that the service is unavailable.

Due to the need of the clients to obtain non-stale data in a short period of time, a latency upper bound of two seconds was defined.

This value was chosen because the maneuvers this system aims to assist are done in four to six seconds on average [42] (e.g, lane change). In order to do so, it is required of the system to provide data that remains valid for a longer period than the sum of the average time to read the data and execute the maneuver [39]. Following this, we will have to retrieve data that must be valid for at least 6 seconds, assuming the two second communication latency plus four second to execute the required maneuver. Although being an acceptable total time, it is only acceptable in scenarios with a limited dynamism such as highways. In urban environments, the surroundings change faster, which will require a decrease in total time to provide the data in a timely fashion. To begin with, we shall start with the biggest latency accepted which is two seconds.

If any client is taking more than two seconds to obtain the data needed from the service, it shall declare that the service is unavailable and work with its own data gathered through its sensors, as it is preferable to act independently than using stale data.

3.3 Proposed Architecture

This section will focus on describing the proposed architecture of the vehicular information system. Our main contribution on that system is the client. We will explain about its existing components, their respective functions and the dataflow present on the system. The presented architecture aims to create a solution to the previously referenced problem where coordination, data consistency and latency must be balanced in order to achieve a viable solution for a vehicular information system.

Figure 3.1 presents the proposed architecture. There are six components that will be described, from the most inclusive to the least:

1. Cloud-based Vehicular Information System - It is the whole system. When talking about it, we refer to all the different clients and the cloud storage/service.

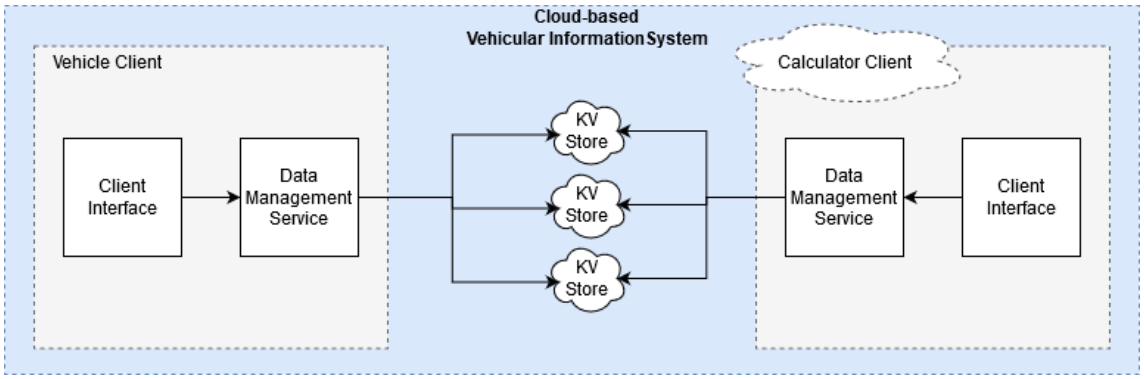

Figure 3.1: Proposed architecture of a cloud-based vehicular information system.

2. Vehicle Client - Represents the vehicle connection to the Cloud Storage and holds the Client Interface and Data Management Service as sub-components.

3. Calculator Client - Represents the connection of the auxiliary client to the Cloud Storage it contains the same sub-components as the previous client.

4. Client Interface - Interface layer which the client can use to interact with the Data Management Service component. It can invoke functions like *Read*, *Write* and *Scan*.

5. Data Management Service - The Client layer that has all the rules and capabilities to correctly interact with all the KV Store components as a single shared memory representation.

6. KV Store - Cloud-based component that stores data. The vehicular information system can have as many as necessary.

This is the architecture that we propose and implement on this dissertation. By using simple KV Stores and leaving the complexities to the clients, we aim to remove most of the responsibilities from the servers, allowing them to use their resources to harbour more vehicles. We also use a shared memory abstraction in the Data Management Service component, which allows the client to access the KV Stores as a shared memory. This shared memory implementation works with a quorum of KV stores and grants availability to the cloud-based vehicular information system. The Client Interface component will allow the client to write on its register, read any register and scan (snapshot) the shared memory.

3.4 Final Remarks

This chapter described how the architecture was idealized by first explaining the existing problem on this type of systems and by offering a scenario where it can be applied.

Afterwards it was defined a single requirement before presenting and explaining the architecture that is going to be followed to create a functional client for a possible cloud-based vehicular information system application.

Chapter 4

Client Latency on Cloud Services

This chapter is dedicated to the tests done in the AWS services to obtain the latency of the basic function calls like *read*, *write* and *scan*. It is organized in four sections, each one for a service and their respective tests and the final one to sum up the conclusions taken from the tests, compare results between services and describe which service will be used for the vehicular information system cloud storage. The focal points are pricing and latency on read/write operations. The pricing is important because the services are based on a pay-as-you-go policy and one of the goals is to create a cloud based information system as less expensive as possible, without compromising the quality of service and ease of implementation. If not, it will be used to give a benchmark on service prices. The low latency are also important because the system will need that to be able to give timely responses to its clients. All the prices on this chapter were last checked and updated on August 2020.

The services used were the following:

- The S3 service [35] provides object storage that is scalable and available secure with a high degree of performance.
- The EC2 service [43] allows you to run virtual machines in the cloud and use them as you see fit. In this case three instances of a Linux OS running a Zookeeper [41] cluster with three nodes were used.
- The DynamoDB service [24] is a key-value database that self-scales and provides quick access to those values.

A free-tier account was used to access all the services used in this chapter. The Amazon services were the only ones tested because it was decided beforehand that this was the provider to be used for this project's cloud needs. This choice was done mostly due to their reputation, quality of service, ease of use, documentation availability and richness of the interfaces.

The tests were done with local clients connected to the respective service through a wireless connection to the internet, and will invoke the respective basic functions that

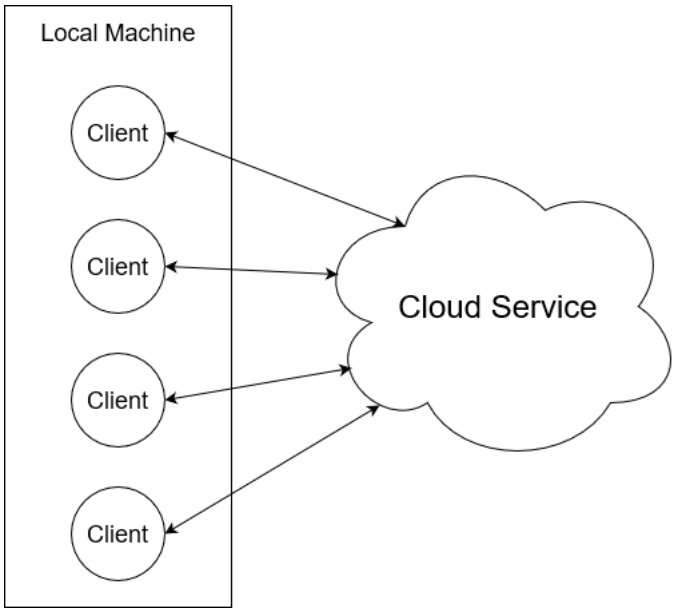

Figure 4.1: Diagram of connection between clients and the cloud services used for testing.

allows to *read*, *write* and *scan* data. Figure 4.1 shows the diagram of how the connection between clients and cloud service will be made. All the necessary clients for the current test will be created in a single machine. All tests were done after doing some function calls on the tested service before starting to annotate values for the tests. This process is called warm-up and will make the tests have more stable latencies at the start, with the eventual first call higher latency. The connection between the client and cloud service will not be a bottleneck because the amount of data exchanged is small.

4.1 AWS S3

This section describes the tests done with S3 [44]. This service provides object storage in the cloud to the user by using buckets as storage points. Those buckets can be created by the user and shared to other trusted users. The bucket owner is the one who pays for the costs. The bucket can be compared to a ”storage unit” located in the cloud that keeps the files under defined rules.

The tests done create one file/object in the bucket per client and hold all the uploaded information on it. Each file has 100 bytes of data, more than enough space for vehicular clients to store their location and other necessary sensor information. Those objects will be stored in a bucket which could be compared to a ”disk” that stores the files in the cloud. Those objects can be atomically read, written, updated or deleted through by the S3 API. All of these tests were made in a computer physically connected to the internet. The bucket deployment choice was EU-London.

To check the costs of this service, the AWS S3 price table [45] was checked and

S3 Standard	Storage Pricing
First 50 TB/Month	0,024$ /GB
Next 450 TB/Month	0,023$ /GB
Over 500 TB/Month	0,022$ /GB

Table 4.1: Storage pricing of the S3 services on data.

analysed on which kind of conditions the user would be paying for using it, and will be later compared with the other services. For latency calculations, four test were done:

1. Get the latency of reading a single object to get a good metric to compare with the multiple read tests, where the closer the value from those tests to this one, the better. It will also be compared with the write latency value.

2. Get the latency of a single write to be able to have a reference value on how fast it is possible to upload an object on this service and to compare with the read latency.

3. Check the viability of using a single-threaded client to read all the objects in a bucket and have that value to compare with the fourth test.

4. Use a multi-threaded client, reading all the objects in parallel, and confirm if a multi-threaded solution fares better than the single-threaded one in this use case and if its preferable to use in any kind service that needs to read multiple independent objects (read order does not matter).

The latency of a function call is the elapsed time since its invocation until it returns.

4.1.1 Pricing

As one of the goals of the vehicular information system is to be as cheap as possible, a thorough inspection of the pricing tables [45] of this service is needed. The EU-London-based bucket pricing values are shown in Tables 4.1 to 4.3. As it can be seen there are three aspects that sum to the total cost: the storage space, requests and transfers. The first one (Table 4.1) divides the cost in three categories depending on the amount of TB of objects stored on the bucket. Assuming the vehicular information system is not expected to surpass 50 TB/month, so it is expected to pay around 0,024$/GB per bucket. For writing and reading data (Table 4.2), the values are straightforward as the owner will have to pay 0,0053$ and 0,00042$ per 1000 requests, respectively. By comparing these last two prices it can be concluded that this service is cheaper if the clients it serves are read-focused, as it is cheaper to invoke a read than a write. If the system needs to be more write-focused it can become more expensive to the user.

Table 4.3 shows the pricing per GB/month of data transferred from the S3 system to the clients. The first GB of data transferred is free but scales up to 0,09$ for the

S3 Standard	Request Pricing
PUT, COPY, POST, LIST requests (per 1000 requests)	0,0053$
GET, SELECT, and all other requests (per 1000 requests)	0,00042$

Table 4.2: Request pricing of the S3 services on data.

S3 Standard	Data Transfer Pricing
Up to 1 GB/Month	0,00$ per GB
Next 9999 TB/Month	0,09$ per GB
Next 40 TB/Month	0,085$ per GB
Next 100 TB/Month	0,07$ per GB
Greater than 150 TB/Month	0,05$ per GB

Table 4.3: Data transfer pricing out of Amazon S3 to internet.

next 9999 TB of data. As can be seen in the table, the price per GB decreases in each milestone reached. At this point it is not possible to predict the total amount of data being transferred in and out of a cloud-based vehicular information system, but we can expect the data transfer pricing to be one of the most costly parameters to be taken into account. The data transfer into Amazon S3 from the internet is free.

4.1.2 Read Latency

The goal of this test is to verify the average latency of a read call of a single object to get a good metric to analyse the multiple read tests that will be done afterwards. To measure the latency it was created an AWS client that calls a function that creates a local timestamp at the start and at the end of the read call. Knowing that it is a blocking call, by subtracting the timestamps it is possible to have the time it took to make the read call. The pseudo-code snippet can be checked on Algorithm 1.

This test consists of running the Algorithm 1. 1000 times and averaging the returned values. The results can be seen in Figure 4.2. The figure represents the latency (vertical axis), in milliseconds, each run took (horizontal axis). It also shows, in orange, the average latency from those runs. The occasional spikes were caused by the wireless connection of the machine running the tests and therefore can be ignored as they are not caused by delays on the service. The average latency read time of an object in the cloud is of 66,7 ms with a standard deviation of 23,48 ms and a 95-percentile of 87 ms. This average value will be used as a reference for comparison with the multiple read latency.

Algorithm 1: Method used for checking the time a function call took to execute.

```
Result: Total time to execute funcCall()
startTime = getCurrentTime();
funcCall();
return getCurrentTime() - startTime;
```

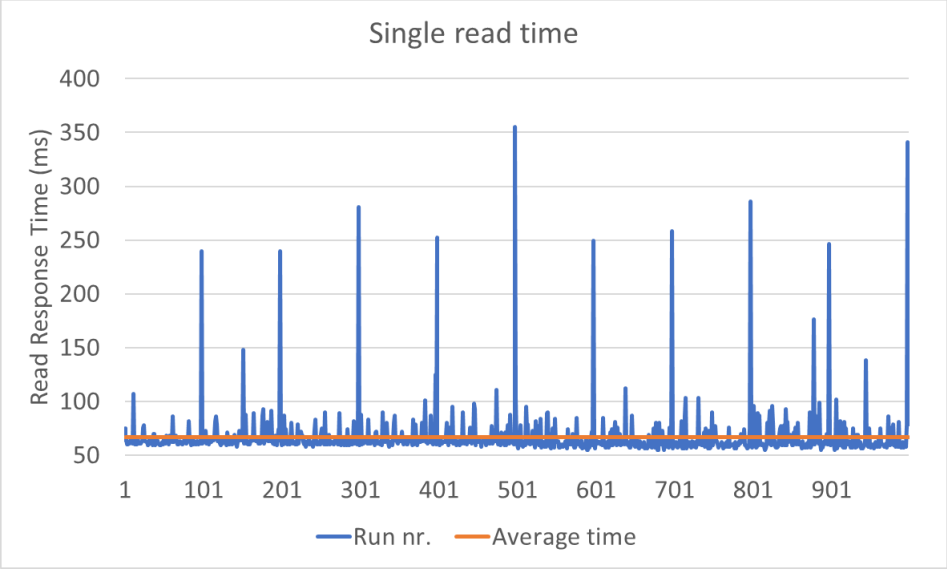

Figure 4.2: Single read time on AWS S3.

4.1.3 Write Latency

This test was made to evaluate the time it takes to write a single object to a bucket. As the data management service will require both read and write operations to be done quickly in order to be able to analyse the most recent data and write its processed information, it is preferable for those times to be similar.

Using the same methodology described before, it was possible to gather the data shown in Figure 4.3 which can be interpreted in the same way as the figure from the previous test. The time spikes were also caused by the same problem as this test was made in the same machine. The average write time is of 130 ms with a standard deviation of 41,21 ms and a 95-percentile of 170 ms. Although the average time difference between this test and the previous one is around double, the time scale is so small that 130 ms is still fast for sending a small object to the cloud.

4.1.4 Multiple Read Latency with a Single-threaded Client

This test's objective is to check the viability of using a single-threaded client to read K files on a fast-responding system such as a vehicular information system for autonomous vehicles. The latency it takes to read those files will be the determining factor on whether

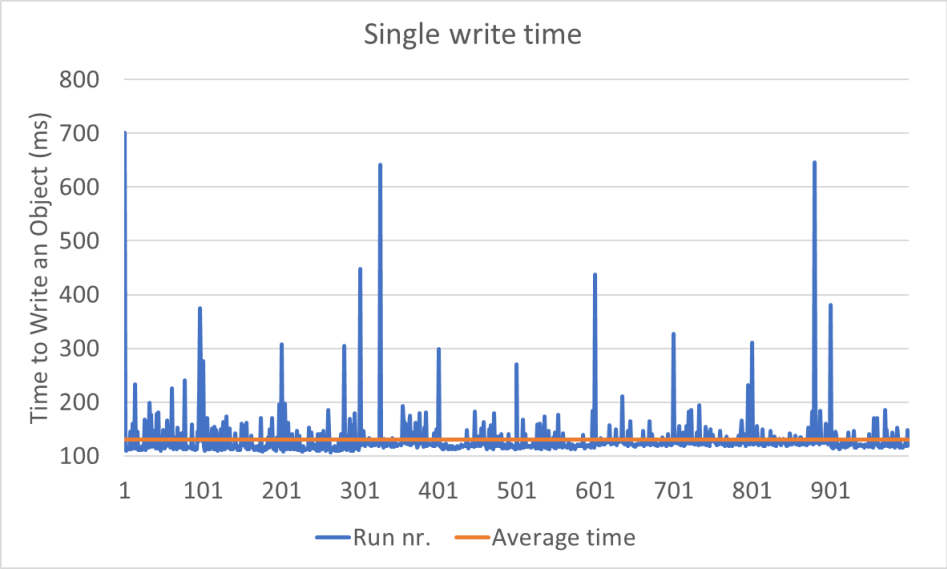

Figure 4.3: Single write time on AWS S3.

the clients will be single or multi-threaded. This choice will also influence the technological requirements of the vehicle clients as a multi-threaded solution requires better CPUs. The time will be measured by creating a client that uses the same algorithm of the previous tests, replacing *funcCall()* with a function that invokes multiple read calls, each on a different object, K times. The test consisted in reading K objects, ten times and averaging the time it took to finish those reads. The test was made with $K = 100$ objects and the results can be seen in Figure 4.4.

The results can be interpreted in the same way as the previous ones, where the horizontal axis shows how much time each of the ten runs took with the time scale on the vertical axis, in milliseconds. It also shows the average time line in orange. The average latency to read $K = 100$ objects is 7347,14 ms with a standard deviation of 1840,86 ms and a 95-percentile of 10752,9 ms.

This result was expected since the average result of a single read multiplied by K would give an approximate value of a single-threaded client reading those K objects. The value is not exact due to the possible variations in latency that each read can partake.

Having the fast-responding data management service in mind, having less than 100 reads done in a minute shows how inefficient and incapable a single-threaded client can be, as the system would require to read a higher number of objects in a smaller time period. From this test we can conclude that every single-threaded client, using a single read native function, in either service tested, will always get the approximate latency following the $K \times t$ equation, where K is the number of objects to read and t is the time it takes to execute a single read. Because of that the next AWS services will not be tested considering a single-threaded client.

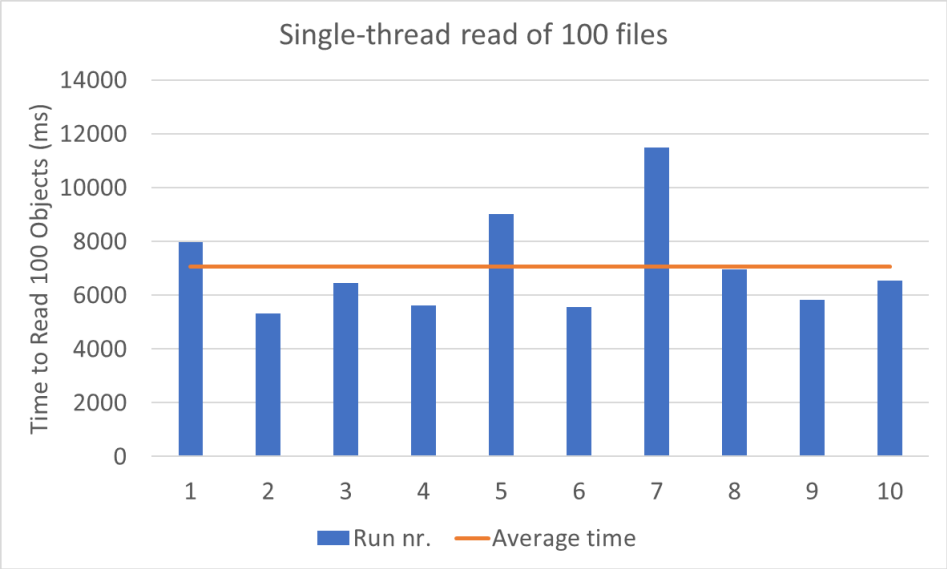

Figure 4.4: Single-threaded read runs on S3.

4.1.5 Multiple Read Latency with a Multi-threaded Client

This test's objectives are to obtain latency metrics to compare with the single-threaded client and to confirm that this type of client is a better implementation for a fast-responding system. This client now uses a multi-threaded function which reads all the objects existing in the bucket (calls are now done in parallel). It creates and uses a thread pool to submit all the objects to read. The thread pool is assigned with the same number of objects to be read by the client (K). This way it will be able to process all the requests that are not waiting for a remote response. The test is done using this new client but in the same manner and under the same parameters as the previous one, with a slight difference. Instead of testing only the read time of 100 objects, it will also be done a test for the read time of 1000 objects. This was done to obtain values for a closer representation of the number of objects a small to moderate cloud-based vehicular information system would have the client/data management service read. The results can be seen in Figures 4.5 and 4.6 and can be interpreted in the same way as the figure from the previous test. The read 100 objects test got an average time is 172,5 ms with a standard deviation of 88,59 ms and a 95-percentile of 324,74 ms and the read 1000 objects test got an average of 1300 ms with a standard deviation of 346,14 ms and a 95-percentile of 1872 ms. The first runs in the tests took more time than the other ones due to the fact that the warm-up previously done was not enough to prepare the service for such workload.

From those values collected it is possible to conclude that this type of client can read more objects than the single-threaded client in a smaller time window and shows that this type of client is better than its counterpart on reading speed. This also shows that the multi-threaded reader is better to implement on the data management service due to its read speeds.

Figure 4.5: Multi-threaded read on a system with 100 objects.

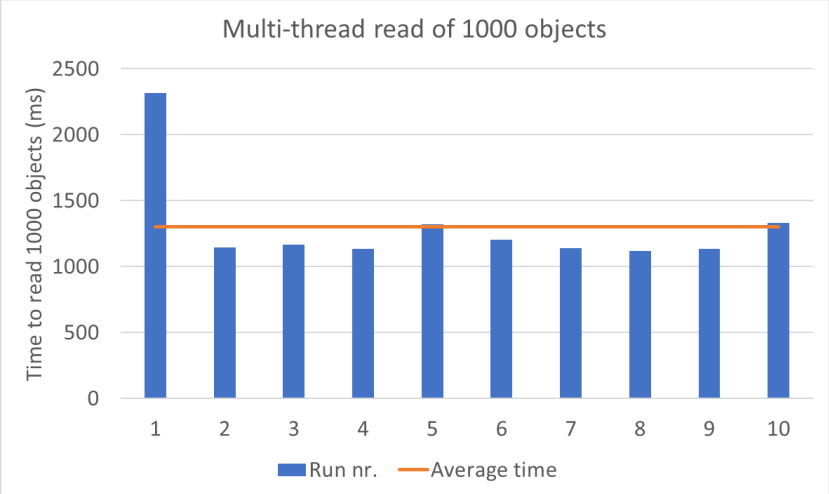

Figure 4.6: Multi-threaded read on a system with 1000 objects.

4.2 AWS EC2 Running a Zookeeper Cluster

This section is dedicated to the tests done with AWS EC2 running a Zookeeper cluster [43] [41] and their analysis in order to achieve a conclusion about its pricing and read/write latency compared to the other services. The objective of these tests is to obtain those metrics. Before making the tests it was needed to prepare two things: the server and the client service. The server consists of three connected virtual machines on the EC2 running a Zookeeper cluster (each instance ran a Zookeeper server). The instances are running an image of the Linux OS and are categorized as a t2.small general purpose instance with 1 vCPU (generally a CPU thread or core), 2GB of RAM and a small storage space. The cluster has a single master and two followers and is used to replicate the database to ensure availability. The client service is a multi-threaded program where it has a thread pool with clients accessing the database by creating an ephemeral child node in the database hierarchical tree structure and execute writes on its respective node and reads on all the existing nodes on the same level (which are another client nodes). Each client instance is capable of updating itself and read the information of the other child nodes. The child nodes are ephemeral and have a unique ID and location information.

To check the costs of this service, it was verified the EC2 price table and the Data transfer table and calculated the run cost for 24 hours as the instances are on a pay-per-hour basis. To check the latency on this service three tests were made. One for the single read, another for the single write and the last one for multiple reads. The first two tests are for obtaining a metric on the single action times for a better understanding and analysis of this service as a possible cloud storage for the vehicular information system and the last one is to check the time spent on *scan* of the system. That will give a good metric to understand how much time it will take on average to read a K node system.

As one might notice, this is a very specific service to test, especially when AWS offers other simpler, easier and more autonomous databases. The reason to test this is to obtain latency metrics using the same database design from the work of *Correia* [40] to have comparative metrics for an already tested system where he concluded that a better database could be found as an alternative, due to the inefficiency of Zookeeper to keep up with this specific task, which it was not primarily designed for.

4.2.1 Pricing

Like before, this section serves to point down the pricing [46] of this service; it uses a pay-per-hour policy where the price varies depending on the instance size and type. Each size has a certain amount of resources assigned such as CPU power, RAM and storage space. Besides the size there are various types of instances such as general purpose or compute optimized. As referenced in the introductory part of this section the instances used are categorized in the general purpose. For the tests described below it was chosen

EBS	Storage Pricing
General Purpose SSD (gp2) Volumes	0,116$ per GB-month of provisioned storage
Provisioned IOPS SSD (io1) Volumes	0,145$ per GB-month of provisioned storage AND 0,076$ per provisioned IOPS-month
Throughput Optimized HDD (st1) Volumes	0,053$ per GB-month of provisioned storage
Cold HDD (sc1) Volumes	0,029$ per GB-month of provisioned storage

Table 4.4: EBS pricing of storage options for Amazon EC2.

a t2.small size, costing 0,026$ per hour per instance. A system that runs three t2.small general purpose instances for 24 hours a day, will reach a total cost of 1,872$ per day. Bear in mind if the system is to have an increasing number of instances, with the possibility for each instance to be auto-scalable in order to keep up with the number of vehicle clients, it is expected that this price will increase.

Adding to those costs, there is also a data transfer pricing from EC2 to the internet. It works the exact same way as the data transfer pricing from the S3 service, maintaining the same costs with the same milestones. It can be analysed in Table 4.3.

Besides those costs, there is also the Elastic Block Store (EBS) pricing [47] associated with each instance. Table 4.4 shows the options one can choose from and their respective cost. EBS [48] provides block-level storage volumes for use with EC2 instances. There are four different options, each with its own characteristics. The one used for the tests was the General Purpose Volumes (gp2) which has 0,116$ per GB-month of provisioned storage.

4.2.2 Read Latency

The objective of this test is to obtain the average single read latency in a Zookeeper service to use as a comparison metric for the next test, the multi-threaded multiple read. To realize this test it was used the client service with one thread of the thread pool reading one existing ephemeral node in the server cluster. To obtain the time values it was used the Algorithm 1 and replaced *funcCall()* with the respective read call of the Zookeeper API. Just like in previous experiments, the read call was invoked 1000 times with each latency being registered and averaging the values in the end. The calls were made consecutively without any time break between them. The results can be seen in Figure 4.7. The average read time of a node is of 46,15 ms with a standard deviation of 4,49 ms and a 95-percentile of 51 ms. These values bring no surprise as Zookeeper is built as a read focused system [41], making the reads have small latency.

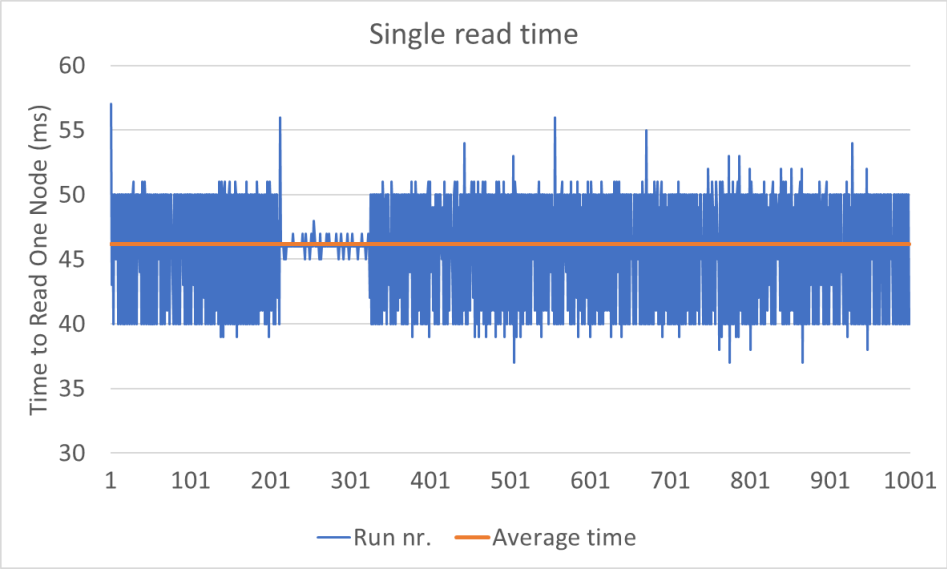

Figure 4.7: Single read time on EC2 w/ Zookeper.

4.2.3 Write Latency

This test's objective is to obtain the average single write latency in a Zookeeper server to use as a comparison metric with the read test values. Using the same approach used in the previous test, the service now proceeds to write 1000 nodes counting the time it took to write on each one and getting the average value. Figure 4.8 shows an average time of 52.01 ms with a standard deviation of 1,45 ms and a 95-percentile latency of 55 ms. As mentioned on the Zookeeper paper [41], the write times should be substantially higher than the reads, as their requests must be sent to the leader, and afterwards, go through an atomic broadcast (requiring extra processing) and, because their transactions must be logged to a non-volatile store before the servers send the acknowledgement back to the leader. This is not reflected on this test because the latency between the machines in the cluster to execute the broadcast is smaller than the latency between local client and the cloud.

Summing up, the values got from this test represent the time the client takes to start and finish a function call and that may not represent the true time to execute the said function on a distributed service such as the Zookeeper.

4.2.4 Multiple Read Latency with a Multi-threaded Client

The objective of this test is to get a metric on how much time the system takes to respond to a *scan*. To accomplish that, it was used a test setup similar to the multi-threaded read test done for the previous service, maintaining the same algorithm and number of runs. The differences being on running the test only on a 1000 node system and using the client service to create 1000 nodes and use one reader client thread to read all the nodes.

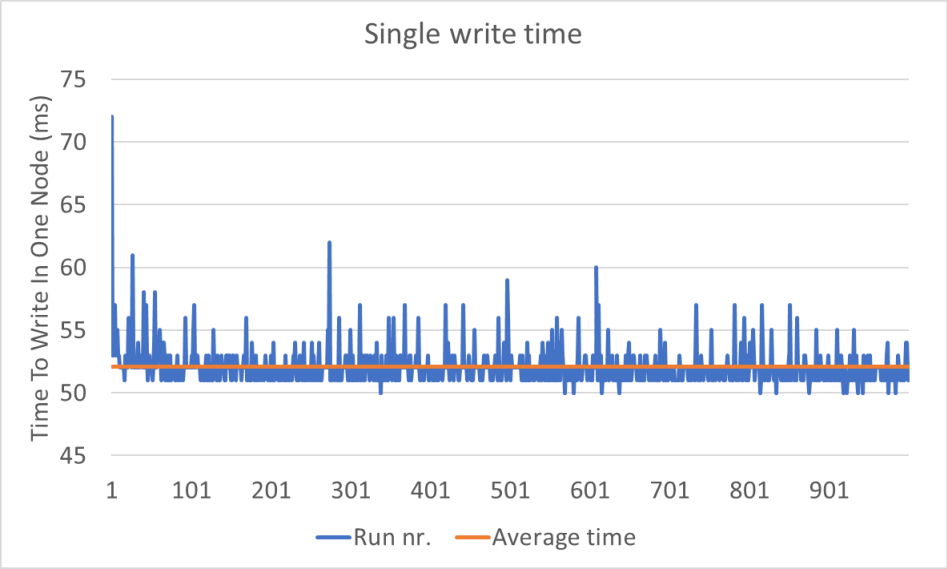

Figure 4.8: Single write time on EC2 w/ Zookeper.

The results can be seen in Figure 4.9 with the average read time of 242,2 ms and with a standard deviation of 102,92 ms and a 95-percentile latency of 425,55 ms.

4.3 Amazon DynamoDB

This section is dedicated to the DynamoDB (DDB) [24] service, a cloud-based database. This service provides the user with a simple interface where its only responsibility is to create the desired tables with the required keys and it is ready to use. This interface hides the automatic scalability and resilience features, making it easy to use, even for an inexperienced user. The pricing, the write time of one row and *scan* time of the table will be measured in order to obtain metrics for this service. To set up the tests it was created a table with two columns; one for the client ID and the other as a field for client information. It was decided to use only one column for the client information to be as similar to the tests done on the other services. Besides that, it was created a client service for writing rows and to scan the table. The first analysis, the pricing one, consists on checking the respective price table and analyse it. The remaining tests are done by measuring the latency of the respective functions being analysed. The results will be used as time metrics for this service. These tests were done with a free AWS account which brought some restrictions on the use of this service. These restrictions will be addressed below. The tests done are on a table with an on-demand capacity mode because it supports an unstable throughput of users and adapts to it to better than its counterpart (provision capacity mode). The selected mode has the pricing policy focused on the number of read and write operations.

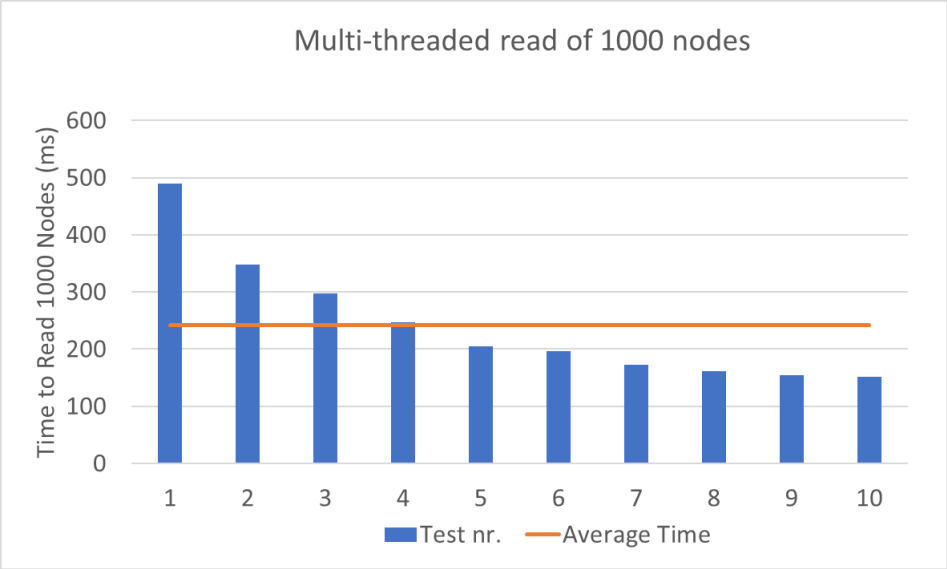

Figure 4.9: Multi-threaded read on a system with 1000 nodes.

	DynamoDB
Write request units	1,4846$ per million request units
Read request units	0,297$ per million request units
Data Storage	First 25 GB stored per month free/ 0,29715$ per GB-month thereafter

Table 4.5: DynamoDB price table.

4.3.1 Pricing

This section focuses on analysing the costs of using this service [49]. It uses a simple pay per number of requests policy associated to its on-demand mode. Similar to the S3 service, the pricing parameters are divided into requests and storage of information. Table 4.5 shows the pricing of this service. The write is the most expensive request, having to pay more than 1$ per million writes. Afterwards, comes the read requests with a price of 0,297$ per million requests and finally comes the data storage price, where the first 25 GB/month are stored for free and paying 0,297$ per GB/month on the next ones. Like the S3 service, this one favours the read over write operations, price-wise. This means if a user wants to use this service for a write-heavy workload, he should be prepared to pay more for it.

Like the previous services it also has costs associated to data transfer with the conditions and costs being exactly the same.

4.3.2 Write Latency

This test measures the time it takes to write a two column row in the table. To obtain that time it was used the same algorithm applied on the previous tests. Because DynamoDB

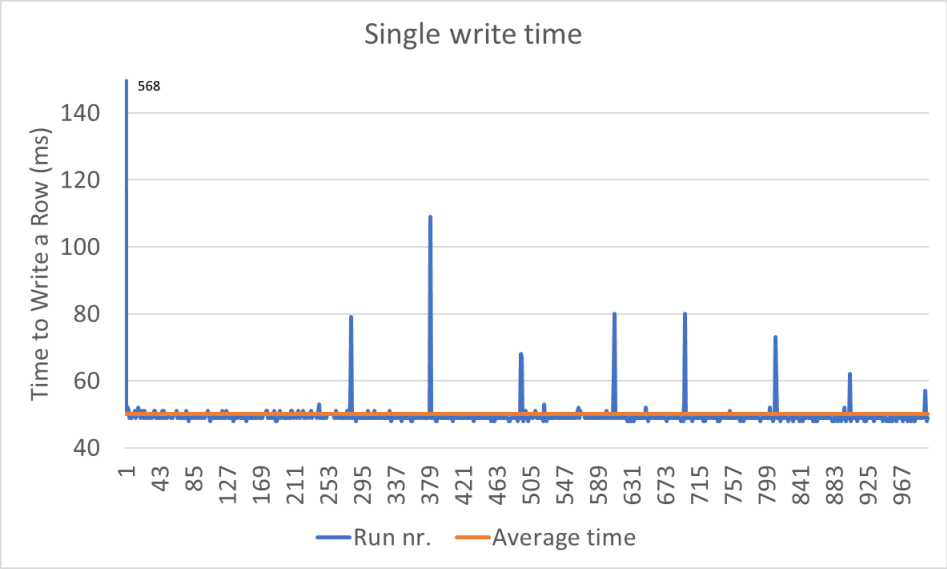

Figure 4.10: Single write call time on DDB.

prioritizes availability over consistency [50], its replication technique is more optimistic compared to the one used in Zookeeper, making the write available as soon as it receives the write request from the client. This test consisted in doing 1000 writes continuously without a time break between them.

Figure 4.10 presents the results from those tests. There were 1000 writes with a time average of 50 ms with a standard deviation of 16,65 ms and a 95-percentile of 51 ms. As can be seen in the figure, the first run created a peak by receiving an answer after more than 500 ms. This is due to the service taking more time to provide the first answer. The next requests were be answered faster.

The other eventual peaks are either caused by the internet connection latency or the service. Due to their rare occurrences and small increase of time, the chance to not provide a timely write is low.

4.3.3 Table Read Latency with Scan

Like the services before, this service was tested on the speed of getting all the information in the system. The objective is to obtain latency metrics about the scan. This service already has a function that does that automatically, removing the need for the client to invoke individual read calls with multiple threads; that function is called a *scan*. This test used the *scan* function to return all the relevant data (chosen by the client) from the table. To measure the time to execute a *scan*, Algorithm 1 was used. This way the database only receives one function call per client. The table has been setup with 1000 entries before the execution of the test.

As mentioned in the introduction, this service has some restrictions with the free-tier. By having a client constantly doing scan calls, the service automatically times out that

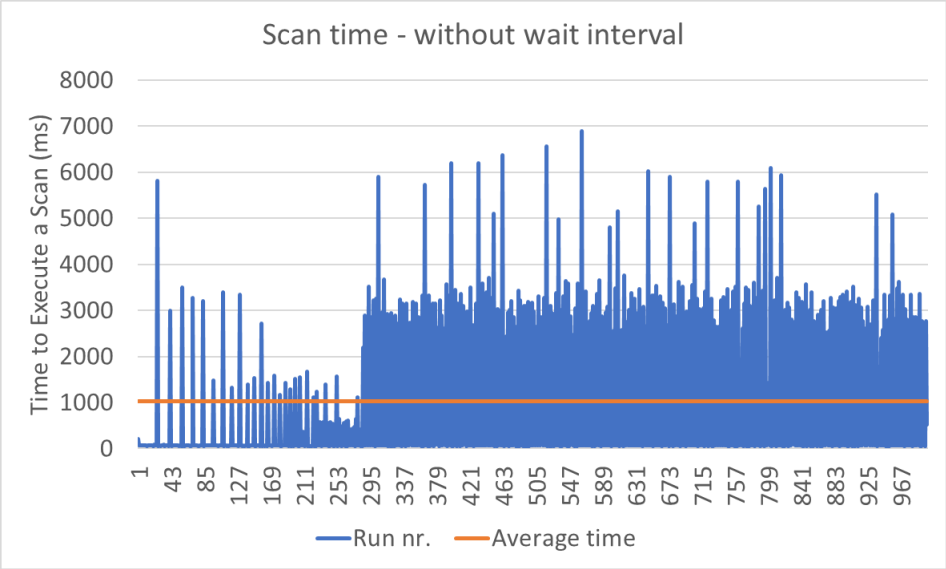

Figure 4.11: Scan function call of 1000 entries on DDB.

client which means that for a short period of time, the client can not do any remote calls to the service. This is because the DynamoDB service blocks clients that keep sending the same function call consecutively over a period of time to prevent DoS attacks. This behaviour was discovered after the first test run which had huge variations in time between calls. That test can be seen in Figure 4.11, with the average latency of 1031 ms and the standard deviation of 953,64 ms and a 95-percentile of 3453,05 ms. The work around was to make another test with 250 consecutive scan calls and, at the end of those runs, wait some time before doing another set of runs; with that it was possible to circumvent the problems described above. The results of this test can be seen in Figure 4.12. This situation will not pose a problem if a client does its calls with a time interval bigger than 500 ms between them.

The time variation seen after the 250th run is due to the said blockage of the client by the service, taking over 2 seconds to reply. This is not a performance problem as this system is prepared and built around the idea of serving a big number of clients' requests on a small time period.

Getting to the second test with the workaround described above, represented in Figure 4.12, the average time is of 95,6 ms with a standard deviation of 42,89 ms and a 95-percentile of 133 ms. The peak seen on the 1st and 251st run is due to the client program starting again. the first call always takes more time to return a value.

Taking the previous service tests where the single read time was always inferior to the full system read, using inductive reasoning this will also be the case. The expected time of the single read time is to be under or equal to the scan time which is around 100 ms.

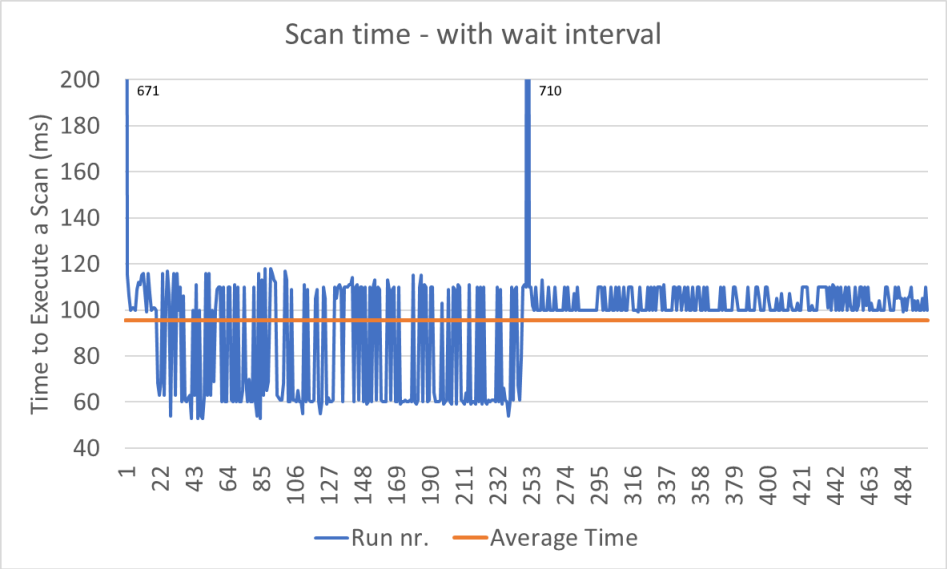

Figure 4.12: Scan function call with the workaround of 1000 entries on DDB.

4.4 Discussion

This section summarizes the metrics got from testing the services and display the thought process done to choose the service that better fit the needs for the idealized vehicular information system. The metrics to be discussed are pricing and latency values. Besides that, it will also be discussed the ease of use and implementation and scalability, coming from the (limited) experience of doing these tests. To compare the latency values, let us use the required two second upper bound for acceptable values for communication.

4.4.1 Pricing Comparison

Pricing-wise, the S3 and DDB use a pay-per-request metric and the EC2 uses a pay-per-hour metric (with the model used in the tests). To better compare the prices between the three services, let us imagine that the system hypothetically executes a million of each request type per day and never uses more than 25 GB of storage/month. If the system is being run on the EC2 service, it uses three t2.small instances for 24 hours/day with a total of 25 GB of gp2-provisioned storage volumes. For all services let's define that there is 1 TB of data transfer per month and that each month is comprised of 30 days. Table 4.6 shows the price of running each service per day under these conditions.

From the table it is possible to see that the DDB service is the cheapest of the three options, having the EC2 on second and the S3 service on third. Bear in mind that the EC2 service, due to scalability needs, might need to add more instances running, and most likely, increase the size of the already running instances, involving a higher cost than the one referenced here. That cost can overcome S3 costs quickly, although if a multi-bucket solution is used, the costs might equalize.

Services	Storage Price	Requests Price	Runtime Price	Data Transfer Price	Total Cost/day
S3	0,02$	5,72$	-	2,99$	8,73$
EC2	0,99$	-	1,87$	2,99$	5,85$
DDB	0$	1,78$	-	2,99$	4,77$

Table 4.6: Total cost of use of the services per day under a defined use case.

	Scan / 1000 reads	Write
S3	1300 ms	130 ms
EC2	242,2 ms	52,01 ms
DDB	95,6 ms	50 ms

Table 4.7: Comparison between the services on the fastest *scan* and *write* tests.

4.4.2 Latency Comparison

By comparing the latency metrics, the services are pretty balanced, having all the single operations, except the S3 write, to roam under the 100 ms. Although that is the case, what matters the most is the read time of the system as it will be needed as a primary function on the data management service. In that case the S3 service is the slowest one with its average time roaming around 1300 ms to do 1000 reads using a multi-threaded client. The only one above the second mark. However, bear in mind, that through the research of *Correia* [40], he concluded that the latency on Zookeeper would vastly increase over distance between service and client and it would not be a viable solution to use in a membership service. The results from the fastest scan method doing 1000 reads and single write from each service can be seen and compared in Table 4.7.

4.4.3 Other Comparisons

In terms of scalability and replication, S3 and DDB services are the easiest to use. They do everything on the background without needing any external changes. The Zookeeper mounted on the EC2 instances is a bit different. Even if the instances scale to comply with the resource needs of the Zookeeper cluster, it might be needed to create new instances to add new nodes to the cluster. Although that can be programmed to be done automatically, the objective here is to find the best ”out of the box” solution. On this topic, the EC2 falls behind the other services.

Last but not least, comes the ease of use which ties the DDB and S3 as they just require the client to log-in and use.

4.4.4 Conclusion

In conclusion, all the services have advantages and disadvantages associated with them and its usability depends on the use cases the services are submitted to. On this specific case, to create a shared memory to base the data management service, it could either be on the DDB or S3 service. Due to an already existing library created by *Oliveira et al.* [1] that is capable to emulate a shared memory based on the S3 service, we decided to use the S3 service for our cloud storage component of the vehicular information system.

Chapter 5

Client Models for the S3 Service

This chapter is dedicated to defining and creating client models that communicate with the S3 service in order to store and obtain data. The clients are based on the fat client architecture [51]. This means the clients need to do most, if not all, of the computing process to obtain data. This is due to the nature of the S3 service, where it only holds objects and has a simple API of requests. The clients will implement a modified library based on the Two-Step Full Replication algorithm [1], which provides integrity and replication guarantees. This library also emulates a shared memory based on multiple buckets by controlling how the clients access it. The shared memory is based on SWMR registers which can execute three functions: *scan*, *read* and *write*. There are two client implementations.

This chapter also describes the tests done to the client models created and their respective analysis. The tests are based on getting the latency of the *scan* and *update* of each kind of client for later comparison between both of them and check which one has less. The latency was measured using the same technique as the tests done on Chapter 4, using the Algorithm 1. All of the tests on this chapter were made with all clients being emulated in a single machine connected through fiber to the internet. Although the clients could do three different actions (*update*, *read* and *scan*) in the idealized client interface, the tests on the models only used two of them: *update* and *scan*. The reason behind this choice is because the *scan* function is a *read* call to all the registers. It is also expected for the *scan* to be the action to take the most time to execute, as it depends on the total number of clients. This dependency exists because the *scan* requires to read all the registers of the system. If each client has its respective register on the shared memory, the bigger the number of clients, the bigger the number of registers and thus, the scan will require more time to complete.

5.1 Atomic Snapshot Client Model

This client is constructed under the proposed client architecture and uses a modified version of the Two-Step Full Replication [1] library as the data management service component and an implementation of the unbounded single-writer algorithm by *Afek et al.* [2] as the Client Interface component. The atomic snapshot algorithm gives the client the ability to obtain an atomic snapshot of the shared memory being emulated by the modified Two-Step Full Replication API, which in turn, is connected to the S3 service. Both of these algorithms are explained in more detail in Chapter 2.

Using the Two-Step Full Replication algorithm as a data management service for this client guarantees the same integrity, authenticity and resilience levels as the default clients created using only that algorithm. The main changes made in the Two-Step Full Replication library were the following:

- It uses AWS S3 buckets exclusively instead of using a mix of key-value store services offered by multiple providers, transforming this library from multi-cloud to a single-cloud solution. This simplification also removes the Byzantine fault tolerance.

- The buckets use folders to be more organized and facilitate reads. Each folder represents a register which holds all the versions of the register.

- To make the client interface simpler and the data management service faster, the registers were changed from MWMR to SWMR. This choice was also made because clients do not need to write on other clients' registers.

- The default registers were changed from regular to atomic to be able to implement the Atomic Snapshot algorithm. In order to make them atomic, the read calls executed a write-back to the register when necessary. Since the writes were already time-stamped and signed due to the security and integrity requirements, it did not need any changes.

- It was created a *list* function in the Data Management Service component which returns all of the most recent values from all registers in a more efficient way compared to using the normal read function, as it would be more resource and time-consuming. The read function invokes a list call of metadata objects from the buckets which returns all the metadata information of a certain register and after checking which one is the most recent it returns the actual object (Figure 2.1). If this function was used in a loop to read all the registers it would do a number of registers (n) metadata lists (one for each register). Instead of using all of those metadata list calls, the new list function uses only one metadata list call to get all

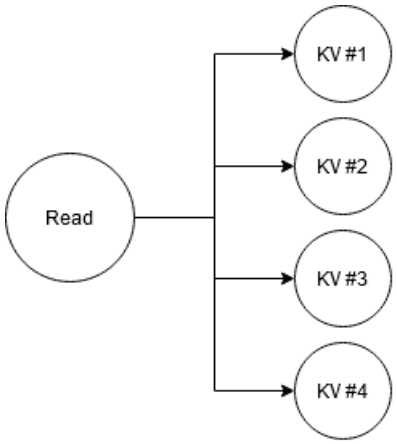

Figure 5.1: Workflow diagram of the Read function, where it reads one register.

of the metadata information of the bucket and separates all that metadata information throughout their respective registers. Afterwards it either reads the most recent object from each register in a single-threaded (ST) or multi-threaded (MT) version.

- Changed how the write function checked for the most recent object. Now that the registers are single-writer they only need to check at startup for an already existing time-stamp using the *maxValidVersion* call (Figure 2.1) since it will always know from there which value the most recent timestamp has. If there is not an already existing time-stamp on the register, it starts with the default value. This change improves performance and requires less calls to the cloud which reduces the overall cost.

We will now proceed to explain how the *read* call works and how the *list* function implementation differs from the ST and MT variants. Starting with the *read* call, which already is implemented in the default Two-Step Full Replication library [1], it already uses multi-threading to reach multiple KVSs and quorum decisions which made the system theoretically fast already. Figure 5.1 shows how the *read* call is done using the four (default) threads. It lists the register's stored objects on each KVS, and does its quorum decision, returning the most recent object on the register.

Moving onto the *list* function, it does a *read* call for each existing register, and collects the most recent values of each read into a single response. This is where the ST and MT versions of this client will differ. One has a loop that reads one register at the time (ST), as seen in Figure 5.2 and the other has a thread pool with submitted threads reading registers simultaneously (MT), as seen in Figure 5.3. By default the MT version uses a thread pool with four threads for testing (unless said otherwise).

The Atomic Snapshot client implementation is represented in Figure 5.4. It uses the atomic snapshot interface which has the *scan* and *update* functions from the atomic snapshot algorithm, seen in Figure 2.2, plus a single register read, which is based on calling

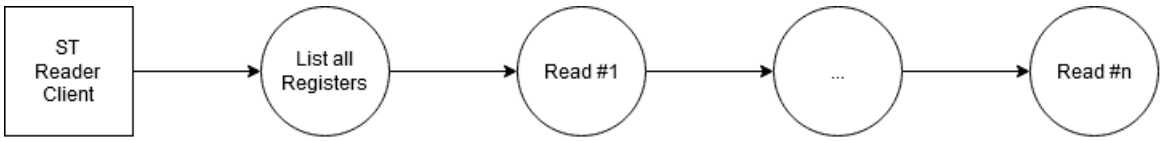

Figure 5.2: Workflow diagram of the Single-Threaded List function.

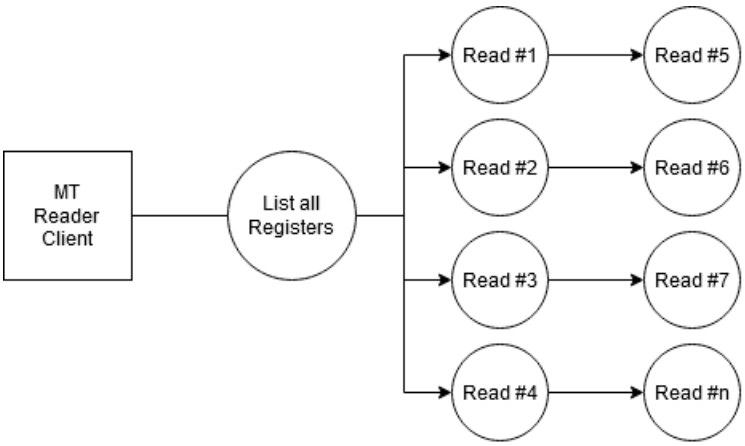

Figure 5.3: Workflow diagram of the Multi-Threaded List function.

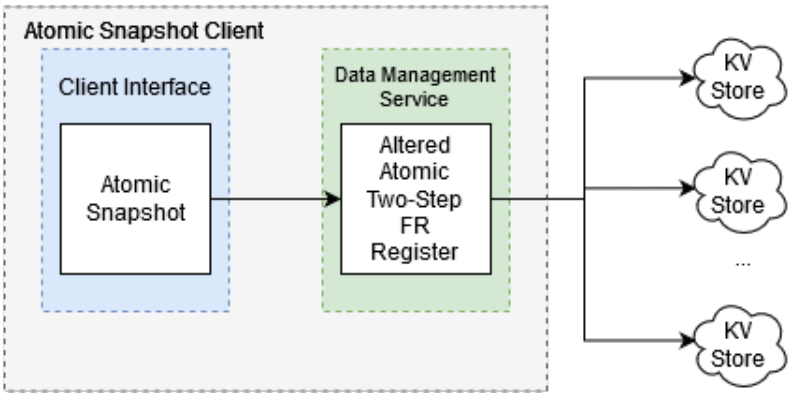

Figure 5.4: Simplified diagram of the Atomic Snapshot client.

the scan function and filtering the desired register data. That interface also makes the client to only be able to act on a register in a single-writer multi-reader fashion.

It can change the value from its own register by calling the *write* function and still be able to invoke the *scan* or *read* call to other registers. This was the simplest possible change to transform the Two-Step Full Replication Algorithm MWMR registers into SWMR. Going into the Two-Step Full Replication Algorithm class, it takes care of integrity and authenticity by hashing the value to be written with a signature and availability by using four replicated buckets in different areas, just like its default implementation explained in Chapter 2. It also uses multi-threading to communicate with the KVSs simultaneously by default.

5.2 Testing the Atomic Snapshot Client

The tests on this section focus on achieving the latency results of the two versions (ST and MT) of the Atomic Snapshot client under the same conditions. The results will be used to verify which one has better execution latency and to later compare with the other client implementation. Each test starts with four empty buckets and has a runtime of 5 minutes. The system was tested with 5, 10 and 15 clients simulated in a single computer, with each client being assigned to a client thread pool. From those n clients, half were writer clients, who only had to update their register and the remaining clients had to read all the registers (scan) from the other clients. Each client is running its respective call (scan or update) in a loop with a two second cooldown between them. In other words, it invokes a new call two seconds after the previous one is finished in order to maintain the CPU less constrained with tasks. Two tests were made for each n-client system, one using the scan function in a single-thread and another in a multi-threaded manner by using a thread pool with the number of threads with the same size of registers it needed to read. The objective of the tests is to compare the *scan* latency of the ST and MT variants. The tests results can be seen in Table 5.1. The ”ST” and ”MT” variants represent the Single-Threaded and Multi-Threaded test respectively.

Because the clients were simulated in a single computer, it was expected the latency to increase in the tests due to the CPU processing.

5.2.1 Results and Discussion

This section is dedicated to the discussion of the results obtained from testing the latency of the *scan* and *update* functions in the Atomic Snapshot client. Table 5.1 has the summarized results of all the tests done on this client. Bear in mind that the scan and update values are similar between the same *n*-client system (comparing ST and MT variants) because the atomic snapshot implementation used has the update calling the *scan* function, as it implements the atomic snapshot algorithm (Figure 2.2). This makes both functions

	Update	Scan
5 client ST	5,61s	5,62s
5 client MT	5,43s	5,55s
10 client ST	20,60s	20,93s
10 client MT	18,39s	24,38s
15 client ST	32,92s	55,32s
15 client MT	39,07s	36,84s

Table 5.1: Comparison table on the average latency to execute a command in all Atomic Snapshot client tests with a 2 second cooldown between calls.

have a similar latency, as the only difference between both of them is a write call. From previous testing, it is already know that a single write on the S3 takes around 130 ms and that is the minimum time difference to be expected between the scan and update functions on this client.

By analysing the results, it is possible to see that none of the n-client tests done go under two seconds latency mark. This shows that this client can not be used with the defined required latency (2 seconds). It was also noticed that the latency to execute a command would increase over the total runtime of the test. This is due to the ever increasing number of objects being written on the buckets caused by the client writes. A good solution to counteract the latency increase over the runtime of the system, would be implementing a Garbage Collector (GC) which would keep the number of objects to a minimum and maintaining the initial latency. Although it would make the system faster, by taking into account the latency of the initial calls (where all the registers were empty), that solution would not be enough, as an empty register doing the necessary calls with this data management service does not do it under the required time period. Figure 5.5 represents the average operation latency of a 5-client MT Atomic Snapshot client system with its respective trendline (m = 110,32). This shows that with this Atomic Snapshot client implementation is not possible to obtain latency under the required upper bound of two seconds.

Besides that, we also noticed that the ST version sometimes would take less time to execute a function call compared to the MT version. We believe that this is either caused by CPU throttle, where each client requires lots of processing power to execute and since more threads are submitted in the MT version, there are more threads to be scheduled, or because four threads submitted in the thread pool are not enough to make a difference between ST and MT versions. This problem will be analysed in the next client implementation.

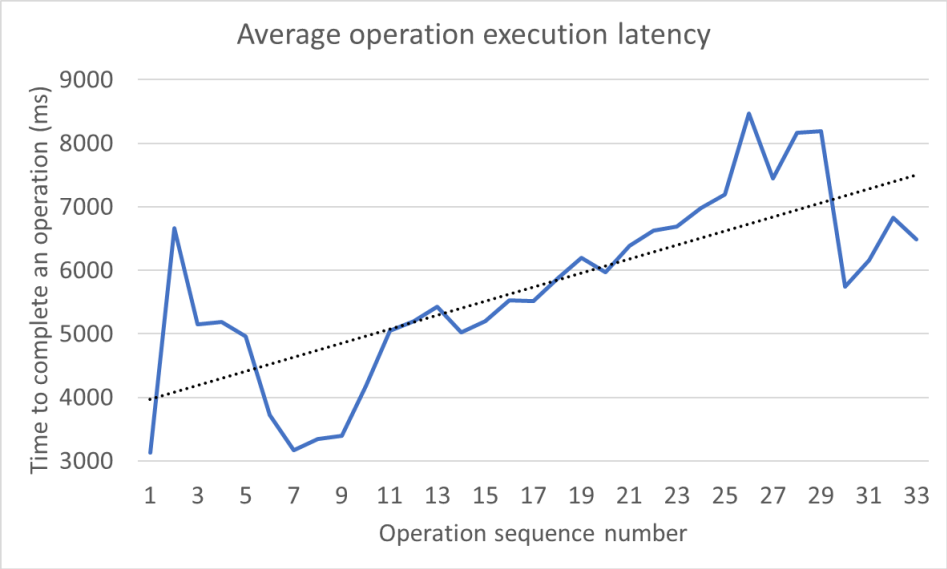

Figure 5.5: Average latency to execute an operation on a 5-client MT system with the Atomic Snapshot client.

5.3 Fast Snapshot Client Model

This client is an adaptation of the Atomic Snapshot client model with less strict register consistency, while maintaining the rest of the guarantees.

Figures 5.4 and 5.6 show the implementation differences of this model and the first one. One of the main differences is that the Atomic Snapshot class (client interface component) was replaced by a Fast Snapshot. Instead of implementing the unbounded single-writer (Algorithm 2.2), it simplifies the calls to their most basic form. To obtain a snapshot, the client invokes the *list* function (present in the Data Management Service component) and to write on the register, the client invokes the *update* function, which uses the *FR-write* function. We also changed the *read* function. It is no longer based on the *scan* call as it does needs to provide atomic guarantees. Now the *read* is done by simply accessing the desired register and invoke the *FR-read* function. These changes were done to increase the performance of these functions. We removed the existing write-back because the register's atomic guarantees were no longer needed. The write-back was being done every *scan* function which implied at least $2n$ (n being the number of registers) write-backs per scan call. The Fast Snapshot Client interface can be seen in Algorithm 2.

Besides those changes we also implemented a simple garbage collector client, to improve performance, as seen in Figure 5.7 which has its own direct connection to the KVSs. It checks each folder per bucket, in a multi-threaded manner, for its object size using a *list* call and act accordingly. It can either be run in a virtual machine on the cloud or local machine. A more in-depth explanation will be provided in the next section.

Algorithm 2: Fast Snapshot Client Interface

```
Function scan()
    return list();
end
Function update(data)
    FR-write(data);
end
Function read(registerID)
    register = getRegister(registerID);
    return register.FR-read();
end
```

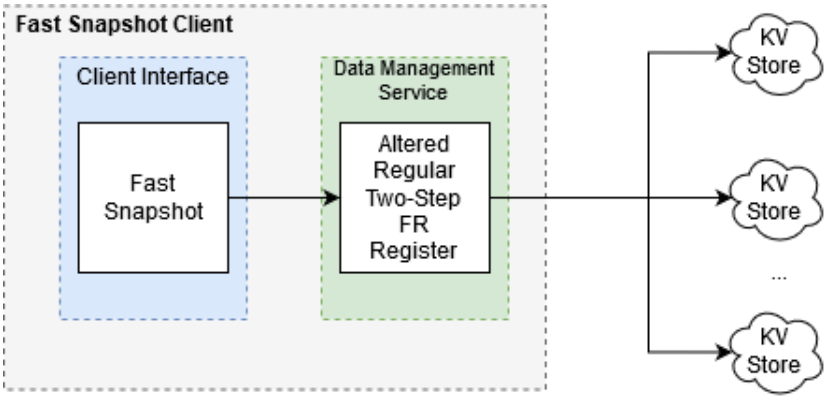

Figure 5.6: Simplified diagram of the Fast Snapshot Client.

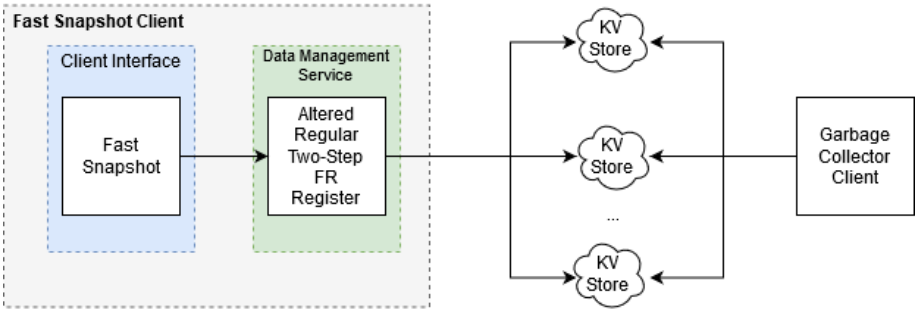

Figure 5.7: Simplified diagram of the Fast Snapshot Client with the Garbage Collector Client.

	Update	Scan
5 client ST	0,13s	1,06s
5 client MT	0,13s	1,07s
10 client ST	0,13s	3,46s
10 client MT	0,14s	3,53s
15 client ST	0,13s	7,60s
15 client MT	0,13s	7,62s

Table 5.2: Comparison table on the average time needed to execute a command in all fast snapshot client tests without Garbage Collector.

5.4 Testing the Fast Snapshot Client

This section is dedicated to the results obtained from the tests done to the Fast Snapshot Client. The initial tests are done in the same manner as it was done in the Atomic Snapshot client model, which is comparing in a same n client system, for a duration of 5 minutes, the latency it took for a client, on average, to complete its *scan* and *update* calls in a ST and MT variant. Unless said otherwise, the MT variant test used a thread pool with four threads like in the Atomic Snapshot client model tests. All of these tests were done with a two second cooldown between commands by the same reason done on the previous tests.

5.4.1 Results and Discussion

The results from the tests can be seen in Table 5.2. On this model, the ST and MT tests did not have any relevant differences between them. The *update* function took, on all tests, less than 0,15 s, on average, to finish. This value was expected, since the *update* function does not need to do any other action on the register besides writing on it. Those latency values are consistent with the ones made in the write tests of the S3 service in Chapter 4. The *scan* values are similar in the same numbered client systems because the thread pool is set to a small number of threads in comparison with the number of registers to read, and not taking full advantage of the multi-threading. This problem will be analysed further below as well as the data storage problem where the stale data is not removed from the shared memory.

5.4.2 Garbage Collector Implementation

To tackle the existing problem of the latency increase of the *scan* function over the course of a test due to the constant increase in numbers of objects on the buckets, we implemented the Garbage Collector Client in the cloud-based vehicular information system, like it can be seen in Figure 5.7. This Garbage Collector Client implementation can be created anytime during runtime, as it is a separate client. This client uses the S3 bucket API to have direct access to the bucket and delete files from it. When starting the GC client it

is possible to decide on a time period which it will sleep after finishing the cleaning, to reduce the CPU usage. After that it will do another cleaning sweep. If the time period is set to zero it is constantly running. We made this client to create a thread to access each bucket and make each thread work on each bucket as necessary.

When it is cleaning, it checks if a bucket folder has more than 10 objects in it, and if it does, it starts deleting the older file versions until he has only the 4 most recent versions in it. These values were chosen arbitrarily to obtain a noticeable latency difference between a full and cleaned register, as the latency difference between reading four to ten objects is high enough to be easily spotted. It was decided to keep four objects to maintain some older versions as backup if needed, as a real-world application of this client might need.

We made some tests to obtain data on the efficiency of the scan and update functions of the Fast Snapshot Client, with the garbage collector client implemented. Those tests are represented from Figure 5.8 to Figure 5.10 and followed the same principle as the previous tests with the difference of having a garbage collector client running simultaneously. The figures can interpreted by checking the x-axis which represent, in the 5 minute runtime of the test, the number of scans made. The y-axis shows the time in seconds it took for the scan x to complete. The black doted line shows the trendline of the time it took to finish a scan over the course of the test, where the lowest the slope, the better.

Figure 5.8 represents a 15 client MT test where the garbage collector client was not activated. This was done to obtain comparison values and use as group control. It was checked the time it took for every scan to complete, and from its analysis it was possible to confirm the that the last scans take more time to complete compared to the first ones. That can be confirmed by the steep slope on the trendline (m = 0,4157). This was the result of the writer clients keeping updating their register without the system ever deleting older values. Keep in mind that a list on the register implies the need to get all the objects metadata of that register in each replicated bucket and check, for the most recent version. Thus, the higher the number of objects in a bucket, the longer the list will take to be accomplished. A possible solution for this problem is to implement a garbage collector which will clean the registers by being activated with a temporal or file size trigger.

Having the control test done, we did the necessary tests using the garbage collector client to compare with it. The first one was a 15 client MT simulation with the garbage collector being invoked every five seconds to clean the registers. The next test was done the same way but with the difference of having the garbage collector being called every two seconds. The results of the tests can be seen in Figures 5.9 and 5.10. The average time of the commands executed during those tests can be found in Table 5.3.

Starting with the 15 client system with the garbage collector cleaning every 5 seconds, which can be seen in Figure 5.9, it shows that the slope steepness decreased drastically (m = 0,0004) compared with the slope analysed in Figure 5.8. That shows the difference between using and not using a garbage collector, decreasing the average scan latency from

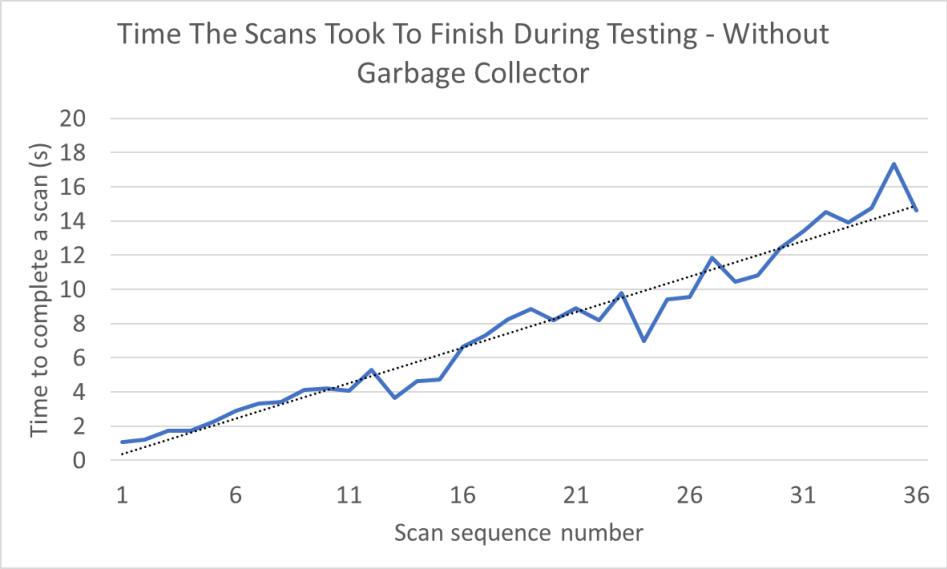

Figure 5.8: Scan latency in a 15 client system without GC.

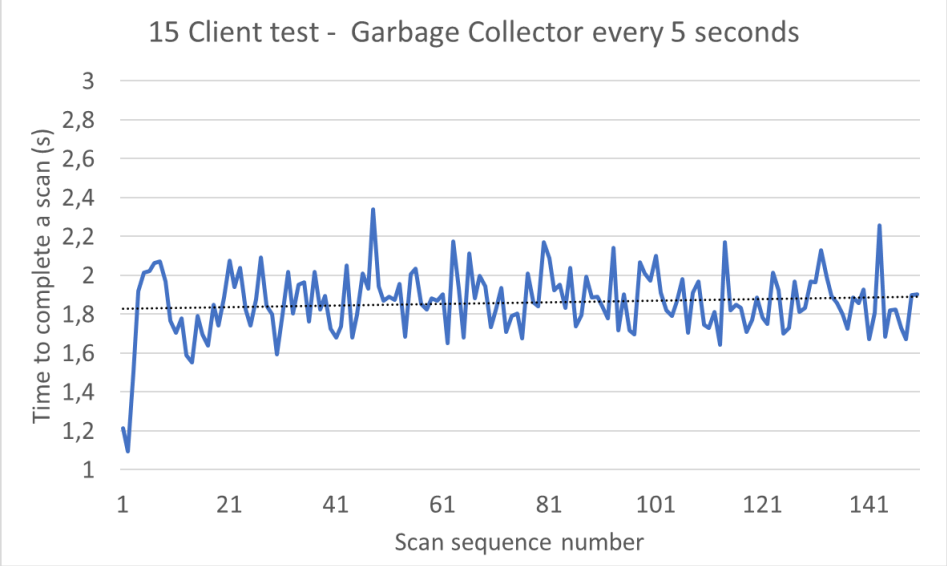

Figure 5.9: Scan latency in a 15 client system with GC running every 5 seconds.

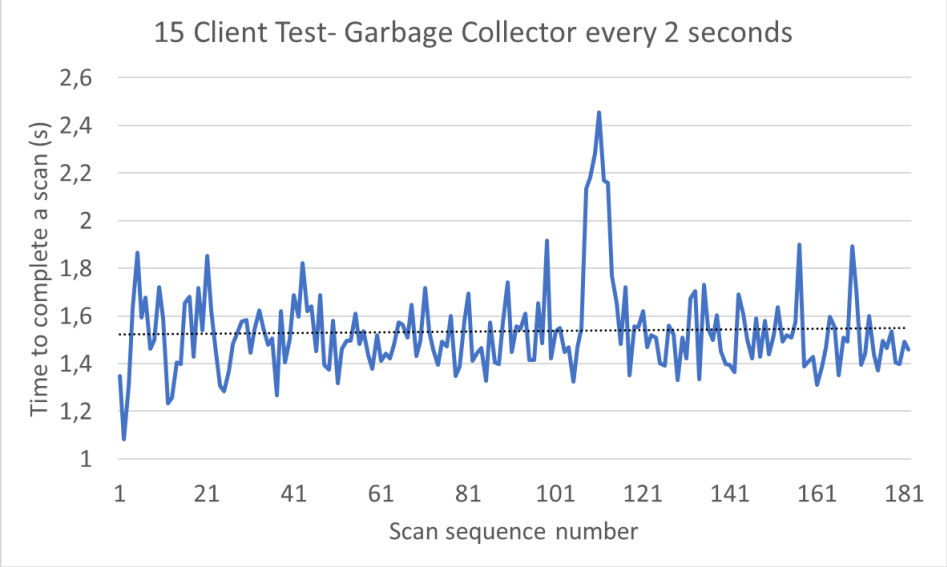

Figure 5.10: Scan latency in a 15 client system with GC running every 2 seconds.

	15 client - no GC	15 client - 5s GC	15 client - 2s GC
Update	0,13s	0,13s	0,14s
Scan	7,62s	1,86s	1,54s

Table 5.3: Comparison table on the average latency needed to execute a command in a system with or without Garbage Collector using different settings.

7,62s to 1,85s.

The results of the 15 client system test with the garbage collector cleaning every 2 seconds can be analysed on the same table as the previous test and in Figure 5.10. The slope steepness was further reduced (m = 0,0001) as well as the average latency to complete a scan. This result was expected as more the garbage collector client works and keeps the registers clean, the less versions are on the registers and needed to be read.

Note that by making the GC sweeps more frequent, the period of the functions are reduced but keep the same amplitude. Looking at the amplitude it is possible to see the difference between the same shared memory having scans when it has less and more objects, being those the relative minimum and maximum points in the function, respectively. To reduce the amplitude and stabilize it to the minimum value, it would be needed to change the maximum number of versions per register to be the minimum number of versions per register $+1$.

Although there is an associated cost with the *list* call required to delete objects (the delete call is free), the latency reduction shown compensates the cost associated.

Concluding on the garbage collector client, although we had promising results, we noticed that we were wrongly accessing the buckets by bypassing the shared memory abstraction by using this garbage collector client implementation. To solve this problem, further fast snapshot client implementations will have their own implementation of a garbage collector, where each client will have the responsibility to clean their own register by keeping count on the number of updates done and if they did enough updates to clean the desired older data stored. This change also distributes responsibilities between clients and reduces the use of cloud resources, saving in costs and leaving less clients to interact with the shared memory simultaneously. Although those changes are going to be applied on further implementations, we believe that having a garbage collector working is always better than having none, and even by changing the garbage collector implementation, the results obtained from using one will always be positive.

5.4.3 Changing the Scan Thread Pool

Although the scan latency was reduced through the use of a garbage collecting method, the difference between the ST and MT variants was still small. This was concerning as we expected to obtain a bigger scan latency difference between ST and MT variants, in particular, it was expected for the MT variant be faster than the ST one. Because of

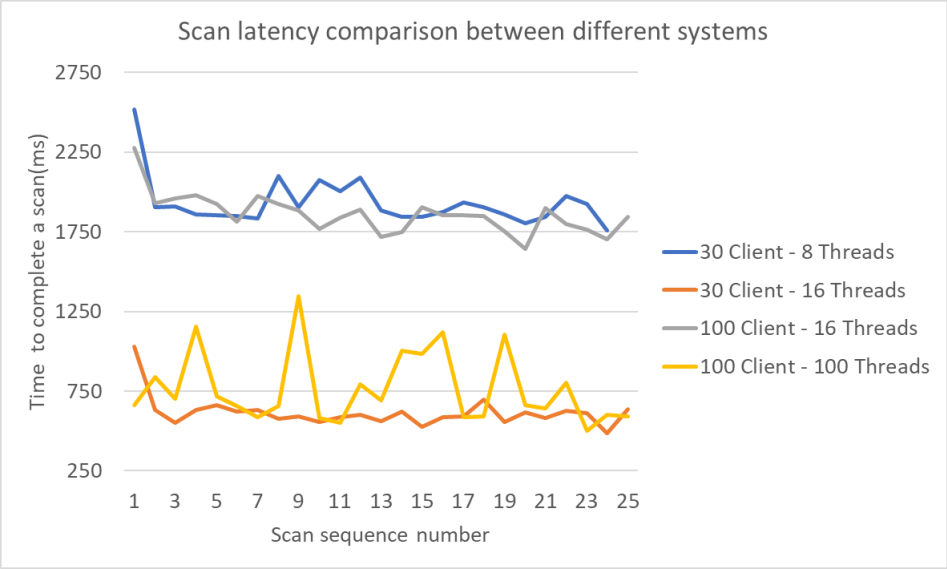

Figure 5.11: Scan latency comparison between different systems.

that, we decided to investigate the reason behind this problem. A test was made which consisted in increasing the number of threads in the thread pool attributed to each scan function on the MT variant, expecting to have direct effect on the scan latency.

As the *update* function was being continuously done in 0,13 s and was not dependant from the scan thread pool, the following tests were only done to the *scan* function. The tests considered a 30 and a 100 client system, with the GC turned on and with a runtime of five minutes, where the only variable to be changed was the number of threads dedicated to the scan function thread pool (T). The results obtained can be seen in Figure 5.11, where it shows the latency it took to execute the first 25 *scan* calls during each system test. By analysing it, we can see that a scan thread pool with a number of assigned threads closer to the total number of clients has lower latency. Table 5.4 shows the average latency, standard deviation and 95-percentile of the results obtained on the tests.

These results support the conclusion that a scan thread pool with more assigned threads, do their scans faster than ones with less. Besides that, it was also noticed a more than one second improvement comparing the 100 client tests of 16 and 100 threads, and even the 100 client test with 100 threads is faster than the 30 client, 8 thread system. Summing up, the best option to obtain faster scans is to employ as many threads as possible in the scan function, maxing out on the number of registers the client reads in parallel.

5.5 Final remarks

This chapter was dedicated to the implementation and test of two clients that used the S3 service to emulate a shared memory. The atomic snapshot client prioritizes consistency

Scan	Average latency	Standard deviation	95-Percentile
30 client - 8T	1,93s	0,149s	2,097s
30 client - 16T	0,60s	0,076s	0,681s
100 client - 16T	1,86s	0,122s	1,98s
100 client - 100T	0,77s	0,220s	1,148s

Table 5.4: Average latency, standard deviation and 95-percentile of the tests done on the *scan* in an n-client system with *T* threads per client with garbage collector.

over speed and the fast snapshot client, the opposite. Although less consistent, it can do its *update* and *scan* functions under the two second mark, something that the first client is not able to. It was on the second model that a few problems were analysed such as the growing number of older writes being kept on the registers and the influence of the number of threads assigned to the scan thread pool, as the latency results were not being as low as expected on this model. To solve these problems, we implemented a functioning GC client to check their influence over the scan latency and tested the client's scan thread pool to reach a conclusion on the ideal number of threads to be attributed to the thread pool assigned for the *scan* function in the MT version. It also shows that the Fast Snapshot client MT version is the only one meeting the two second requirement and will be the one to be adapted and used for making the vehicular and calculator client for the cloud-based vehicular information system test.

Chapter 6

Implementing a Cloud-based Vehicular Information System

The Fast Snapshot Client model was used by two kinds of clients: the vehicle clients and calculator client. Vehicle clients represent vehicles in a real scenario, and therefore they periodically update a specific register in the cloud with vehicle data (which could be their position, speed, intentions, etc). On the other hand, the calculator client represents a service that could be provided to vehicles, like a membership service or an accident warning service, and hence reads vehicle data from several registers and performs some calculations, to obtain the information that must be provided to vehicles. This information is then written in a specific register for each vehicle (the information might be different for each vehicle) which will finally be read, also periodically by the vehicle clients.

While in the previous chapters our objective was to evaluate and compare the performance of a generic cloud client considering different client models, we now consider a more realistic usage of the shared memory abstraction provided by the Fast Snapshot Client model, when it is used by two different kinds of clients to support information exchange. We aim at being able to discover if the approach scales to a large number of vehicle clients, given that now there will be several and different operations being performed simultaneously on the cloud by both types of cloud clients.

6.1 Assumptions

For this vehicular system to work properly and get valid data to analyse we made the following assumption:

- The clients are not malicious and thus, write the correct information on their registers and do not impersonate other clients.

This assumption is to guarantee security and that no faulty data is put on the vehicular information system, as it has no way to protect itself from it.

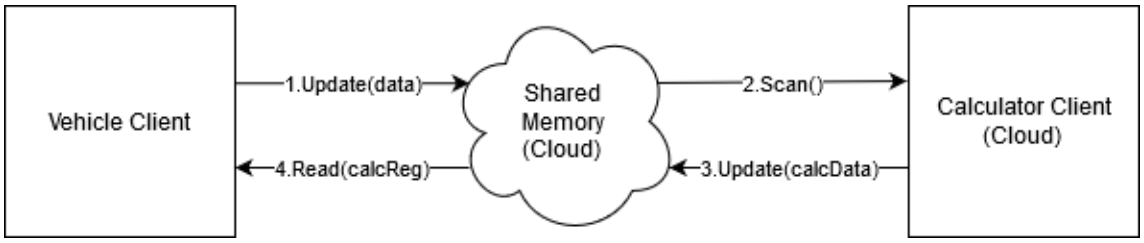

Figure 6.1: Dataflow diagram of the cloud-based vehicular information system between a vehicle client and its respective calculator client.

6.2 Implementation Fundamentals

This section gives an overview of the dataflow to be used by the cloud-based vehicular information system as well as its components and their respective functions. It will be using the Fast Snapshot Client data management service which will emulate a shared memory on the AWS S3 and use its client interface functions to make two types of clients, the vehicle and calculator client. The vehicle client, which is remotely connected, can write sensor information (such as location) on its respective register and read from one of the calculator client registers. The calculator client, which is based on the cloud, can retrieve the information written by all the vehicle clients by performing a scan and write on its register. The two types of clients will be working together on this system. This means that the vehicle clients will be working with the calculator client in order to obtain the necessary data to make a decision. Although there is that relationship between the two clients, they will not communicate directly, but read each other's registers. Also note that the same type of clients will always work independently.

The vehicle clients and calculator client will communicate the following way:

1. The vehicle client updates its register with sensor information
2. The calculator client reads the all the vehicle client registers, and does the necessary calculations.
3. The calculator client writes the results of the respective vehicle analysed in its register.
4. The vehicle client reads its appointed calculator register with the processed data and acts accordingly.

Figure 6.1 presents the interactions described above between a vehicle and a calculator client. The diagram shows two squares and a cloud figure. Each square represent a specific client and its interactions on the shared memory, represented in the middle as a cloud figure. The objective of this system is to host a large number of vehicle clients and their respective calculator clients. By following this structure it is hoped to achieve fast responses, redundancy, availability.

6.3 Clients Implementation

As described before, the Fast Snapshot Client implements the *update*, *scan* and *read* functions. The vehicle client only requires the *update* and the *read* function where it reads from a specific register. The calculator client needs the *scan* and *update* functions. The calculator client requires the *scan* function as it needs to fetch the data from the vehicular client registers.The biggest difference present on these clients is that they also run a garbage collector thread instead of leaving that responsibility to a dedicated client in the cloud due to the reasons mentioned in Section 5.4.2. On this specific vehicular information system it was decided that it was sufficient for the garbage collector to only maintain the two most recent values written on it.

Let us define that the shared memory can support up to n vehicle clients and that it will have at maximum $2n + 1$ registers which are divided through their numeric ID's in three parts:

- Vehicle Registers - Each existing register is controlled by a vehicle (or vehicular) client and goes from ID 0 to $n - 1$. This vehicular information system initializes a new register ID (writes on the shared memory the first value) when a vehicle joins.
- Calculator Registers - All of these registers are controlled by the calculator client. Each register is controlled by a dedicated thread. Goes from ID $n + 1$ to $2n + 1$. A calculator register with the ID $n + x + 1$ provides data to the vehicle client with the ID x.
- Vehicle Counter Register - This register, with ID n, is controlled by a specific thread from the calculator client called "Vehicle Counter Thread" to check if there is any new vehicle joining the current cloud-based vehicular information system by checking for new initialized registers by using the *scan* function, from ID's 0 to $n - 1$. This register exists to maintain the shared memory abstraction where each joined process requires to have a register assigned. The register is not used to hold any data.

The shared memory structure and maximum number of vehicle clients supported (n) is known by all the types of clients and will be used by them to access relevant registers for their respective responsibilities. Therefore, by knowing its ID (x) and the maximum number of vehicle clients supported (n), the vehicle client knows which calculator register ID has the specific information for it by using the formula ($n + x + 1$). The calculator client, also uses that knowledge to be able to do a *scan* function call from 0 to $n - 1$, having the guarantee that it will find all the existing vehicular clients registers required to do its calculations.

Figure 6.2 shows the shared memory organization mentioned above and provides an example of how a vehicle client with ID x interacts with the shared memory by writing

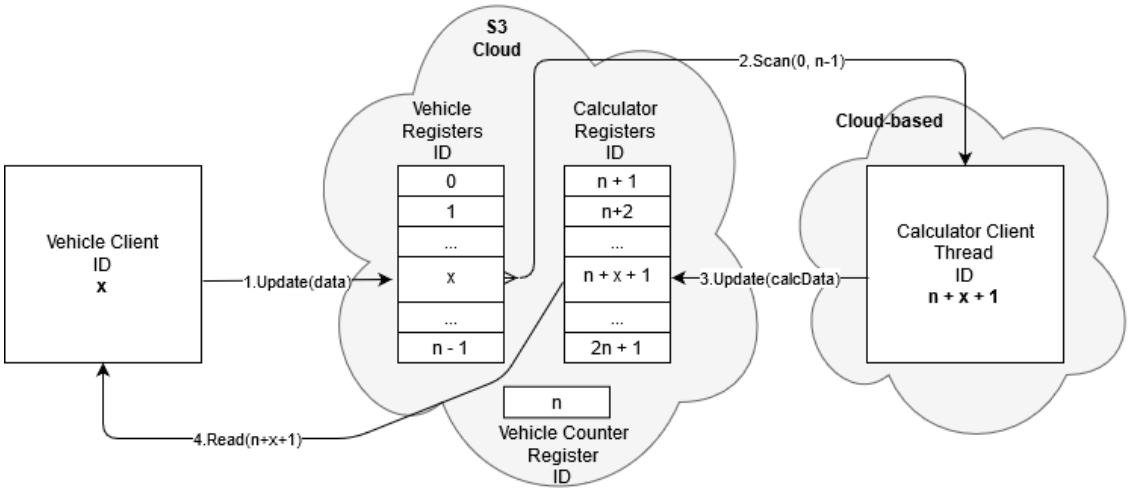

Figure 6.2: Dataflow diagram between a vehicle client with ID x and its respective calculator register with ID $n + x + 1$.

on its respective register x and reading its respective calculator register $n + x + 1$.

Figure 6.3 shows a more abstract view of the cloud-based vehicular information system, showing the calculator client based on the cloud with its respective calculator threads, where each interacts with a calculator register and the vehicle counter thread whose function is to detect any new initialization on the client registers group and create a new calculator thread for each vehicular client who joined.

6.4 System Analysis

This section is dedicated to analyse the total execution latency of the different clients present on this vehicular information system and check the maximum number of clients it can serve without surpassing the required time limit. We acquired the latency of both types of client completing their required set of function calls from start to finish. In other words, we measured the execution latency of a vehicle client to execute an *update* and *read* call and the execution latency of a calculator client thread to execute a *scan* and *update* call. The tests had a runtime of five minutes, having the client executing its functions every two seconds after the last set of functions was finished. Every time the client finished its execution, the latency was measured and saved. At the end of the test we averaged those values.

The vehicular clients average latency, standard deviation and 95-percentile of invoking the functions *update* and *read* can be seen on Table 6.1. Being independent from the number of active clients on the vehicular information system, the vehicular client latency does not vary that much throughout the tests. The values obtained are quite satisfactory as

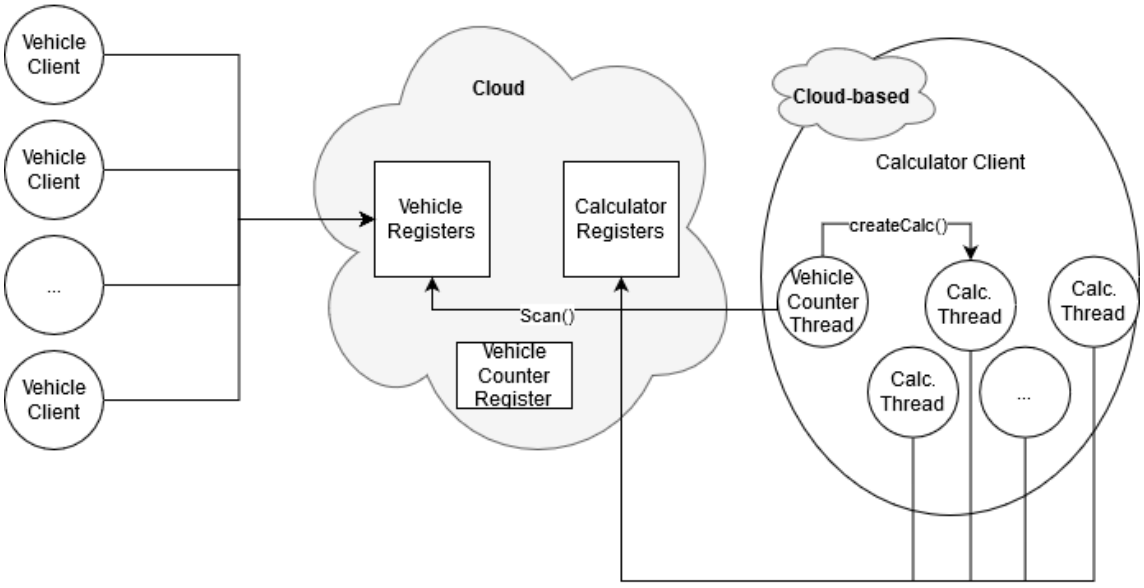

Figure 6.3: Diagram of the implemented cloud-based vehicular information system with special focus on the Calculator Service interactions.

	Average time	Standard deviation	95-percentile
500 Vehicle Clients	0,232s	0,017s	0,250s
750 Vehicle Clients	0,240s	0,018s	0,252s
1000 Vehicle Clients	0,238s	0,018s	0,250s
1500 Vehicle Clients	0,230s	0,016s	0,251s

Table 6.1: Average latency, standard deviation and 95-percentile, in seconds, of the vehicle clients routines in the vehicular information system with n vehicle clients.

	Average time	Standard deviation	95-percentile
500 Vehicle Clients	1,28s	0,282s	1,95s
750 Vehicle Clients	1,56s	0,279s	2,01s
1000 Vehicle Clients	2,13s	0,562s	3,35s
1500 Vehicle Clients	2,76s	0,418s	3,08s

Table 6.2: Average latency, standard deviation and 95-percentile, in seconds, of a single calculator client thread to execute its routine with n vehicle clients in the vehicular information system.

they are well under the required two second mark and because these values are not affected by the number of clients running in the vehicular information system, as each acts on one register only. This is not the case for the calculator client, where every calculator thread executes the *update* and *scan*, and it is known that it is dependant on the number of total vehicle clients present in the vehicular information system.

The calculator client does not do any calculations on this implementation, as that is not what is intended to be analysed in this thesis, besides, it is assumed that the calculator client has infinite resources (as it is implemented in the cloud) and so, it is possible to deduce that the time it needs to calculate a result is so small that is negligible.

To test the calculator client, we created a calculator client thread and ran it while the shared memory had n vehicle registers with two objects each. Those registers were not being updated while the calculator client ran because running a high number of clients in a single machine would affect the latency results. That is why it was decided to use registers with two objects, the maximum number of objects per register and thus obtaining the higher latency values possible in that specific system.

The results of the calculator client tests can be seen in Table 6.2. It shows the average latency, standard deviation and respective 95-percentile that one calculator client took to execute its *scan* function (using the thread pool with the same number of threads as the number of clients) and *update*.

Using the two second requirement as the determining factor to limit the maximum number of participating vehicular clients on this specific vehicular information system, we can draw a number between 750 to 1000 vehicular clients. We do not provide a specific number of clients as it is not possible to do so unless we test by trial and error through every possible number of clients in the defined range.

6.5 Final Remarks

This chapter explained how the Fast Snapshot Client model was used by vehicular and calculator clients for the vehicular information system. Afterwards, it presented the vehicular information system architecture followed by some assumptions and requirements. We also explained how the shared memory was being organized to be used in this setting.

An analysis of the system was made to answer the question of how many vehicular clients can be run in this cloud-based vehicular information system, without having the latency surpass the two second mark. The analysis consisted on various tests, changing the number of participating vehicular clients and the threads used for the *scan* thread pool of the calculator client thread to match the number of vehicular clients.

The biggest limitation of this vehicular information system is on how the calculator client handles the leaving vehicle clients. When a vehicle client leaves the system, the calculator client is not capable to recognize the ID of the client who left and thus, keeps running the specific calculator thread. This wastes cloud resources and consequently, increases the cost of use.

Chapter 7

Conclusion and Future Work

In this dissertation we presented a solution for cloud-based vehicular information systems which require timely responses while guaranteeing some data coherence. To be able to test the architecture capabilities we started by making an analysis in price and latency of communication between a client and different Amazon cloud services. After analysing three of the AWS provided services, S3, EC2 and DDB, we chose the first service and implemented a client library that allowed the clients to access a set of S3 Key-Value Stores as a shared memory. Using that library, we created two different clients that were able to *scan*, *read* and *update* the registers of the emulated shared memory. One works with atomic consistency guarantees and the other one with regular consistency. Having latency speed in mind we chose the fastest client implementation and used it as two different clients for a cloud-based vehicular information system implementation.

From testing the vehicular information system it was possible to analyse how many vehicular clients could work together in a cloud-based vehicular information system without breaking the two second requirement. The number of vehicular clients supported was between 750 and 1000, as the number depends on the internet connection.

7.1 Future work

Although the conclusion has been drawn there is still room for improving the data management service that was used during the testing in order to make it fully functional and capable for real-world usage. The algorithm used was heavily adapted to be used on this setting and might be unoptimized. So we propose to make a new algorithm implementation specific for this use case.

To improve our current implementation, we can transform it in a multi-cloud solution as it can provide better performance and tolerate Byzantine faults [52].

On the vehicular information system, to solve the vehicle client leaving problem, we propose the use of an active set abstraction [53]. On this case, an active set holds a set of processes that joined the active set. Those processes, to be able to join the active set

require to implement three functions. One that allows the process to join the active set, another to leave and the final one to obtain a set of active processes who are currently joined in the active set. By having the Vehicle Counter Thread in the active set and by making all the Vehicular Clients join it at the time of connection and leave it at the time of disconnection, it is possible for the Vehicle Counter Thread process to check for any leaving clients and inform the Calculator Client to stop any Calculator Thread that was working for a vehicular client that left. This will solve the Calculator Threads problem where they would keep working for vehicle clients that had already left the vehicular information system.

Our final proposal is to create a similar data management service component based on DynamoDB in order to compare with our current data management service implementation. The DynamoDB service proved to be faster and cheaper than the S3 service, and was not chosen as it would require much more time to fully implement a data management service with the same guarantees as the one used for the S3 service. If this solution proved to be better, it could be improved into a multi-cloud solution using another cloud service providers with similar services.

Bibliography

[1] Tiago Oliveira, Ricardo Mendes, and Alysson Bessani. Exploring Key-Value Stores in Multi-Writer Byzantine-Resilient Register Emulations. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] Yehuda Afek, Danny Dolev, Hagit Attiya, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. volume 40, pages 1–13, 01 1990.

[3] Daniel F. Howard and Danielle Dai. Public perceptions of self-driving cars: The case of berkeley, california. 2014.

[4] Wilko Schwarting, Javier Alonso-Mora, and Daniela Rus. Planning and decision-making for autonomous vehicles. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(1):187–210, 2018.

[5] Wu He, Guoqiang Yan, and L.D. Xu. Developing vehicular data cloud services in the iot environment. volume 10, pages 1587–1595, 05 2014.

[6] Chaudhary Muhammad Asim Rasheed, Saira Gilani, Sana Ajmal, and Amir Qayyum. *Vehicular Ad Hoc Network (VANET): A Survey, Challenges, and Applications*, volume 548, pages 39–51. 03 2017.

[7] Giuseppe Araniti, Claudia Campolo, Massimo Condoluci, Antonio Iera, and Antonella Molinaro. Lte for vehicular networking: A survey. *IEEE Communications Magazine*, 51:148–157, 05 2013.

[8] K. Abboud, H. A. Omar, and W. Zhuang. Interworking of dsrc and cellular network technologies for v2x communications: A survey. *IEEE Transactions on Vehicular Technology*, 65(12):9457–9470, 2016.

[9] Md Whaiduzzaman, Mehdi Sookhak, Abdullah Gani, and Rajkumar Buyya. A survey on vehicular cloud computing. *Journal of Network and Computer Applications*, 40:325 – 344, 2014.

[10] K. Hammoudi, H. Benhabiles, M. Kasraoui, N. Ajam, F. Dornaika, K. Radhakrishnan, K. Bandi, Q. Cai, and S. Liu. Developing vision-based and cooperative vehicular embedded systems for enhancing road monitoring services. *Procedia Computer Science*, 52:389 – 395, 2015. The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015).

[11] Subir Kumar Sarkar, T. G. Basavaraju, and C. Puttamadappa. *Ad Hoc Mobile Wireless Networks: Principles, Protocols, and Applications*. CRC Press, Inc., USA, 2nd edition, 2013.

[12] Rakesh Shrestha, Rojeena Bajracharya, and Seung Yeob Nam. Challenges of future vanet and cloud-based approaches. *Wireless Communications and Mobile Computing*, 2018:1–15, 05 2018.

[13] E. Ahmed and H. Gharavi. Cooperative vehicular networking: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 19(3):996–1014, March 2018.

[14] M. Celebiler and G. Stette. On increasing the downlink capacity of a regenerative satellite repeater in point-to-point communications. *Proceedings of the IEEE*, 66:98 – 100, 02 1978.

[15] Dong Chen. A survey of ieee 802.11 protocols: Comparison and prospective. In *Proceedings of the 2017 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering (ICMMCCE 2017)*, pages 569–578. Atlantis Press, 2017/09.

[16] Daniel Jiang and Luca Delgrossi. Ieee 802.11p: Towards an international standard for wireless access in vehicular environments. pages 2036 – 2040, 06 2008.

[17] Jacob Dethan. Wireless broadband (umts). 03 2015.

[18] John Kenney. Dedicated short-range communications (dsrc) standards in the united states. *Proceedings of the IEEE*, 99:1162 – 1182, 08 2011.

[19] Akhil Gupta. A survey of 5g network: Architecture and emerging technologies. *Access, IEEE*, 3:1206–1232, 08 2015.

[20] X. Ge, Z. Li, and S. Li. 5g software defined vehicular networks. *IEEE Communications Magazine*, 55(7):87–93, 2017.

[21] S. Patidar, D. Rane, and P. Jain. A survey paper on cloud computing. In *2012 Second International Conference on Advanced Computing Communication Technologies*, pages 394–398, Jan 2012.

[22] Michael Armbrust, Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. *University of California at Berkeley UCB/EECS-2009-28, February*, 28, 01 2009.

[23] Meenu Dave. Sql and nosql databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 08 2012.

[24] What is amazon dynamodb? `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html`. (Accessed on 12/09/2019).

[25] A technical overview of azure cosmos db — azure blog and updates — microsoft azure. `https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db/`. (Accessed on 14/09/2019).

[26] Cloud bigtable: Nosql database service — google cloud. `https://cloud.google.com/bigtable`. (Accessed on 15/09/2019).

[27] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *ACM Trans. Storage*, 9(4), November 2013.

[28] Yan Zhang, Stein Gjessing, Wenlong Xia, and Kuanli Yang. Toward cloud-based vehicular networks with efficient resource management. *Network, IEEE*, 27, 08 2013.

[29] W. Richard Stevens. *UNIX Network Programming, Volume 2 (2nd Ed.): Interprocess Communications*, chapter 12 - Shared Memory Introduction. Prentice Hall PTR, USA, 1998.

[30] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: concepts and systems. *IEEE Parallel Distributed Technology: Systems Applications*, 4(2):63–71, 1996.

[31] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distrib. Comput.*, 11(4):203–213, October 1998.

[32] Amazon web services (aws) - cloud computing services. `https://aws.amazon.com/`. (Accessed on 5/11/2019).

[33] Build your next great idea in the cloud — microsoft azure. `https://azure.microsoft.com/en-us/`. (Accessed on 5/11/2019).

[34] Google cloud computing, hosting services & apis. `https://cloud.google.com/`. (Accessed on 5/11/2019).

[35] Introduction to amazon s3 - amazon simple storage service. `https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html`. (Accessed on 08/09/2020).

[36] J. S. Plank. Erasure codes for storage systems: A brief primer. *;login: the Usenix magazine*, 38(6), December 2013.

[37] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[38] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[39] Antonio Casimiro, Emelie Ekenstedt, and Elad Michael Schiller. Membership-based manoeuvre negotiation in autonomous and safety-critical vehicular systems. *ArXiv*, abs/1906.04703, 2019.

[40] T. Correia. Design and implementation of a cloud-based membership system for vehicular cooperation. Master's thesis, Faculdade de Ciências, Universidade de Lisboa, October 2019.

[41] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[42] Guofa Li, Shengbo Li, Lijuan Jia, Wenjun Wang, Bo Cheng, and Fang Chen. Driving maneuvers analysis using naturalistic highway driving data. 09 2015.

[43] What is amazon ec2? - amazon elastic compute cloud. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html`. (Accessed on 5/11/2019).

[44] What is amazon s3? - amazon simple storage service. `https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html`. (Accessed on 5/11/2019).

[45] Amazon s3 simple storage service pricing - amazon web services. `https://aws.amazon.com/s3/pricing/`. (Accessed on 14/08/2020).

[46] Ec2 on-demand instance pricing – amazon web services. `https://aws.amazon.com/ec2/pricing/on-demand/`. (Accessed on 14/08/2020).

[47] Amazon ebs pricing - amazon web services. https://aws.amazon.com/ebs/pricing/. (Accessed on 13/08/2020).

[48] Amazon elastic block store (amazon ebs) - amazon elastic compute cloud. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html. (Accessed on 13/08/2020).

[49] Amazon dynamodb pricing for on-demand capacity. https://aws.amazon.com/dynamodb/pricing/on-demand/. (Accessed on 14/08/2020).

[50] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[51] Fundamentals of communication and networking: Thin versus thick client computing - wikibooks, open books for an open world. https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_2/Fundamentals_of_communication_and_networking/Thin_versus_thick_client_computing. (Accessed on 28/05/2020).

[52] R. Mendes, T. Oliveira, V. V. Cogo, N. F. Neves, and A. N. Bessani. Charon: A secure cloud-of-clouds system for storing and sharing big data. *IEEE Transactions on Cloud Computing*, pages 1–1, 2019.

[53] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long-lived and adaptive atomic snapshot and immediate snapshot (extended abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 71–80, New York, NY, USA, 2000. Association for Computing Machinery.

[54] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, Philadelphia, PA, June 2014. USENIX Association.

[55] Avinash Lakshman and Prashant Malik. Cassandra — a decentralized structured storage system. *Operating Systems Review*, 44:35–40, 04 2010.

[56] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 213–226, Berkeley, CA, USA, 2008. USENIX Association.

[57] Mohammed AlZain, Eric Pardede, Ben Soh, and James Thom. Cloud computing security: From single to multi-clouds (pdf). *Hawaii International Conference on System Sciences*, 0:5490–5499, 01 2012.

[58] S. Gilbert and N. Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.