Universidade de Aveiro
2021

Ricardo Carvalho Chaves

On the Study of IPFS as a Content Distribution Protocol for Vehicular Networks

Estudo do IPFS como Protocolo de Distribuição de Conteúdos em Redes Veiculares

Universidade de Aveiro
2021

Ricardo Carvalho Chaves

On the Study of IPFS as a Content Distribution Protocol for Vehicular Networks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica da Doutora Susana Isabel Barreto de Miranda Sargento, Professora Catedrática do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro, e co-orientação científica do Doutor Nuno Miguel Abreu Luís, Professor Adjunto do Instituto Superior de Engenharia de Lisboa, e do Doutor Ricardo Matos, Responsável pela Gestão de Tecnologia e Propriedade Intelectual na Veniam.

o júri / the jury

presidente / president	**Doutor Paulo Jorge Salvador Serra Ferreira** Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (por delegação da Reitoria da Universidade de Aveiro)
vogais / examiners committee	**Doutor Pedro Nuno Miranda de Sousa** Professor Auxiliar do Departamento de Informática da Universidade do Minho
	Doutora Susana Isabel Barreto de Miranda Sargento Professora Catedrática do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro (orientadora)

agradecimentos / acknowledgements

Esta dissertação foi o trabalho mais difícil e demorado que alguma vez fiz. Não tanto pelos aspetos técnicos ou a escrita, mas como é algo maioritariamente individual, tenho menos motivação do que quando há mais alguém. Um pouco na lógica de "quando há outros, mesmo que não hajam prazos, tem de ser feito bem e o mais cedo possível porque há um grupo a depender de mim, não os posso prejudicar", enquanto que sozinho isso não existe porque o único prejudicado sou eu. É difícil mudar/corrigir porque é preciso ir fundo para trocar uns fios de sítio, mas qualquer pressão externa, mesmo que leve, ajuda. Portanto quero começar por agradecer aos meus pais não só por me perguntarem regularmente como vão as coisas e que progresso fiz, mas também pelo interesse e todo o apoio que me deram desde sempre. Agradeço também ao meu irmão, tia, madrinha, avós, tios e primos que constituem a minha família. Não se escolhe onde se nasce, portanto tenho muita sorte em estar nesta.

Agradeço obviamente à outra família (essa sim pode-se escolher) por todos os momentos excelentes ao longo da vida, mas principalmente nos cinco anos de curso em Aveiro. Nunca imaginei vir a ter amizades assim e deste calibre, naturalmente estou-lhes grato (pelas palavras até parece um "obrigado e adeus", mas não é, não planeio deixar estas amizades caírem) por poder estar com eles, construir histórias, aventuras, e partilhar conversas, interesses e tempo. Foi, e é, um prazer dar-lhes o meu tempo.

Agradeço ao Instituto de Telecomunicações pelas condições de acolhimento, não só durante esta dissertação, mas também pelas bolsas de investigação e projetos em que me permitiu participar. Obrigado à professora Susana Sargento, Miguel Luís e Carlos Senna pelo profissionalismo, apoio, paciência, tempo e boa disposição que sempre mostraram. Agradecer ao pessoal do NAP pela disponibilidade, bom ambiente e lanches cinco estrelas. Mas mais do que isso, pela recetividade e amizade demonstradas. Acho excelente poder estar envolvido em projetos reais de investigação enquanto estudante, fora das cadeiras e contexto académico. Talvez um dia todas as áreas e universidades possam ter algo assim.

Agradeço à Veniam pela oportunidade de fazer estágio de dissertação junto deles. Obrigado Diogo, Ana, equipa do vOS e todos na Veniam, não só pela oportunidade que me deram, mas também pela maneira como fui recebido, mesmo tendo ficado pouco tempo. Espero estar sempre em sítios onde possa falar praticamente da mesma maneira com qualquer um, independentemente de cargo ou responsabilidades.

Finalmente, agradecer ao Dirk McCormick, Steven Allen e David Dias da equipa do IPFS, pelo apoio e disponibilidade em me ajudarem a perceber melhor o IPFS e o porquê das coisas serem como são.

Obrigado a todos, vemo-nos nas páginas seguintes.

Resumo

Nos últimos anos, as redes veiculares (VANETs) têm sido o foco de grandes avanços devido ao interesse em veículos autónomos e em distribuir conteúdos, não só entre veículos mas também para a "nuvem" (Cloud). Tipicamente, fazer um download/upload de/para um veículo exige a utilização de uma ligação celular (SIM), mas os custos associados a fazer transferências com dados móveis em centenas ou milhares de veículos rapidamente se tornam proibitivos. Uma VANET permite que estes custos sejam consideravelmente inferiores - mantendo o mesmo volume de dados - pois é fortemente baseada na comunicação entre veículos (nós da rede) e a infraestrutura.
O InterPlanetary File System (IPFS - "sistema de ficheiros interplanetário") é um protocolo de armazenamento e distribuição de conteúdos, onde a informação é endereçada pelo conteúdo, em vez da sua localização. Foi criado em 2014 e tem como objetivo ligar todos os dispositivos de computação num só sistema de ficheiros, comparável a um *swarm* BitTorrent a trocar objetos Git. Já foi testado e usado em redes com fios, mas nunca num ambiente onde os nós têm conetividade intermitente, tal como numa VANET. Este trabalho tem como foco perceber o IPFS, como/se pode ser aplicado ao contexto de rede veicular e compará-lo a outros protocolos de distribuição de conteúdos.
Numa primeira fase o IPFS foi testado numa pequena rede controlada, de forma a perceber a sua aplicabilidade às VANETs, e resolver os seus primeiros problemas como os tempos elevados de descoberta de vizinhos e o fraco desempenho de hashing.
De modo a poder comparar o IPFS com outros protocolos (tais como a solução proprietária da Veniam ou o BitTorrent) de forma relevante e em grande escala, foi criada uma plataforma de emulação. Os testes neste emulador foram efetuados usando registos de mobilidade e conetividade veicular de alturas diferentes de um dia, com um número variável de ficheiros e tamanhos de ficheiros. Os resultados destes testes mostram que o IPFS está a par do protocolo V2V da Veniam (desenvolvido especificamente para V2V e VANETs), e que o IPFS é significativamente melhor que o BitTorrent no que toca ao tempo de descoberta de vizinhos e transferência de informação.
Uma análise do desempenho do IPFS em cenário real também foi efetuada, usando um pequeno conjunto de nós da rede veicular da STCP no Porto, com o apoio da Veniam. Os resultados destes testes demonstram que o IPFS pode ser usado como protocolo de disseminação de conteúdos numa VANET, mostrando-se adequado a uma topologia constantemente sob alteração, e alcançando débitos até 2.8 MB/s, valores parecidos ou nalguns casos superiores aos do protocolo proprietário da Veniam.

Abstract

Over the last few years, vehicular ad-hoc networks (VANETs) have been the focus of great progress due to the interest in autonomous vehicles and in distributing content not only between vehicles, but also to the Cloud. Performing a download/upload to/from a vehicle typically requires the existence of a cellular connection, but the costs associated with mobile data transfers in hundreds or thousands of vehicles quickly become prohibitive. A VANET allows the costs to be several orders of magnitude lower - while keeping the same large volumes of data - because it is strongly based in the communication between vehicles (nodes of the network) and the infrastructure.

The InterPlanetary File System (IPFS) is a protocol for storing and distributing content, where information is addressed by its content, instead of its location. It was created in 2014 and it seeks to connect all computing devices with the same system of files, comparable to a BitTorrent swarm exchanging Git objects. It has been tested and deployed in wired networks, but never in an environment where nodes have intermittent connectivity, such as a VANET. This work focuses on understanding IPFS, how/if it can be applied to the vehicular network context, and comparing it with other content distribution protocols.

In this dissertation, IPFS has been tested in a small and controlled network to understand its working applicability to VANETs. Issues such as neighbor discoverability times and poor hashing performance have been addressed.

To compare IPFS with other protocols (such as Veniam's proprietary solution or BitTorrent) in a relevant way and in a large scale, an emulation platform was created. The tests in this emulator were performed in different times of the day, with a variable number of files and file sizes. Emulated results show that IPFS is on par with Veniam's custom V2V protocol built specifically for V2V, and greatly outperforms BitTorrent regarding neighbor discoverability and data transfers.

An analysis of IPFS' performance in a real scenario was also conducted, using a subset of STCP's vehicular network in Oporto, with the support of Veniam. Results from these tests show that IPFS can be used as a content dissemination protocol, showing it is up to the challenge provided by a constantly changing network topology, and achieving throughputs up to 2.8 MB/s, values similar or in some cases even better than Veniam's proprietary solution.

Contents

Contents i

List of Figures iii

List of Tables v

1 Introduction 1
- 1.1 Content Distribution 1
- 1.2 IPFS 2
- 1.3 Real VANET 3
- 1.4 Objectives and Contributions 3
- 1.5 Structure 4

2 Related work 5
- 2.1 Content Distribution 5
  - 2.1.1 Single server (direct download) 5
  - 2.1.2 Content delivery network (CDN) 6
  - 2.1.3 Peer-to-peer 7
  - 2.1.4 Distributed Hash Table (DHT) 9
- 2.2 Vehicular Ad Hoc Network (VANET) 11
  - 2.2.1 Architectures 12
  - 2.2.2 Communication technologies 12
  - 2.2.3 Features and limitations 13
  - 2.2.4 Challenges 14
  - 2.2.5 VANET evaluation environments 15
- 2.3 VANETs and P2P 19
  - 2.3.1 DHT 22
  - 2.3.2 Network Coding 22
- 2.4 Security 26
- 2.5 Final notes 27

3 InterPlanetary File System 29
- 3.1 Introduction to IPFS 29
- 3.2 Components 30
  - 3.2.1 Identities 30
  - 3.2.2 Network 30
  - 3.2.3 Routing 31

3.2.4 Exchange . . . 31
3.2.5 Objects - *Merkle* DAG . . . 32
3.2.6 Files . . . 33
3.2.7 Naming . . . 33
3.2.8 IPFS Usages . . . 34
3.3 Final remarks . . . 35

4 IPFS Extensions to the Vehicular Network **37**
4.1 Hardware . . . 37
4.2 Initial testing . . . 37
4.2.1 IPFS configurations and notes . . . 38
4.2.2 Test specification . . . 39
4.2.3 Connectivity . . . 40
4.2.4 Data Storage . . . 40
4.2.5 CPU . . . 40
4.2.6 Transport Encryption . . . 43
4.2.7 Hashing . . . 43
4.2.8 Connection Persistency . . . 44
4.3 Final remarks . . . 45

5 Network Mobility Emulator **47**
5.1 Motivation . . . 47
5.2 EmuCD - a new emulator . . . 48
5.2.1 Architecture . . . 49
5.2.2 Implementation . . . 50
5.2.3 How to test a new protocol . . . 54
5.2.4 Kernel parameters . . . 54
5.3 Emulator performance . . . 56
5.4 Summary and Conclusions . . . 60

6 Results - Emulated and Real **63**
6.1 Emulated results . . . 63
6.1.1 Time of day . . . 64
6.1.2 Duration . . . 65
6.1.3 File size . . . 67
6.1.4 Chunk size (IPFS) . . . 68
6.2 Emulator vs VANET . . . 69
6.3 Real Testbed Results . . . 72
6.3.1 Setup . . . 73
6.3.2 Results . . . 74
6.4 Final remarks . . . 77

7 Conclusions and future work **79**
7.1 Conclusions . . . 79
7.2 Future Work . . . 81

Bibliography **83**

List of Figures

2.1 Left - Single server distribution; Right - CDN CD (adapted from[1]). 6
2.2 Illustration of a peer-to-peer network (adapted from [2]). 7
2.3 Example of a decoded torrent file with a single file. 8
2.4 SUMO's interface, taken from SUMO's official website.[3] 16
2.5 Typical communication of ParkCast (taken from [1]). 21
2.6 Representation of the usage of network coding (adapted from [2]). 23

3.1 Example of the peer selection process in IPFS. Adapted from a Bitswap presentation, September 2019, by Dirk McCormick of Protocol Labs. 33
3.2 Beginning of a block request (CID1) using IPFS. Adapted from a Bitswap presentation, September 2019, by Dirk McCormick of Protocol Labs. 34

4.1 NetriderV4. 38
4.2 Profile of Node TX during a test using IPFS. 41
4.3 Profile of Node RX during a test using IPFS. 42
4.4 Comparison between hashing algorithms, taken from Blake2's website.[4] 44

5.1 Architecture of the EmuCD Emulator. 51
5.2 Frontend of the network emulator running a scenario with 200 nodes in Oporto. 52
5.3 CPU usage of EmuCD without any protocol, 100 nodes. 57
5.4 RAM usage of EmuCD without any protocol, 100 nodes. 58
5.5 CPU usage of EmuCD running BitTorrent on 100 nodes, 50MB file. 59
5.6 RAM usage of EmuCD running BitTorrent on 100 nodes, 50MB file. 60
5.7 CPU usage of BitTorrent during a stress test, 100 nodes, 50MB file. 61
5.8 RAM usage of BitTorrent during a stress test, 100 nodes, 50MB file. 61
5.9 CPU usage of IPFS during a stress test, 70 nodes, 50MB file. 62
5.10 RAM usage of IPFS during a stress test, 70 nodes, 50MB file. 62

6.1 Comparison between 9h-10h and 15h-16h, using IPFS in 200 nodes, 50MB file. 64
6.2 Comparison between 200 nodes, 50MB file, 9h to 10h against 15h to 16h, using IPFS, Sprinkler and BitTorrent. 65
6.3 Comparison between 200 nodes, 50MB file, 9h to 10h against 15h to 16h, using IPFS, Sprinkler and BitTorrent (zoomed in the first 20 minutes). 65
6.4 Comparison between 200 nodes, 50MB file, from 4h to 5h, using IPFS, Sprinkler and BitTorrent. 66
6.5 Comparison between 200 nodes, 50MB file, from 4h to 11h, using IPFS, Sprinkler and BitTorrent. 67

6.6 Comparison between 200 nodes, 5MB file, from 15h to 16h, using IPFS, Sprinkler and BitTorrent. 68
6.7 Comparison between 200 nodes, 50MB file, from 15h to 16h, using IPFS, Sprinkler and BitTorrent. 68
6.8 Comparison between several chunk sizes when using IPFS from 15h to 16h in a 200 node network, 50MB file. 69
6.9 Comparison between the default, two best and two worst chunk sizes when using IPFS from 15h to 16h (only first 20 minutes) in a 200 node network, 50MB file. 70
6.10 Scenario 1 - Comparison of download times of a 50MB file dissemination in a 5 node VANET using IPFS vs EmuCD using IPFS. 71
6.11 Scenario 2 - Comparison of download times of a 50MB file dissemination in a 5 node VANET using IPFS vs EmuCD using IPFS. 72
6.12 Scenario 3 - Comparison of download times of a 50MB file dissemination in a 5 node VANET using IPFS vs EmuCD using IPFS. 73
6.13 Graph of the download times (s) between the first and last block (50 MB file). . 75
6.14 Percentage of bytes received in a download (IPFS in blue vs Sprinkler in red). . 76
6.15 Percentage of IPFS duplicate blocks in a download. 77

List of Tables

2.1 Summary of the discussed content distribution strategies and protocols. 25

4.1 Comparison of write speeds and IPFS download times between different storages. 40
4.2 Times to download a 50MB file using IPFS under varying parameters (seconds) - averages of 4 tests. 45

6.1 Download times (s) between the first and last block (50 MB file). 75
6.2 Download throughput, calculated dividing data received by download time (units in MB/s). 75
6.3 Number of peers in a download. 77

Acronyms

AP	Access-Point
ARP	Address Resolution Protocol
CAP	Content Access Point
CAR	Content Access Router
CD	Content Distribution
CDN	Content Delivery Network
CDVC	Content Delivery solution based on Vehicular Cloud
CFG	Coalition Formation Games
CID	Content ID
CSCF	Cooperative Store-Carry-Forward
CV2X	Cellular V2X
DAG	Directed Acyclic Graph
DHT	Distributed Hash Table
DSRC	Dedicated Short-Range Communications
ICE	Interactive Connectivity Establishment
IPFS	InterPlanetary File System
IPNS	InterPlanetary Naming System
Loop	loop over orderly phases
LRBF	Local Rarest Bundle First
MAC	Medium Access Control
NCTUns	National Chiao Tung University Network Simulator
OBU	On-board Unit
OTA	Over-the-Air
P2P	Peer-To-Peer
RSU	Roadside Unit
RTT	Round Trip Time
SFS	Self-Certified Filesystems
SLNC	Symbol Level Network Coding

SUMO	Simulation of Urban MObility
TLS	Transport Layer Security
TraNS	Traffic and Network Simulation Environment
V2I	Vehicle-To-Vehicle
V2N	Vehicle-To-Network
V2V	Vehicle-To-Vehicle
VANET	Vehicular Ad-Hoc Network
VCS	Version-Control System
VDTN	Vehicular Delay Tolerant Networks
Veins	Vehicles in Network Simulation

Chapter 1

Introduction

Vehicles are becoming more complex year after year. Featuring automated driving or information/entertainment centers, software is not only for computers, servers or smartphones anymore: it powers private cars, trucks, buses and several commercial fleets. Consider, for example, Tesla's vehicles that support Over-the-Air (OTA) software updates[1], remote updates simply requiring an Internet connection to update everything, from driving-related settings to the air conditioning. Other companies such as Aptiv[2] or Waymo[3] also use software extensively in their vehicles and solutions. All of these manufacturers have one problem in common: they need to send data from their vehicles to the Cloud (performance metrics, logs, etc.) and from the Cloud to their vehicles (such as OTA updates), requiring the vehicle to be in the range of a WiFi-based connection (many vehicles do not have a WiFi chip) or to have a cellular interface. The latter is a solution chosen in the majority of cases because of its reliability and availability (the vehicle does not have to be in the range of a wireless router and it works almost everywhere), but its data costs are very high.

Vehicular Ad-Hoc Networks (VANETs) combine the wireless elements of communication technologies, such as WiFi and Dedicated Short-Range Communications (DSRC) - defining standards for wireless Vehicle-To-Vehicle (V2V) and Vehicle-To-Vehicle (V2I) communication - and the availability of cellular networks by creating a mesh network where every vehicle communicates with each other and the Cloud. This network can be used, not only for information and entertainment purposes, but also to improve the daily safety conditions when driving.

Several challenges arise when creating a VANET however, as they present a volatile and highly dynamic environment by nature: as vehicles move in and out of range from each other, connections are short-lived, preventing large amounts of data from being exchanged during a single connection. Vehicular Delay Tolerant Networks (VDTN) overlays can mitigate this issue by making use of relays and store-carry-forward mechanisms to ensure that messages can be delivered to the destination without a synchronous end-to-end connection [3].

1.1 Content Distribution

Content distribution (or content delivery), as the name suggests, is the process of delivering content from a sender to a receiver. Content can have many types and sizes: a small 50-byte

[1]https://www.teslarati.com/tesla-over-the-air-update-problems-legacy-auto/
[2]https://www.aptiv.com/solutions/autonomous-mobility
[3]https://waymo.com/

packet, an email, a webpage, a video, etc. Everything that reaches our phones and computers from the Internet can be considered content being distributed to us.

This delivery may involve only one server (sender), one client (receiver) and every router in between (responsible for forwarding the packets in the same way multiple post offices forward letters and packages). It may be more complex and involve different servers depending on the clients and where they are, as it happens in a Content Delivery Network (CDN)[4]. It can even go one step further and replace the notion of *server* and *client* with *peer*, a node in a network (Peer-To-Peer (P2P)) that can act both as a client and as a server, requesting content from multiple peers and sending what it has to those requesting it [5]. This is similar to what happens in a vehicular network, except that the peers are moving and it can be difficult to know each peer's location in any given time.

1.2 IPFS

The InterPlanetary File System (IPFS) project [6] began development in 2014 and seeks to evolve the infrastructure of the Internet and the Web, combining the best ideas from Git[4], BitTorrent[5] and Kademlia [7] to build a hypermedia distribution protocol addressed by content and identities, enabling the creation of distributed applications while simultaneously also being a distributed file system.

IPFS works like a P2P network (decentralized), as its nodes are distributed which also prevents a central point of failure – in a typical network, if the server hosting a specific content goes offline, its data is no longer accessible. A vehicular network can be compared to a P2P network, as the concepts of server and client are replaced with the notion of nodes/peers (some times acting as servers, others as clients) – in a VANET the network's nodes are its vehicles and roadside infrastructure. So, theoretically, a protocol working in a P2P network could also work in a VANET. However, there are two important characteristics of a VANET that can potentially prevent this: intermittent connectivity and node mobility.

Vehicles have to be connected using wireless communication technologies. Unless two vehicles are parked or consistently travelling together, it is very hard to establish a persistent wireless link between them – they may be connected for 30 seconds and only reconnect a hour later. Furthermore, it also becomes very hard to route packets between vehicles as their mobility keeps changing the network's topology - in one moment vehicle A can communicate with vehicle B using another one between them as a *middleman*, and in the next moment they can have eight vehicles between them.

Both of these issues are not present in a typical network: computers and servers do not have their cables constantly being disconnected, and if a server is located in London, it will not be in Amsterdam 30 minutes later (the geographical scale between two cities and between the positions of two vehicles within the same area is different, but it is for the sake of the example).

[4]GIT - https://git-scm.com/

[5]B. Cohen, "The BitTorrent Protocol Specification" - http://www.bittorrent.org/beps/bep_0003.html

1.3 Real VANET

Veniam's[6] VANET in Oporto is one of the largest VANETs globally, consisting of over 400 buses, and it is a live, production environment, allowing any passenger to use free WiFi in every bus through this network. Traditionally, WiFi in a bus is provided through a cellular connection; in Oporto however, requests and packets are routed through the VANET until they reach the infrastructure - over 40 Roadside Units (RSUs) that feature a reliable backhaul connection. The downtown Oporto area holds a density of 4 RSU/km^2, while the remaining metropolitan area has a density of 0.6 RSU/km^2. The deployment of RSUs can be expensive upfront, but the use of an infrastructure reduces daily costs when comparing with using the cellular network.

Under the scope of this dissertation, Veniam provided access to 5 vehicular nodes, used to test IPFS and Sprinkler [8], their own V2V content dissemination protocol.

1.4 Objectives and Contributions

The main goal of this work is the study of IPFS and whether or not it can be used as a content distribution protocol in a VANET; more specifically, if it can be used as a "middleware" to download software updates from the Cloud to the On-board Units (OBUs). Using cellular connections to distribute content to large fleets can quickly become prohibitive, even if the data to be downloaded has a small size. On the other hand, only using RSUs or APs may not be the most efficient strategy (depending on the RSU density), since a vehicle may stay 10 seconds in the range of an RSU and only encounter another RSU 10 hours later. It is why VANETs are ideal, using vehicles to distribute data between each other.

The work developed in this dissertation led to the following contributions:

- analysis and explanation of IPFS and its components, focusing on its data exchange protocol Bitswap;
- vehicular network emulator for content distribution, allowing IPFS (and other protocols) to be tested under controlled conditions;
- adaptations to IPFS in order to better suit it for running in a real VANET;
- conclusions derived from the results of running IPFS in an emulated VANET;
- validation and verification of the feasibility of using IPFS as a content distribution protocol in a real VANET.

Part of the work developed in this dissertation, namely the vehicular network emulator for content distribution, was submitted to the Future Internet journal under the "Delay-Tolerant Networking" special issue[7], accepted and is already published [9]. A second paper on the adaptations of IPFS to vehicular networks, its comparison with other protocols and performance is being prepared.

[6]Veniam, The Internet of Moving Things - https://veniam.com/
[7]https://doi.org/10.3390/fi12120234

1.5 Structure

The remainder of this document is structured as follows:

- **Chapter 2** - *Related work* - This chapter discusses relevant concepts, related work and the state of the art in the topics of *Content Distribution*, *Vehicular Ad Hoc Networks*, *P2P* and its connection with vehicular ad hoc networks, and finally a brief mention of some security aspects in VANETs.
- **Chapter 3** - *IPFS* - This chapter describes in detail the InterPlanetary File System (IPFS) protocol and its constituting components, focusing on data exchange and also on network aspects, routing, files and naming. In the end some potential use-cases for IPFS and examples of real-world applicability are enumerated.
- **Chapter 4** - *IPFS Adapted to the Vehicular Network* - This chapter explains the adaptations made to IPFS in order to better suit the protocol for running in a vehicular network. It describes the lab testing conditions and hardware, and why these adaptations are needed in a vehicular network.
- **Chapter 5** - *Network Mobility Emulator* - This chapter is about the developed emulator: the motivation for it and why the existing ones were not enough, its architecture, performance, hardware requirements, and additional useful information.
- **Chapter 6** - *Results - Emulated and Real* - This chapter discusses the results of running IPFS and other related protocols, not only in the emulator, but also in a real VANET. It is also presented a validation of the emulator by comparing it against a real VANET.
- **Chapter 7** - *Conclusion and Future Work* - This chapter presents the most relevant conclusions to be taken from each chapter, followed by what could be improved in future work.

Chapter 2

Related work

This chapter explains the main concepts important to understand what came before IPFS and what it is trying to improve upon. It begins by explaining what Content Distribution (CD) is and some examples in a *regular* context (without nodes disappearing or becoming unreachable, as they do in a vehicular network). Then, the fundamentals about CD in VANETs and peer to peer networks (P2P) are also detailed. Because several protocols and strategies analyzed in this chapter use network coding, some insights into it are also presented.

In order to better compare different protocols, a new emulator was developed, presented in Chapter 5. This chapter also discusses the most popular existing VANET evaluation environments, contextualizing existing platforms and why they were not enough and a new one was developed.

Finally, security in VANETs is briefly addressed, presenting some examples of how this topic can influence a network's behaviour and some of the researched solutions.

2.1 Content Distribution

Everything that reaches our phones and computers from the Internet can be considered content being distributed to us. Whether a user is requesting a web page to check the daily news, or wants to watch a video, stream a movie or an audio track, everything is content being requested from platforms and services, whose goal is to distribute it as quickly and efficiently as possible.

2.1.1 Single server (direct download)

The most common method is to directly download the required content from a server. If for example a user wants to read the news on *website.com*, he types the website's domain on a web browser and waits for the page to load. From the moment he presses *Enter* to the page appearing on the screen, the domain name had to be converted (resolved) to an IP address using a DNS server; then the request or requests were placed into IP packets destined to the server's address; the packets are then routed between machines until they reach their destination. Once the server receives them, it replies with packets of its own, containing the components of the page in *website.com*: HTML, CSS, Javascript, images and any other required file to render the page.

2.1.2 Content delivery network (CDN)

As a service grows its userbase, not only in numbers but also geographically, it needs to have multiple servers, as a single machine cannot handle every simultaneous incoming request. It becomes troublesome not only due to the service being slow, but it can be even worse if a user is in a different continent than the server: the Round Trip Time (RTT) of a packet going from Amsterdam to Paris is 10ms, but from Amsterdam to Auckland the RTT is 290ms[1].

A CDN overcomes this limitation: instead of a user in Australia accessing the contents in a server in Amsterdam, it accesses a server in Australia that is a replica of the European server. In a CDN, content is replicated in cache/edge servers scattered over the globe. Client requests are redirected to the most suitable location based on factors such as proximity, server load and response times.

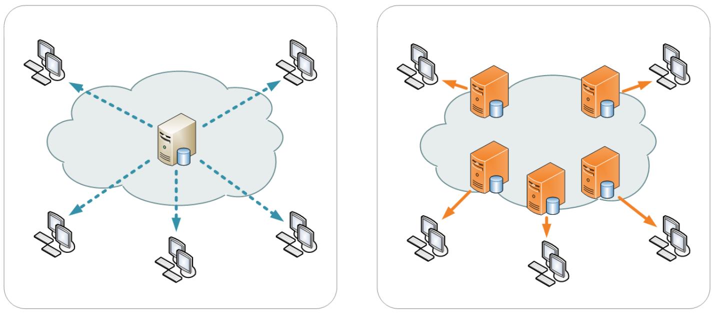


Figure 2.1: Left - Single server distribution; Right - CDN CD (adapted from[2]).

Traditionally this means that a company/service needs to deploy and maintain its own infrastructure at a global scale, which can be hard, not only logistically, but also financially. With the emergence of commercial CDN providers and the advances in Cloud technology and virtualization, it is now easier than ever to have a service distributed worldwide using a CDN, reducing the effort and cost of building dedicated solutions for a particular application and use-case.

One of these commercial CDN providers is Akamai[3], tested by Triukose et al. [10] on whether it actually improves web performance. In order to provide precise results, the download performance from the CDN is compared with the performance of a direct download from the origin server. It was shown that CDN delivery outperforms both non-cached downloads (downloading from a CDN server that does not have the content) and origin (downloading from the original server) delivery in 98% and 96% of cases respectively. In fact, in 67% and 41% of the cases, respectively, the CDN delivery is at least 5 times faster. These results were obtained using measuring points (servers) that are well connected to the Internet and do not experience major bottlenecks. For residential measuring points, the benefits drop significantly, with CDN delivery being at least 5 times faster than origin in only 2.3% of the cases (but still outperforming non-cached and origin downloads in 95.5% and 91% of the cases).

[1]Global ping statistics - wondernetwork.com/pings
[2]Wikipedia's CDN page - en.wikipedia.org/wiki/Content_delivery_network
[3]https://www.akamai.com/

2.1.3 Peer-to-peer

The previous subsections mentioned how content can be retrieved either from a single server, wherever it may be or, in the case of a CDN, from the most adequate server. There is another type of CD, where the content does not come from a fixed server. Instead, the notion of a typical server is "discarded" in favor of the concept of *peer*. In P2P networks, a peer/node can act both as a client and as a server, in the sense that it requests content from other peers, or sends what it has to those requesting it (seeding).

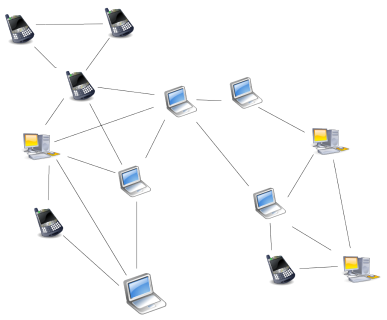


Figure 2.2: Illustration of a peer-to-peer network (adapted from [4]).

P2P was popularized in 1999 with the development of Napster, a peer-to-peer file sharing software[5]. There were already existing P2P softwares, but Napster focused on MP3 files and allowed users to easily share them with each other. It had 80 million registered users at its peak. P2P applications range from file sharing to software distribution and cryptocurrencies. Microsoft uses P2P in a Windows' service called Delivery Optimization[6], which allows a Windows machine to download updates or Microsoft Store Apps from sources other than Microsoft, namely PCs in the local network or PCs on the Internet that are downloading the same files. Using Delivery Optimization shows a 30% to 50% reduction in Internet bandwidth usage required to keep multiple machines on the same LAN updated[7].

BitTorrent

After Napster, multiple P2P protocols appeared, the most popular one being BitTorrent, first released in 2001 (in 2013 the number of nodes in BitTorrent's Distributed Hash Tables (DHTs)[11] varied between 15 and 27 million per day [12]). To send or receive files, a user needs a BitTorrent client (a software that implements the BitTorrent protocol), an Internet connection and a torrent file.

According to the BitTorrent protocol specification[8], a file distribution consists of six entities:

- an ordinary web server such as Apache or Nginx;

[4]https://en.wikipedia.org/wiki/Peer-to-peer

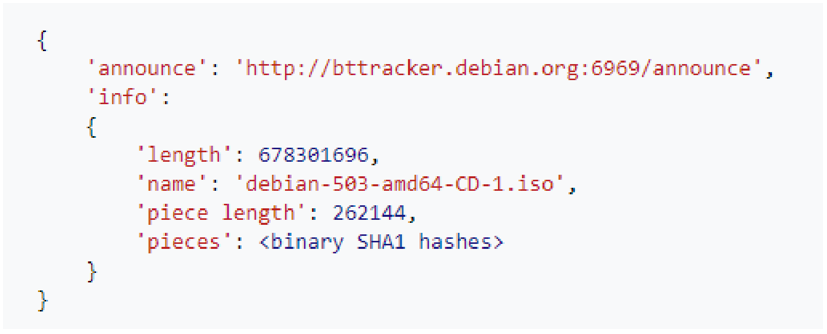
[5]Napster - en.wikipedia.org/wiki/Napster

[6]Microsoft Delivery Optimization - support.microsoft.com/en-us/help/4468254/windows-update-delivery-optimization-faq

[7]Microsoft Delivery Optimization improvements - channel9.msdn.com/Events/Ignite/Microsoft-Ignite-Orlando-2017/BRK2048

[8]www.bittorrent.org/beps/bep_0003.html

- a BitTorrent tracker - assists the communication between peers by keeping track of where file copies reside on peer machines, which ones are available at the time of the client request, and improves the speed of peer discovery;
- a metainfo/torrent file - an encoded dictionary containing the URL of the tracker, suggested file name, file size, piece length and a list of pieces (chunks of a file, represented as 160 bit SHA-1 hashes), as represented in Figure 2.3. A single torrent file can have multiple trackers and multiple files;
- an original seeder - a node with the complete file, the first peer;
- a web browser for the end user to search and/or download the torrent file;
- end user downloader - the client software that implements the protocol and manages the download.



```
{
    'announce': 'http://bttracker.debian.org:6969/announce',
    'info':
    {
        'length': 678301696,
        'name': 'debian-503-amd64-CD-1.iso',
        'piece length': 262144,
        'pieces': <binary SHA1 hashes>
    }
}
```

Figure 2.3: Example of a decoded torrent file with a single file.

In BitTorrent, peer connections are symmetrical and over TCP: data can flow in either direction and messages sent in both directions look the same. On each end of a connection there are two states: one to signal that a connection is choked (no data is sent until unchoking happens) and one to signal interest. Data transfer takes place whenever one side is interested and the other side is not choking. The interest state must be updated constantly. Connections start choked and not interested.

After a connection is opened, several different messages can be exchanged: `have`, `request`, `cancel` and `piece` messages. A `have` message consists of a single number: the index that the downloader just completed and checked the hash of. A `request` message has an index, begin and length fields, and it is sent to request a piece from a peer. A `cancel` message has the same fields as the `request` and is used to cancel a piece download. `Piece` messages contain an index, begin byte and the piece itself (data).

Pieces are downloaded in random order, which is generally enough to maintain a balanced distribution of pieces in the peers. Regarding choking, it is done not only because of TCP's congestion control when sending over many connections at once, but also to ensure that each peers gets a consistent download rate. The deployed choking algorithm only changes who is choked once every 10 seconds, unchokes the four peers who are interested and with the best download rates (if a peer has a better upload rate but is not interested, it is unchoked, and if it later becomes interested, the worst uploader is choked).

BitTorrent clients reward other clients who upload, preferring to send data to clients who contribute more upload bandwidth rather than sending data to clients who upload at a very slow speed. This speeds up download times for the group as a whole, and rewards users who contribute with more upload bandwidth.

The first release of the BitTorrent client did not have a search engine nor peer exchange, so users who wanted to upload a file had to create a small torrent descriptor file that they would upload to a torrent index site. The first uploader acted as a seed, and downloaders would initially connect as peers. Those who wished to download the file would download the torrent which their client would use to connect to a tracker which had a list of the IP addresses of other seeds and peers in the swarm. Once a peer completed a download of the complete file, it could in turn work as a seed (later versions changed this behaviour).

However, in 2005, support for DHTs was introduced, which will be addressed in the following section.

2.1.4 Distributed Hash Table (DHT)

Due to its presence in P2P protocols and applications, namely IPFS and BitTorrent, it should be explained what a DHT is. As the name suggests, it is similar to a Hash Table, enabling the mapping of values to keys, but stored in a distributed way. Tipically, DHTs are autonomous, distributed, decentralized (the nodes form the system without the need for central servers or coordination), fault tolerant and redundant (the system should work and be efficient, even with nodes constantly joining, failing and leaving; a change in the participants must have a minimal impact).

For a DHT to work, it needs keys (unique values usually generated using a hash function); each node is assigned a single key, an ID (hash of a unique value). A node owns all the keys for which its ID is the closest. The "closeness" between two keys is measured by a function that returns a distance value; this can be an abstract notion that is not related to network latency or physical location.

However, for the nodes to actually connect with each other, knowing the keys of the closest nodes is not enough: they need their IP addresses. For this purpose, an overlay network is established, with each node keeping a routing table of its closest nodes. The selection of which nodes should be placed in the routing table is different between DHT protocols, but the essential is that for any key, each node has either a node ID that has that key, or a link to a node whose ID is closer to it.

DHT in BitTorrent

BitTorrent uses a DHT for storing peer contact information of torrents that do not use trackers (server that helps nodes connect to each other), and is implemented over UDP (it should be reminded that the BitTorrent protocol itself uses TCP)[9]. It is called Mainline DHT and is based on Kademlia [7], a DHT created in 2002, but throughout this subsection it will be referred to as *BitTorrent's DHT*.

It should be noted that, in this subsection, when talking about BitTorrent and its DHT, a *node* is a client/server listening on an UDP port implementing the DHT protocol; a *peer* is a client/server listening on a TCP port implementing the BitTorrent protocol. They usually are the same physical machine, as BitTorrent clients include a DHT node.[10]

[9]www.bittorrent.org/beps/bep_0005.html
[10]See footnote 9.

In BitTorrent's DHT protocol, each node has a unique ID, chosen at random from SHA-1's 160-bit key space (20 bytes). The distance function is a XOR between two keys, with the result being interpreted as an unsigned integer (smaller values are closer).

Nodes must maintain a routing table containing the contact information for a small number of other nodes. Nodes know about many others in the DHT that have IDs close to their own, but only know about few that are very far away from their own (always regarding the distance metric given by the distance function).

When a node needs to find peers for a torrent, it uses the distance metric to compare the info-hash of the torrent with the IDs of the nodes in its routing table, contacting the nodes with IDs closest to the torrent's hash. If a contacted node knows about peers for the torrent, the peer contact information is returned in the response; otherwise, the response is the contact information of nodes in its routing table that are closest to the target. The original node then contacts these new nodes, and the search continues.

When querying for peers, the responses also contain a token value (SHA1 hash of the IP concatenated with a secret), in order to prevent malicious hosts from signing up other hosts for torrents. When a node announces that its peer is downloading a torrent, the nodes targeted by the announcement check the token against its IP address. The secret changes every 5 minutes and a token is valid for 10 minutes.

Regarding the routing table, it is subdivided into "buckets", each covering a portion of the 160-bit key space. A table starts empty, with only one bucket covering the entire range. When a node with ID N is inserted into the table, it is placed in the bucket where N is within the minimum and maximum values. Each bucket can only hold 8 nodes; when the bucket reaches the maximum capacity, it is "full". When a bucket is full of good nodes, where a good node is a node that has responded to a query within the last 15 minutes, becoming bad if it fails to respond to multiple queries in a row, no more nodes can be added, even if it is a good node (if any nodes in the bucket became bad, the new good one will replace one). The only case where the bucket is split into two new buckets is if it is simultaneously full, a new good node is to be added, and the current node's ID is inside that bucket (so 7 nodes + current). After the creation of the new buckets, the previous nodes are split into their corresponding buckets.

Regarding queries, they can be of the following types:

- `ping` - carries the sender's node ID, with the response being the node ID of the responding node;

- `find_node` - used to find the contact information of a node ID; the query has the ID of the querying node and the target ID, whereas the response has the ID of the responder and either the contact information of the target node, or a list of the closest nodes;

- `get_peers` - used to get the peers associated with a torrent's hash (infohash); the query carries the ID of the querying node and the 20-byte hash, whereas the response has the ID of the queried node, a token and a list of strings that can either contain peers (if the queried node has peers for that hash) or nodes (if there are not any peers, the nodes closest to the info-hash in the queried node's routing table);

- `announce_peer` - used to announce that the peer of the querying node is downloading a torrent on a port; contains the node ID, the hash of the torrent, the port as an integer and the token received in response to a previous `get_peers` query.

DHT in IPFS

In IPFS, the DHT is the fundamental component of the content routing system, mapping what the user is looking for to the peer (or peers) that store the matching content. Similarly to BitTorrent, the DHT is also inspired by Kademlia [7]. In this subsection, some IPFS specific terms and ideas are mentioned, namely terms such as *peerID* or InterPlanetary Naming System (IPNS). They will be fully explained in Chapter 3. The reason why IPFS's DHT is mentioned here is to provide an easier comparison with BitTorrent's DHT.

Although the basis of IPFS's DHT are explained [6], it is rudimentary and has suffered a great evolution since 2014, so the official documentation[11] is a good source of information.

IPFS uses SHA256 for hashing, so the keys are 256 bit long instead of BitTorrent's SHA1 160 bit keys. There are three types of key-value pairings in the DHT:

- **Provider records** - maps a multihash (content hash) to a peer that has that content; it is used by IPFS to find content;
- **IPNS records** - maps an IPNS key (hash of a public key) to an IPNS record (signed pointer to a path); it is used by IPNS;
- **Peer records** - maps a peerID to a set of addresses at which the peer may be reached; it is used by IPFS when it wants to reach a peer.

There are two DHTs in IPFS: the publicly shared one that nodes use to discover and advertise content, and a LAN DHT, available to nodes that cannot be part of the public network because they do not have public IP addresses and are not publicly dialable (a node can have a public address but still be unreachable because of firewall restrictions).

Every IPFS node maintains a routing table with links to other peers in the network, whether they are public or only in LAN. When a peer connects to another peer, it can be added if: 1) the peer is a DHT server; and 2) the peer has at least one IP address in the public range. After deciding that the peer is qualified to be added, it is determined how close it is to decide which bucket it should go into.

A bucket is a collection of up to 20 peers that have similar addresses (versus the 8 in BitTorrent). If a bucket is full, IPFS determines if any peers can be dropped (after the 10 minute refresh a peer appears offline or unreachable); otherwise, IPFS does not add the peer to the bucket.

The *closeness* of a peer is calculated using the SHA256 hash of its ID, which is interpreted as an integer between 0 and $2^{256} - 1$, and subtracting it from the SHA256 hash of the peerID (also interpreted as an integer) it is being compared to. The peers are ordered from smallest to largest according to this integer and a skip-list is built, where a peer knows other peers that are distanced of around 1,2,4,8,16,... away from it, keeping up to 20 links for each multiple of 2 away.

2.2 Vehicular Ad Hoc Network (VANET)

A Vehicular Ad Hoc Network (VANET) is an extension of a Mobile Ad Hoc Network, a system of self-organizing and self-configuring nodes that act both as router and host. These nodes can either be stationary or mobile leading to constant network topology changes, and can

[11]https://docs.ipfs.io/concepts/dht

have any mobile structure as a node, from bicycles and cars to boats, robots or airplanes/drones [13].

2.2.1 Architectures

VANETs are comprised by a set of vehicles equipped with OBUs, which can exchange information wirelessly with other vehicles' OBUs. This type of communication is called V2V communication. An OBU has a CPU, memory, storage, wireless interfaces, just like a regular computer – except that it is usually smaller and low powered due to portability and energy constraints.

VANETs can also have RSUs that are placed at fixed positions along roads and highways, providing structure to the network, enhancing network performance and offering sturdier connections for data transmission (constant connection, usually to the Internet, and always on due to constant power delivery). Despite not being strictly necessary to create a VANET, an infrastructure domain greatly expands the versatility of the network and serves both as gateway to the OBUs, enabling V2I communication, and to communicate with other RSUs. An RSU also has a CPU, memory, storage and network interfaces, not only wireless to communicate with vehicles, but also wired (such as Ethernet and fiber optic).

2.2.2 Communication technologies

This subsection presents the communication technologies in use or being considered for use in VANETs.

IEEE 802.11p based standards

IEEE 802.11p [14] is a variation of the IEEE 802.11 family meant to support wireless communication in a vehicular environment covering a range of up to 1000m [15]. The Medium Access Control (MAC) and physical layers are based on IEEE 802.11a. It is heavily promoted by vehicle manufacturing industries across the globe, especially in the USA. Due to substantial production volumes, the estimated deployment cost of IEEE 802.11p is predicated to be relatively low when compared with cellular technology. Hence, this technology also called WAVE (wireless access in vehicular environments) has an edge against cellular in VANETs [16].

The main advantage is its range (100-1000m), low delay (200 μs) and low cost, but it is unable to reach high bandwidth values (up to 27 Mbps). Furthermore, in order to support V2I with IEEE 802.11p, naturally the RSUs/infrastructure must also have IEEE 802.11p wireless interfaces.

WiFi

WiFi IEEE 802.11 b/g/n is the same technology present in our homes, phones and computers, providing us with access to the Internet daily. The most common use-case involves an Access-Point (AP) and a client that wants to establish a connection. The AP is running WiFi in the AP mode, preventing it from connecting to other networks using that same interface and serves only as an access point. The client is running WiFi in standard mode, allowing him to connect to an AP, but preventing him from hosting a network in that interface. There is a third mode, WiFi ad-hoc, that enables the creation of an ad-hoc network, replacing the role of AP and client with a node in a P2P fashion.

Ranges up to 100m have been observed in the field, and because it is IEEE 802.11 b/g/n, it can easily connect to an AP. Besides, the higher bandwidth than IEEE 802.11p (80 Mbps, but slower than normal WiFi), WiFi Ad-Hoc's main advantage is the ability to use existing WiFi access points in the streets. This results in a lower deployment cost versus IEEE 802.11p, which requires additional infrastructure deployment.

Cellular

3G and 4G technologies provide reliable security and wide communication coverage (5G is not at the same widespread usage of the previous two generations). In USA, Europe and Japan, many fleet and telematics projects are already using different generations of cellular technology. However, the high cost per GByte, coupled with its high latency rate and limited bandwidth, discourages its possible use as a future communication base for VANETs [16].

CV2X

Cellular V2X (CV2X) [17, 18] is a new standard supported by 5G systems. The specification is an extension of LTE and it has been released by the 3GPP in Release 14. This standard incorporates both direct communication without using a base station (V2I, V2V) and network communication using a downlink/uplink interface. According to the V2X applications defined by 3GPP, V2V and V2I would be implemented over a PC5 interface operating in ITS 5.9 GHz band, while Vehicle-To-Network (V2N), similar to cellular, operates in a commercial licensed band over Uu interface [17].

In 2019 the European Union selected CV2X for autonomous driving[12], whereas the United States of America, despite not having an established choice, have not rejected DSRC/IEEE 802.11p either.

2.2.3 Features and limitations

According to Nidhal et al. (2014) [19], a VANET's main features are as follow:

- **High mobility** - this is one of the defining features; under normal circumstances nodes move all the time with different speed and directions; in fact, the higher the mobility, the more reduced is the network mesh (fewer routes between nodes);
- **Dynamic topology** - given the high mobility, VANET topology changes rapidly, making it dynamic and unpredictable; the connection times are short (depends on the speed), especially between nodes moving in opposite directions; eventually this difficults the detection of a malfunction or attack on the network;
- **Frequent disconnections** - the dynamic topology and the high mobility of nodes, combined with other conditions such as climate and traffic density, lead to frequent disconnections of vehicles from the network;
- **Availability of the transmission medium** - air is the transmission medium of VANETs, and despite the universal availability of this wireless transmission medium (one of the great advantages of inter-vehicle communication), it becomes an issue not

[12]http://ec.europa.eu/newsroom/dae/document.cfm?doc_id=17131

only if there are too many transmissions in the same area (medium becomes saturated) but also impacts security;

- **Anonymity** - if we leave aside the restrictions and regulations of use, anyone equipped with a wireless transmitter operating in the same frequency band as the VANET may be able to transmit and cause issues;
- **Limited bandwidth** - unlike with a physical Ethernet cable, the bandwidth of wireless communications is very limited, whether the used technology is IEEE 802.11p, WiFi or CV2X (although an Ethernet cable also has a bandwidth limit, it is orders of magnitude higher);
- **Limited transmission power** - transmission power of a wireless technology is limited, not only for health and safety reasons, but also because it is unfeasible to maintain a high powered transmission, which leads to the last point;
- **Energy storage and computing** - because the nodes are mobile, energy must be stored in batteries which have a limited capacity and power output; related to this, real-time processing of large amounts of information is a challenge due to the CPUs that are usually present in a VANET node.

2.2.4 Challenges

Some of the features mentioned previously in Subsection 2.2.3 also constitute a challenge for vehicular networks to handle, as pointed by Al-Sultan et al. (2014) [20] and Kumar et al. (2012) [21]:

- **Network scalability** - a VANET can extend itself endlessly, as long as there is a road with a growing number of vehicles;
- **Dynamic topology** - the nodes in a VANET are constantly moving during most of the time, with different velocities and directions, leading to a dynamic network topology; the high velocity of the vehicles, especially in highway scenarios, largely reduces the connected time between nodes;
- **Intermittent connectivity** - in low density scenarios, whether it is a rural environment or an urban one with a small number of nodes, the topology is highly unpredictable and the time a node is disconnected from any neighbor is higher than the time it spends connected;
- **Security and privacy** - keeping a balance between security, privacy and performance is one of the main challenges of vehicular networks: checking the authenticity of a received information is important to the receiver; however, this verification takes time and might violate the sender's privacy;
- **Heterogeneity of applications** - VANETs should be able to supply a large variety of safety and/or information applications. Road safety applications in an emergency scenario require the lowest delay possible and high priority, whereas information/entertainment applications demand a larger use of resources and are able to support higher delays delivering information. With this in mind, it is important to develop an approach ensuring the coexistence of both types of services.

2.2.5 VANET evaluation environments

Evaluating a new software or protocol in a network can be challenging, depending on its complexity and toplogy. This difficulty is aggravated in a VANET, for several reasons that will be discussed later, which is why there are several simulation tools available, and even some emulators, with most of them being used to evaluate the performance of a network and/or protocols [22, 23]. Some simulators are more specific than others, but they usually provide the most common protocols for testing, and support custom protocols – although these require a specific implementation with the simulator's API and/or language.

In order to understand the need for a new emulator, some of the considered options are presented here. Most of the works previously discussed use one of these evaluation environments.

Simulating vs Emulating

The simulators mentioned below are usually discrete-event simulators, where one or more state-machines coordinate the simulation. A distinguishing feature of a simulator is that it is not required to run in real time. With enough hardware resources, one hour of network activity can be simulated in five minutes, with the results ready to be analysed at the end.

An emulator, on the other hand, reenacts everything as if it were a real VANET: each node has one or more network interfaces, an OS, storage, CPU, RAM and the software running, everything in real time. Another relevant aspect is the difficulty/work required when developing and testing new protocols for content dissemination, or even routing, on VANETs in these simulated environments: most of the existing solutions require the protocol to be integrated with the simulator/emulator, according to their rules and APIs, in a predefined programming language, usually C++. This results in the need for the developer to first become familiar with the simulator/emulator, and only then start working on the development of the protocol. Besides, the integration of real traces with the most common simulators is also a task with limitations and challenges, caused by different approaches (simulation versus emulation) and their influence on the software's architecture [24]. However, one of the objectives of this work is to test content dissemination protocols to later use them in real OBUs and RSUs, and to include real traces of both mobility and connectivity radio propagation, in this case from Oporto's communication infrastructure [25].

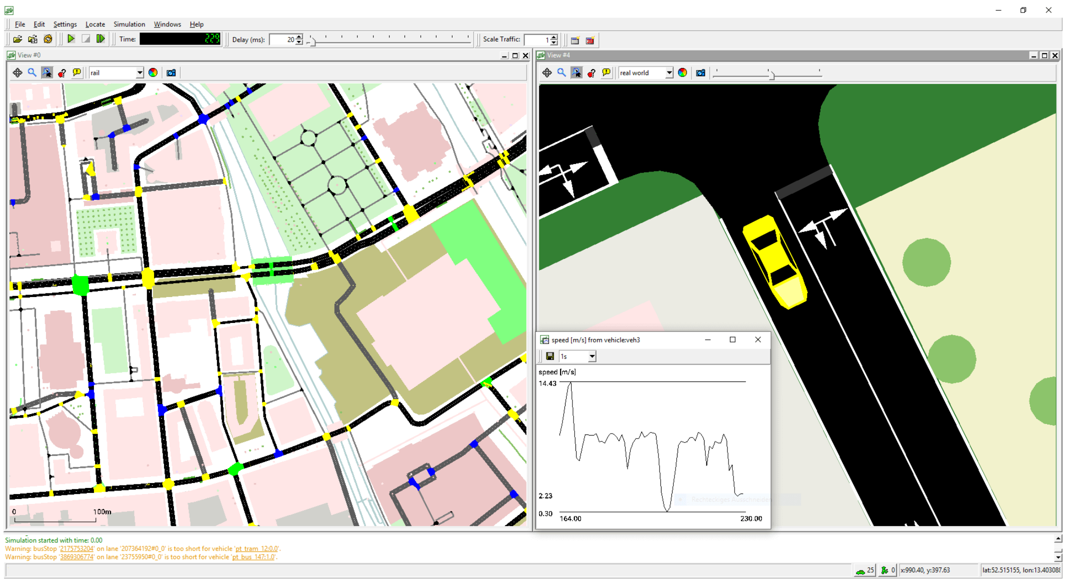
Mobility simulators

Before diving into network simulators and emulators, which focus on the network aspect of the process, mobility simulators should be mentioned, which focus on the mobility aspect of a network. Mobility simulators are based on mobility models employed to simulate vehicles' movement in VANETs. Mobility models can either be collected from a real VANET or simulated. Again, they are not network simulators but can generate a simulated dataset of a VANET (some even take traffic into account), which can then be used in a network simulator/emulator.

SUMO Simulation of Urban MObility (SUMO) [26][13] is an open source traffic simulation package, designed to handle networks with thousands of vehicles. SUMO allows the modelling

[13]https://www.eclipse.org/sumo/

of inter-modal traffic systems including road vehicles, public transport and pedestrians. SUMO includes supporting tools which handle tasks such as route finding, visualization (Figure 2.4 shows the software's interface), network import and emission calculation. SUMO can be enhanced with custom models and provides various APIs to remotely control the simulation.



```
Simulation started with time: 0.00
Warning: busStop '2175753204' on lane '207364192#0_0' is too short for vehicle 'pt_tram_12:0.0'.
Warning: busStop '3869306774' on lane '23755950#0_0' is too short for vehicle 'pt_bus_147:1.0'.
```

Figure 2.4: SUMO's interface, taken from SUMO's official website.[14]

VanetMobiSim VanetMobiSim [27] is a realistic vehicular movement trace generator for network simulators. The traces generated by VanetMobiSim are validated by illustrating how the interaction between featured motion constraints and traffic generator models is able to reproduce typical phenomena of vehicular traffic.

MOVE MOVE [28] is used in VANET simulation to produce realistic mobility models in a rapid manner. MOVE represents an interface with real world databases such as Google Earth. Its output is a trace file which is compatible with network simulators such as Network Simulator (NS).

Network simulators

INET for OMNET++ Quoting from its webpage, OMNeT++[15] is "an extensible, modular, component-based C++ discrete event simulation library and framework, primarily for building network simulators". It is a general purpose simulator, not a network simulator by itself, but when used together with INET[16], it supports wired, wireless and mobile networks.

There are dozens of simulation models and tools available, with the vast majority publicly available on Github; custom protocols can be implemented in C++.

[14]https://www.eclipse.org/sumo/
[15]https://omnetpp.org/intro/
[16]https://inet.omnetpp.org/

One of those tools is OverSim[17], an open-source overlay and peer-to-peer network simulation framework. It contains several models for structured (e.g. Chord [29], Kademlia, Pastry [30]) and unstructured (e.g. GIA [31]) P2P systems and overlay protocols.

NS-2 and NS-3 NS-2[18] is an open source network simulator, supporting the most popular protocols, where-as NS-3[19] is the following version, focused on improving upon the core architecture, software integration, models, and educational components of NS-2.

Focusing on NS-3, it is a discrete event simulator for Internet systems, providing substantial support for simulating of TCP, routing and multicast protocols, over both wired and wireless networks. The programming language C++ is used both for the implementation of the software and to implement the simulation. It is built as a library which may be linked to a custom C++ simulation program and has support for Python.

Riverbed Modeler (previously OPNET) Riverbed Modeler[20] is a tool to simulate the behavior and performance of a network. The main difference of this tool when comparing to other simulators lies in its power and versatility. It allows the creation and simulation of different network topologies, while providing a comprehensive development environment supporting the modelling of communication networks and distributed systems.

The main disadvantage is that the set of protocols and devices is fixed - it does not allow existing protocols to be changed or to add new ones.

ONE ONE [32] is an open source and customizable simulator, allowing for the modelling of node movement, inter-node contacts, routing and message handling. It can be considered as an opportunistic network evaluation system that offers a variety of tools to create complex mobility scenarios when testing DTNs. It supports the implementation of custom protocols.

GrooveNet GrooveNet [33] is a hybrid simulator that allows communication between simulated vehicles, real vehicles and both. It uses a real topography based on a street map to model communication between vehicles. Its architecture incorporates mobility, travel and message transmission models in a variety of link and physical layer communication models. GrooveNet supports multiple network interfaces, GPS and events triggered from the vehicle's on-board computer.

TraNS Traffic and Network Simulation Environment (TraNS) [34] provides a vehicular environment by linking two open-source simulators: a traffic simulator, SUMO, and a network simulator, NS2. Thus, the network simulator can use realistic mobility models and influence the behaviour of the traffic simulator based on the communication between vehicles. The goal of TraNS is to avoid having simulation results that differ significantly from those obtained by real-world experiments.

[17]http://oversim.org/
[18]https://www.nsnam.org/support/faq/ns2-ns3/
[19]https://www.nsnam.org/
[20]https://www.riverbed.com/gb/products/npm/riverbed-modeler.html

NCTUns National Chiao Tung University Network Simulator (NCTUns) [35] is a hybrid simulator and emulator which allows integrated traffic and network simulation. The NCTUns uses this integrated approach to provide a strong feedback between traffic and a network simulator. One of its distinguishable features is the direct use of the TCP/IP protocol stack in the Linux kernel, enabling real-life application programs to be run directly. Quoting the authors directly, "any real-life UNIX application program, either existing or to-be-developed, [thus] can run normally on any node in a simulated network to generate traffic" [36]. Its disadvantages are the required modifications in the simulation machine's kernel and the poor support for mobility datatraces (VANET simulations using NCTUns are either configured using models or through the GUI [37, 38]).

Veins Vehicles in Network Simulation (Veins) [39] is a hybrid simulation framework composed of the general purpose simulator OMNeT++ and the road traffic simulator SUMO. It grants the network simulation capabilities to directly control the road traffic, and thus to simulate the influence of VANET communications on road traffic. Similarly, the road traffic simulation provides information to the network simulation.

Loop loop over orderly phases (Loop) [40] is a trace-based simulator for VANETs which takes vehicles' geographical locations and radio reception events, and creates a synthetic environment to simulate communication protocols on the target VANET. Its motivation was the need for a methodology to evaluate the impact of adding security features to the routing control plane of a VANET. It takes advantage of real data samples collected from an existing VANET, and performs multi-variable evaluations of 24 hour periods in durations ranging from 6 up to 30 minutes. It is an interesting solution developed specifically for VANETs, but its focus is security, not the development of content dissemination protocols.

Network emulators

mOVERS mOVERS [41] is an emulator that is able to recreate scalable vehicular scenarios of data gathering and content dissemination by replicating the same software of the OBUs and integrating with real vehicular mobility and connectivity. It can be used to develop applications that rely on delay tolerant communication using vehicles as information carriers, more specifically routing and content distribution strategies. One of the main advantages is that the source code developed for mOVERS is the same code that runs in the real hardware (OBUs and RSUs), and it has been previously used to test different routing and content dissemination strategies [41, 42, 43], but its main drawback, as most simulators/emulators, is the need for a custom implementation of the protocol within the code (rendering the testing of a closed-source protocol impossible).

The existing solutions require the protocol to be integrated with the simulator/emulator, according to their rules and APIs, in a predefined programming language, usually C++. This results in the need for the developer to first become familiar with the simulator/emulator, and only then start working on the development of the protocol. Besides, the integration of real traces with the most common simulators is also a task with limitations and challenges, caused by different approaches (simulation versus emulation) and their influence on the software's architecture [24]. This is the reason to develop a new emulator based on containers, making the solution completely independent of the protocol being developed. This emulator is later detailed in Chapter 5.

2.3 VANETs and P2P

As previously mentioned, the nodes of a VANET can either be stationary or mobile, usually leading to constant network topology changes. Due to this instability, it becomes a challenge to efficiently route and forward information with classic routing protocols; instead, because the network is a group of nodes (or peers), each with information to send and receive, P2P protocols and its variants are a good fit. P2P networks and VANETs share common challenges and characteristics, which only emphasize the similarities between these two:

- Address problems caused by the topology dynamics;
- Error prone channels;
- User density (scaling up to tens of thousands of nodes);
- Possible non-cooperative nodes.

Over the years, multiple protocols and strategies have been proposed in a continuous quest to achieve lower downloading times, higher fault tolerance and better network connectivity in VANETs. This section presents some of these protocols and strategies. For better understanding of the following strategies, it should be noted that there are two main types of communication between nodes: gossiping and probing. In a gossip protocol, when a node wants to broadcast a message, it selects `t` nodes from the system at random and sends the message to them; upon receiving a message for the first time, each node repeats this procedure [44]. In a probing protocol, beacons/probes are periodically exchanged to better understand available content [8].

Conde et al. (2018) [45] tested an improved version of the Local Rarest Bundle First (LRBF) strategy [43] in a simulator with 24 hours of mobility data from a real vehicular network. In LRBF, a bundle is built containing a selection of the rarest pieces (data packets most important for the majority of the sender node's vicinity), which ensures good results in data distribution throughout the network but results in an increased overhead (excessive size of the control packets in the network). The improvements are regarding the control packets, for which two approaches were proposed: optimization through bands and optimization through bit array. **Optimization through bands** consists of sending out data packets in pairs (or bands), optimizing the control packet's structure and with a performance gain directly proportional to the amount of packets a node receives. **Optimization through bit array** consists of encoding in a bit array information about which data packets have already been received; the size of the bit array is the number of packets in a file; if there is more than one file, there will be more than one bit array to represent each file's information. Because of this, the size of the control packets is fixed, allowing for further optimizations such as compression. The results show that respecting delivery rate, there was not a major difference between the three approaches (LRBF and LRBF with both optimizations). In relation to end-to-end delay, the second metric, there was not much difference either. However, regarding the total size of control packets in the network per hour, the improvements are noticeable. In the first two hours of the experiment, LRBF reached nearly 100MB of network overhead in control packets alone, which is too high as the goal is to disseminate a 75MB file. In the same first two hours, the band approach reached 30MB, and the bit array optimization only reached 8MB of network overhead.

Wang et al.(2017) [46] proposed a Cooperative Store-Carry-Forward (CSCF) scheme to reduce the effect of dark areas caused by a long inter-RSU distance. The scheme uses bidirectional vehicle streams and selects two vehicles in both directions to serve as relays for the target vehicle (carrying the information to the destination). It focuses on the downlink RSU-to-vehicle communication. When a moving vehicle is in range of a RSU and needs to download data from it, the download will occur while the vehicle is inside such range. Before it leaves, the RSU selects a relay vehicle, which stores in buffer the data, and the remaining data is sent to the next RSU through the backhaul connection. The next RSU forwards the received data to a new relay, which will then be sent to the original moving vehicle. The results are theoretical and simulated, showing that when the arrival rate is high, a relatively short time is needed to accumulate enough candidate relays. When the vehicle stream densities in both directions are at the low level, the average outage time is reduced by more than 70% for the smallest dark-area case, and by more than 37% for the large dark-area case.

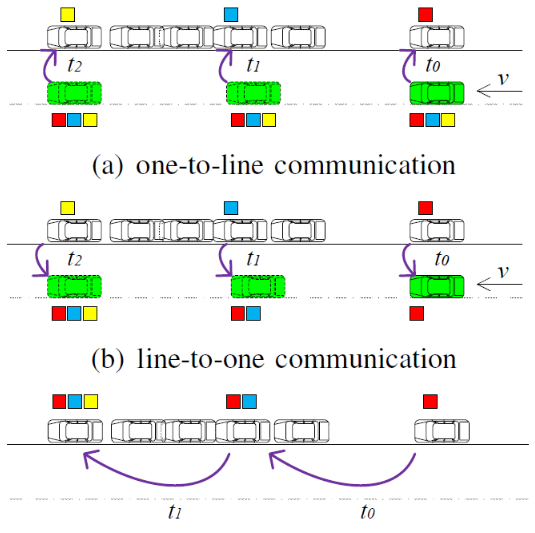
To simplify the topology modelling in a VANET, Huang et al. (2016) [47] propose a cell-based clustering strategy (ECDS) that divides each lane into successive cells of equal size, and treats the OBUs in the same cell as a cluster, simplifying the modelling of the VANET. In order to achieve this, the following work was performed:

- A topology pre-creation and update scheme, based on the cellular clustering;
- Inter-cluster strategies for relay and generation selection;
- Simulations comparing ECDS with Coalition Formation Games (CFG) [48] and CodeOn [49] regarding average downloading percentage and overall finish time.

The cell-based clustering strategy is based on three advantages of treating cells as nodes: 1) vehicles in the same cell are close to each other, usual have the same movement patterns so communications may be interrupted less often; 2) the cell density can be defined and used to estimate the density of an area, helping with the relay and generation section; 3) treating the cells as nodes instead of the OBUs eliminates the changing of the nodes' positions. Simulations were developed using Matlab, focusing on how efficient relay selection and generation selection strategies can speed up the downloading process, and confirming the proposed strategies' efficiency regarding average downloading percentage and overall finish time.

Liu, et al. (2012)[1] proposed ParkCast, an idea that uses roadside parked vehicles to distribute contents in an urban VANET. For each road, parked vehicles are grouped into a line cluster as far as possible, enabling the implementation of longer connection times, sequential file transfers, reducing unnecessary messages and collisions therefore expediting the content distribution. As shown in Figure 2.5, a vehicle-parking contact sequence happens when a moving vehicle is passing by a series of parked vehicles at the roadside.

This design/protocol is based on the assumptions that: 1) the wireless device on a vehicle has a small rechargeable battery to support communications while in parking; 2) vehicles have maps and GPS; and 3) at least 30% of users will be cooperative and share their devices and contents while parked (30% is enough to support the whole system [50]). Content distribution in this scenario can be more predictable and controllable, because of the sequential parking typical at roadside, easing the process of downloading content chunks from a vehicle to parked



(a) one-to-line communication

(b) line-to-one communication

(c) internal communication

Figure 2.5: Typical communication of ParkCast (taken from [1]).

ones, from the parked vehicles to a driving one and between parked vehicles. Contact time increases with the number of parked vehicles, and when this number is large enough, the driving speed of a vehicle does not affect it. The long contact time is guaranteed by extensive roadside parking in urban areas. The parked vehicles are grouped in a line cluster, which has two cluster heads, each at the end of the road. These cluster heads are responsible for periodically reporting their positions, contents, buffer status and are able to manage all parked vehicles and their resources, acting as a local service access point. The only issue occurs when the cluster head suddenly leaves, and a new cluster head is not yet elected. Thus, quasi heads (cluster member next to a cluster head) are used to ensure fault tolerance.

ParkCast shows stable contact opportunities in the simulations and high-quality contacts not affected by traffic changes, being good both in sparse and dense traffic. Compared to SPAWN-like [51] and CodeTorrent-like [52] schemes, ParkCast reduces the average downloading delay almost 100 times in sparse traffic and 20 times in dense traffic, despite not having network coding. In the same time these inter-vehicle schemes download one chunk, ParkCast reaches 100% average downloading rate. In the tests performed, a 100MB file in sparse traffic conditions takes 20min, far less than a typical parking duration, showing that it is attractive for ordinary vehicle users to join ParkCast and share their resources.

Wang et al. (2020) [53] proposed a Content Delivery solution based on Vehicular Cloud (CDVC), where information is identified by content, not location. The vehicular network is made up of the backbone, Content Access Routers (CARs), Content Access Points (CAPs) and vehicles. CARs connect the backbone and CAPs, which communicate with vehicles via a wireless interface. A hierarchy is established between these elements in order to allow content generated by CAPs to be easily accessed by vehicles. A mechanism similar to store-carry-forward is used to deliver content between vehicles. Its name-based mechanism is compared with the address-centric standard, achieving lower content delivery cost and latency. This is an interesting work because, similarly to IPFS, data is referenced by its content, not location/address. However, in CDVC content is identified by name, whereas in IPFS a file is split into multiple blocks, each with its own unique identifier.

Ortega et al. (2020) [54] used IPFS as a foundation (providing content-addressed networking over QUIC and P2P) to a semantic distributed network, with the goal of providing a fast and efficient system to distribute helpful resources such as traffic state or remote sensors in vehicles. It is compared against CDVC and the results of running IPFS on 30 nodes, in a private network simulating real-life V2X conditions, show that IPFS offers a great balance between functionality, performance and security, with latency and cost on par with CDVC (interesting given that CDVC was evaluated in NS-2, so it should have an advantage, whereas IPFS was not).

Up until now, the research works presented in this section have been built and evaluated in a simulated environment. In 2018 however, Recharte et al. [8] presented a single hop co-operative dissemination protocol, and thoroughly evaluated its performance in an operational vehicular network (17 vehicles and 5 roadside units), exploring the role of V2V communication and potential causes of failure. In this protocol, each file is divided into chunks to prepare for transmission, and it was observed that requesting chunks sequentially causes some to become consistently less popular, so chunk selection should be randomized. During the tests, probing overhead was only 1.3% at its maximum, and retransmission overhead (data discarded on connection loss failures) was 2.8% median and 4.2% maximum of the data rate. This protocol's performance was evaluated in Veniam's vehicular network, and is the basis for Sprinkler, Veniam's proprietary V2V content distribution protocol. Sprinkler is one of the protocols compared against IPFS both in Chapter 5 and Chapter 6.

2.3.1 DHT

There have been a few academic works regarding the use of DHTs in VANETs. One of them is the implementation and simulation of a publish/subscribe infrastructure based on a DHT by Mishra et al. (2011) [55]: RSUs were placed in major intersections and formed a DHT based broker overlay, whereas the vehicles could take the role of publisher, subscriber or broker. Vehicles were located without GPS, using broadcasts instead and storing their location in the DHT: each vehicle broadcasts its ID and direction when crossing a RSU, with that RSU hashing the ID to discover which RSU is responsible for keeping that vehicle's location updated. In a simulation scenario with 100 RSUs, a transmission range of 100m and 200m for vehicles and RSUs respectively, and vehicle speeds between 5 km/h and 50km/h, a high density network (5000 vehicles) reached a 80% message delivery rate in 1000 seconds, and the low density network (500 vehicles) reached a 40% message delivery rate in 1000 seconds.

2.3.2 Network Coding

Every protocol, at some point, has problems with slowdowns in download speeds or collisions because, as network traffic increases, the wireless medium becomes saturated. As an attempt to reduce this issue, network coding was created. Network coding was originally proposed by Alhswede et al. (2000) [56]. It is the process under which an intermediate node between source and destination nodes encodes incoming packets into new ones and then forwards them, as opposed to simply forwarding incoming packets. An example is shown in Figure 2.6.

In network coding, data is divided into pieces, with the original K pieces being mixed

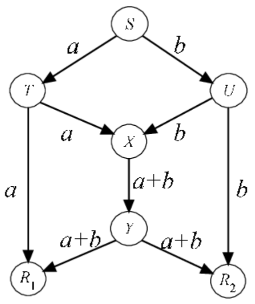


Figure 2.6: Representation of the usage of network coding (adapted from [2]).

together randomly (in random linear network coding), using linear algebra, to create new representations of the data. With this method, an almost infinite number of pieces can be created, and each piece/mixture can be used to reconstruct the data (the K original pieces are not needed, instead any K pieces can be used).

While research shows through simulations that network coding provides efficient and robust distribution, it was believed that it was not practical in real environments because of encoding and decoding overheads, and because protecting against block corruption is difficult [57].

Before network coding was applied to vehicular networks, Gkantsidis et al. (2006) [57] implemented and tested a prototype Network Coding P2P system in the distribution of large files (several GBytes) over the Internet, averaging a decoding time of 6% of the total download time. With respect of resources, during the download period, the CPU overhead was less than 20%, and less than 10% after decoding and while the node was seeding. Furthermore, a constant download performance was observed from the start to the end, as opposed to the slow performance in the beginning of a download in typical P2P systems and near the end, when it becomes harder to find the missing blocks. Since each encoded block is unique and useful to any node, newly arriving nodes can easily obtain useful blocks that they can exchange with other nodes, and nodes at the end of the download do not need to wait before finding their missing blocks.

One of the main content distribution protocols in a VANET is CodeTorrent, by Lee et al. (2006) [52], which investigates the problems of running BitTorrent type P2P file sharing systems (file swarming protocols) in VANETs, and shows that it allows shorter file downloading times compared to existing protocols. When a node (seed node) aims to share a file, it creates and broadcasts to its 1-hop neighbours the description of the file. The file is divided into n pieces, and coded frames are formed – a coded frame is a linear combination of file pieces. To recover n file pieces, a node must collect more than n coded frames carrying encoding vectors that are linearly independent of each other. CodeTorrent is evaluated through simulations: a 1MB file is present in three static APs and a fraction of the nodes are interested in downloading the same 1MB file. This file is divided into 250 4KB pieces, with each piece being transferred using four 1KB packets. A peer must acquire 250 linearly independent coded pieces to decode the file. UDP is used to transfer packets between nodes, relying only on single hop unicast.

Yang and Lou (2011) [49] developed CodeOn, a high-rate, cooperative popular content distribution scheme where contents are actively broadcasted to vehicles from road side access points, and further distributed among vehicles. CodeOn uses Symbol Level Network Coding (SLNC) to combat the lossy wireless transmissions inherent to a VANET. It also actively considers what content each node has: nodes that have useful content broadcast it to neighbouring nodes. CodeOn is compared with CodeTorrent (mentioned above) in simulations, showing that CodeOn outperforms CodeTorrent in every scenario tested (sparse and dense highway, sparse and dense urban), specifically when the average downloaded file percentage is below 90%. It is also more robust to variations in topology and vehicle density, which can be attributed to the push-based design combined with SLNC, providing better reliability and higher rates.

Fairness has also been considered in these protocols. CodeOn's distributions are more concentrated than those of CodeOnBasic (a variation of CodeOn, also push-based, has piece-division but is based on PLNC) and CodeTorrent. This superiority is still attributed to the use of SLNC, since an overhearing node will buffer any innovative clean symbol it received, and therefore, the granularity of information reception is smaller, enabling the vehicles to have similar opportunities to contact with APs and other vehicles. In CodeOnBasic and CodeTorrent, a vehicle either receives a whole piece or nothing, so the variance among reception progresses is larger.

A summary of the discussed strategies and protocols is presented in Table 2.1.

Algorithm/Strategy	**Network Coding**	**Comments**	**Date**
CD based on sparse NC [58]	Yes	Benefited by small block sizes (<512KB), performs worse than non-coding for small content sizes; advantageous when scale grows	2006
Codetorrent [52]	Yes	UDP single hop-unicast transmissions; 4KB pieces	2006
CodeOn [49]	Yes (SLNC)	Scalable push-based CD; performs better than CodeTorrent; fair and concentrated distribution because of SLNC; shows that in a lossy wireless environment, a push based approach is better than typical pull	2011
Mishra et al.'s DHT [55]	No	Publish-subscribe, DHT based VANET, tested in a simulated scenario.	2011
ParkCast [1]	No	Groups parked vehicles in clusters; lower delays than CodeTorrent; much lower cost than AP/RSU installation	2012
IPFS [6]	No	P2P file system; can run on any network protocol; very modular	2014
ECDS [47]	No	Divides lanes into cells, and groups OBUs in clusters inside a cell; the nodes are the cells instead of the OBUs; better performance than CodeOn, specially as the file sizes increase (60% speed increase at most)	2016
CSCF [46]	No	Two vehicles in both directions act as relays and carry information to the target; average outage times are sometimes reduced by 70%, comparing to other CD protocols	2017
Conde et al.'s LRBF [45]	No	Delay tolerant CD with optimized control packets; tested with a dataset from a real network	2018
Sprinkler [8]	No	Single-hop dissemination protocol with probing and a randomized chunk selection; tested in Veniam's vehicular network	2018
CDVC [53]	No	Content addressed VANET, establishing hierarchies and domain zones to facilitate content delivery.	2020
Ortega et al.'s semantics [54]	No	IPFS is used to facilitate the distribution of semantic data as JSON files between vehicles and the infrastructure.	2020

Table 2.1: Summary of the discussed content distribution strategies and protocols.

2.4 Security

Despite their promising features, the communication trust and privacy issues in VANETs should still be carefully resolved before they fully flourish in real lives. This topic is specially valid in the vehicular network scenario because the nodes can be cars, trucks or any moving object, and these pose a large threat in the extreme situation of a vehicle being compromised: it can lead to crashes or theft of unmanned vehicles. In a less extreme situation, for example if the message broadcasted in VANETs is not authenticated, then drivers cannot estimate the traffic situation according to the received messages; if the underlying authentication method reveals the real identity of a vehicle, the location privacy of the vehicle would be disclosed.

Security is a particularly relevant issue in a VANET because, since every V2V link is wireless, it is relatively easy for an intruder in the correct frequency to saturate the medium, cause a denial of service attack, introduce malware, etc., which could render the network unusable.

There have been some efforts to tackle this aspect of a VANET. Dasgupta et al. (2013) [59] proposed TruVal, a trust value mechanism for authentication of vehicles, using a layered framework to assign a trust value to a vehicle: a vehicle must have a desired trust value and be registered under an authority in order to access the network. Typically, authentication is performed using certificates and signatures, but if we instead use aggregate signature mechanisms, computing resources and bandwidth can be saved [60].

Sun et al. (2010) [61] proposed an identity-based cryptosystem where certificates are not needed for authentication, while still satisfying fundamental security requirements including authentication, non repudiation, message integrity and confidentiality. It focuses on the misbehaviour of insider nodes, as it is more difficult to prevent than outside nodes. Their work consists in a: 1) threshold signature-based scheme, 2) pseudonym-based scheme to assure vehicle user privacy and traceability, 3) a privacy-preserving mechanism based on threshold authentication, guaranteeing that any additional authentication beyond the threshold will result in the revocation of misbehaving users.

Although Shabbir, Munazza, et al. (2016) [62] proposed a technique to detect and prevent Distributed Denial of Service attacks in VANETs, the denial of service issue might be the hardest to tackle, since nothing can be done at the physical layer to prevent someone from spamming random packets in the vehicles' wireless frequency, potentially reducing the available bandwidth and ultimately blocking communications.

In an ideal world, security is a relevant aspect that should be taken into account when developing a protocol/product, and placed at the same level of routing or data management. However, it is a topic that is rarely the main focus when developing a working prototype or product, as it is not necessary for it to work. Security in VANETs is still under active research and improvement. This Section mentioned its impacts, leading to privacy issues or extreme threats, and how some researchers have been tackling these issues by developing authentication mechanisms adapted to VANETs [59, 60, 61] or by trying to detect and prevent denial of service attacks [62].

2.5 Final notes

This chapter started with the fundamentals of content distribution, discussing the difference between a direct download and a download using a CDN or a P2P network. A P2P protocol (BitTorrent) was also explained, as a peer-to-peer network has close similarities with a VANET. Distributed hash tables were then presented, along with their corresponding applications and uses in both BitTorrent and IPFS.

With the basis of content distribution established, the focus shifted to VANETs: the typical architecture and technologies used, their defining advantages (such as high mobility, flexibility and wireless communications) and the challenges and limitations they present (frequent disconnections, limited bandwidth, etc.).

Section 2.3 paired the previous concepts of VANETs, P2P and content distribution in order to explore some of the work that has been developed along the years. Table 2.1 presented a summary of the different protocols and strategies discussed, and whether they used network coding or not. Section 2.4 presented a brief insight into the security challenges faced by a VANET.

Finally, Section 2.2.5 presented the existing VANET evaluation environments (simulators, emulators or a combination between both) in order to understand the existing tools and why a new one was developed, EmuCD, later presented in Chapter 5.

Chapter 3

InterPlanetary File System

This chapter explains the InterPlanetary File System (IPFS) [6] and its main components. Different relevance and detail is given to each component. For example, it is given more detail on how blocks are exchanged rather than on how objects are represented (and the filesystem aspect of IPFS), especially considering the previous explanations of how data was exchanged in BitTorrent.

In the particular case of IPFS's DHT, it is not detailed in this chapter. It was explained in the scope of the related work in Chapter 2, allowing for a closer comparison to BitTorrent's DHT.

This chapter closes with some examples of how IPFS has been used, and possible use-cases for future projects and scenarios.

3.1 Introduction to IPFS

IPFS [6] is a peer-to-peer distributed file system to connect all computing devices with the same system of files, similarly to the web. IPFS provides a high throughput content-addressed block storage model, with content-addressed hyper links, forming a data structure upon which one can build a global filesystem, a personal sync folder, a versioned package manager for software – similar to Git[1], a Version-Control System (VCS) — a web CDN, and much more.

IPFS is a distributed file system which combines successful ideas from BitTorrent[2], Git and Self-Certified Filesystems (SFS) [63]. It is peer-to-peer, without privileged nodes, with each node storing IPFS objects locally. Nodes then connect to each other and transfer these objects, which represent files and other data structures. One of the most important ideas that IPFS implements is that what is addressable is the content, not a server or typical URL. Taking the example of a HTML page served at *site.com/index.html*, even if the content remains the same, if the domain changes or the filename changes, that URL address is no longer valid; however, in IPFS it remains reachable. Each file or directory has a Content ID (CID), a SHA256 hash that uniquely identifies the file. The CID only changes if the content changes. However, loosing track of a file because it was altered is not ideal, and for that there is IPNS (InterPlanetary Name System), where the *name* is the hash of a public key, stored in the

[1]GIT - git-scm.com

[2]BitTorrent - http://www.bittorrent.org/beps/bep_0003.html

DHT. This way it is possible to create a single, stable IPNS address that points to the CID for the latest version of a website, for example.

IPFS content can be accessed by other clients that are running the IPFS software, or through gateways using a web browser. A gateway is no more than an IPFS client that is running a web server, so anyone can host one as long as they have a public IP. Taking the example of `ipfs.io/ipfs/QmXyNMhV8bQFp6wzoVpkz3NqDi7Fj72Deg7KphAuew3RYU`: *ipfs.io* is a domain pointing to the IP of a server that, not only hosts the IPFS homepage, but is also a gateway, returning the content in */ipfs/CID*; here the CID is the root of a website that serves free audiobooks, ready to be played in the browser, entirely IPFS/P2P based.

3.2 Components

The IPFS protocol is divided into a stack of sub-protocols or areas, which are discussed in the following subsections:

- *Identities*, managing node identity generation and verification;
- *Network*, managing connections to other peers;
- *Routing*: by default a DHT that maintains information to locate specific peers and objects;
- *Exchange*, a block exchange protocol (BitSwap) that implements efficient block distribution;
- *Objects*: a Merkle DAG of content-addressed objects with links, used to represent data structures;
- *Files*: a versioned file system hierarchy;
- *Naming*: a self-certifying mutable name system.

3.2.1 Identities

Nodes are identified by a `NodeId` (the cryptographic hash of a public-key, SHA256 is used by default). Each node stores its public and private keys. When first connecting, peers exchange public keys and check if the hash of the other node's public key is equal to the other node's ID.

3.2.2 Network

IPFS can use any transport protocol and provide reliability if underlying networks do not provide it. It uses the Interactive Connectivity Establishment (ICE) NAT traversal techniques to assure connectivity, optionally checks the integrity of messages using a hash checksum, and optionally checks authenticity of messages by digitally signing them with the sender's private key. IPFS can use any network, including overlay networks, because it stores addresses as formatted byte strings for the underlying network to use (example of a SCTP over IPV4 connection:`/ip4/10.20.30.40/sctp/1234`). This type of representation is part of *multiformats*[3], a collection of formats that aim to future-proof systems by adding self-described values; because this is an address, more specifically it is a *multiaddr*[4].

The networking library that supports all this process is called **libp2p**. Originally, **libp2p** was the networking layer for IPFS, but given its usefulness and potential for other projects, it was modularized out of IPFS[5]. It currently supports TCP, QUIC, WebRTC, Websockets, uTP and Bluetooth as transport protocols (depends on the platform that is running IPFS), and if a connection is already established and secure, it supports multiplexing over that existing connection (e.g. instead of opening a TCP connection for a DHT query, and another TCP connection for a block exchange, the operations are multiplexed).

Additionally, it uses stream multiplexers such as YAMUX[6] in order to better handle NAT traversal and achieve a more efficient flow control when compared to just using TCP, for example. It operates on top of an existing underlying connection (TCP, Unix sockets, etc.) and provides stream-oriented multiplexing, bi-directional streams (can be opened by either client or server) and enables persistent connections over a load balancer. YAMUX implements the same interface as TCP, so from a software point of view, opening a new stream in YAMUX is the same as opening a new TCP stream without the potential overhead associated (based on top of an already open TCP connection).

In fact, in the case of YAMUX the interface is so similar to TCP that the documentation uses the terms *SYN, RST, ACK, FIN* as if they were TCP messages[7]. When a stream is being opened, a new frame with a new `StreamID+SYN` is sent, with the reply consisting of the received frame and an `ACK/RST`. Because there is already an underlying TCP stream open, data can start flowing in the YAMUX stream before the `ACK` is received. When a stream is to be closed, a node sends `FIN`; when the other node replies with a `FIN` as well, the stream is closed. Alternatively, `RST` can be used to close the stream immediately.

3.2.3 Routing

IPFS (libp2p) uses a DHT based on S/Kademlia [7] and Coral [64] to find other peers' network addresses and peers who can serve particular objects. Already detailed in Section 2.1.4, the DHT stores small values ($<=$ 1KB) directly, and for larger values it stores references, which are the `NodeIds` of peers who can serve the block. IPFS provides a routing interface, allowing the DHT routing system to be swapped for one that fits the users' needs, as long as the interface is met.

3.2.4 Exchange

Data exchange happens by exchanging blocks with peers using a BitTorrent inspired protocol called Bitswap. Bitswap is the underlying manager for network related block operations: IPFS asks for blocks, Bitswap fetches them from the network.

Bitswap has two main responsabilities: to fetch blocks requested by IPFS from the network, and to send blocks in its possession to other peers who want them. When a client requests

[3]https://tools.ietf.org/id/draft-snell-multihash-00.html
[4]https://github.com/multiformats/multiaddr
[5]https://github.com/libp2p/libp2p
[6]https://github.com/libp2p/go-yamux
[7]https://github.com/libp2p/go-yamux/blob/master/spec.md

blocks, Bitswap sends the `CID` of those blocks to its peers as `wants`. When Bitswap receives a '`want`' from a peer, it responds with the corresponding block[8].

Requesting blocks

When IPFS requests a block, Bitswap: 1) sends a `want-have` message with the block CID to all peers in the session to ask who has the block - a session is a mechanism that allows a peer to make related requests to the same group of peers (like a "priority contact" group) -; and 2) broadcasts the `want-have` to all peers it is connected to, asking if they have the block. Peers with the block respond with a `have` message and are added to the session. If no connected peers have the block, Bitswap queries the DHT to find peers that have the block.

Simultaneously, while sending `want-have` messages searching for the block, it actively sends a `want-block` message to one of the peers in the session to request the block. If the peer does not have the block, it responds with a `dont_have` message, in which case Bitswap selects other peers and tries again. The choice of a peer to send the block request is done using a probabilistic algorithm, favouring peers that: 1) have previously sent `have` for the block; 2) were discovered as providers of the block in the DHT; and 3) were the first to send blocks to previous session requests.

Figure 3.1 shows the peer selection process: when requesting multiple `CIDs` (i.e. every block of a file), requests are split across peers, with peers being ordered by latency. The number of `CIDs` to be requested to each peer is calculated using a split factor, which varies according to the unique/duplicate ratio.

In Figure 3.2, peer A asks who has CID1; peer B has CID1 so peer A requests the block from peer B. Meanwhile, peers C and D reply that they have CID1, but because peer A already sent a *want-block* to B, it is B that sends the block.

Whenever a block is received, it checks to see if any peers sent any messages for that block. If so, it sends `have` or the block (depending on the message) to those peers. Bitswap keeps requests from each peer in separate queues, ordered by the priority specified in the request message. In order to select the peer for the next response to be sent to, Bitswap chooses the peer with the least amount of data in its send queue. That way it will tend to keep peers busy by always keeping some data in each peer's send queue[9].

Serving blocks

Upon receiving a `want-have`, Bitswap checks if the block is in the local blockstore. If so, it responds with `have`, and if the block is small enough, it sends the block instead of the message. Otherwise, it checks a `send_dont_have` flag on the request: if true, it sends `dont_have`; otherwise, it does not respond.

3.2.5 Objects - *Merkle* DAG

On top of the DHT and BitSwap, IPFS builds a Merkle Directed Acyclic Graph (DAG) [65], where links between objects are cryptographic hashes of the targets embedded in the sources. It is a generalization of the Git data structure, with useful properties such as content addressing

[8]https://github.com/ipfs/go-bitswap/blob/master/docs/how-bitswap-works.md
[9]See footnote 8.

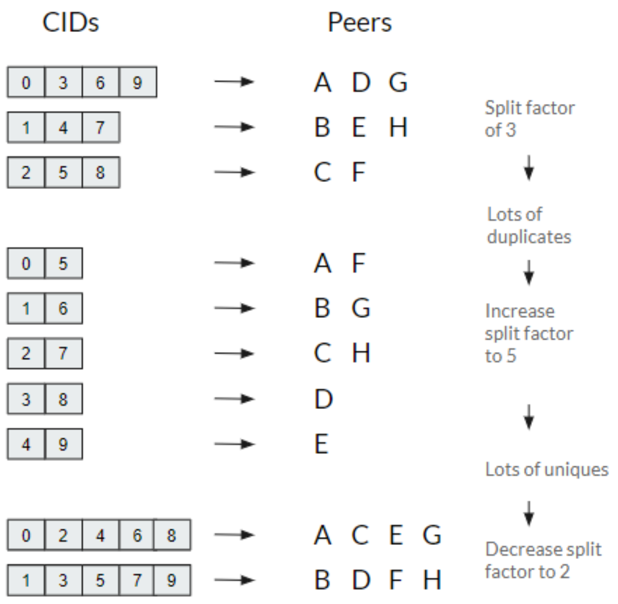


Figure 3.1: Example of the peer selection process in IPFS. Adapted from a Bitswap presentation, September 2019, by Dirk McCormick of Protocol Labs.

---

or tamper resistance, while also preventing deduplication (all objects that hold the exact same content are equal and only stored once).

IPFS objects can be traversed with a string path API, as one does in the traditional UNIX filesystems and the Web. Full paths in IPFS are of the form /ipfs/<hash-of-object>/<name-path-to-object>.

With content-addressing, anyone can publish an object in the DHT, done in a fair, secure and entirely distributed way.

3.2.6 Files

IPFS defines a set of objects to model a versioned filesystem on top of the Merkle DAG, similar to Git:

- Block: a variable-sized block of data (blob) (if a file is larger than the block size, it is split into blocks);
- List: an ordered sequence of blocks or other lists;
- Tree: it represents a directory, a map of names to hashes;
- Commit: it represents a snapshot in the version history of any object. As long as a single commit and all the children objects it references are accessible, all preceding versions are retrievable, and the full history of the filesystem changes can be accessed.

3.2.7 Naming

Every user is assigned a mutable namespace at /ipfs/<NodeId>, with the ability to publish an Object to this path signed by its private key. When other users retrieve the object, they

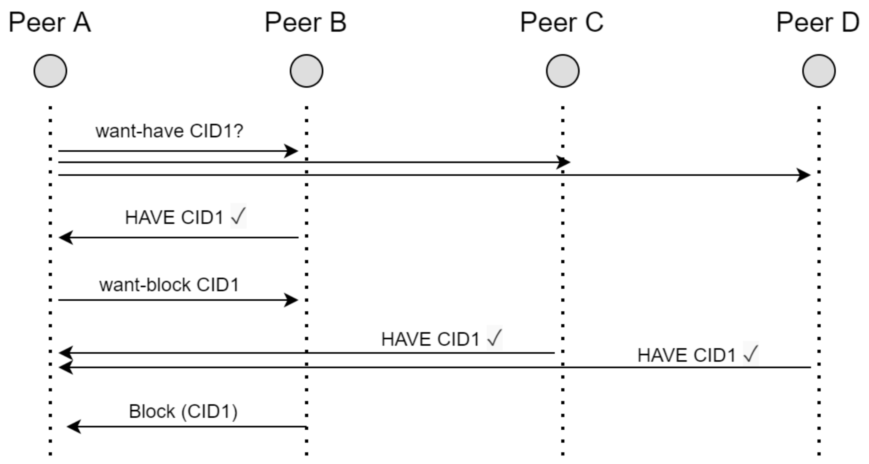


Figure 3.2: Beginning of a block request (CID1) using IPFS. Adapted from a Bitswap presentation, September 2019, by Dirk McCormick of Protocol Labs.

can check if the signature matches the public key and NodeId, thus verifying the authenticity of the published Object.

3.2.8 IPFS Usages

IPFS alone (excluding libraries that were modularized, like *libp2p*) has multiple use cases, all of them involving files, of course. It can act as filesystem: locally mounted under /ipfs and /ipns; mounted as personal sync folder that automatically versions, publishes and backs up any writes; as the boot or root filesystem for a Virtual Machine. Another application is using IPFS as a CDN for webpages, as an integrity checked CDN for large files or even as an encrypted CDN.

In fact, Netflix recently (February 2020) used IPFS as a CDN to help speed up image distribution for faster Continuous Integration pipelines[10]. Netflix wanted to address how to efficiently pull container images in a large scale, multi-region environment (image layers can reside in different regions). Downloading a 300 MiB image using IPFS was 20% faster than DockerHub, a popular library and community for container images[11].

Rehman et al. (2020) [66] used IPFS in a vehicular network architecture, in combination with blockchain technology to facilitate data sharing between vehicles.

Further applications involve using IPFS as a versioned package manager for software, or as a communications platform, or even as a permanent web, where links do not die. In 2017, Spanish courts blocked the *ipfs.io* gateway, as it was being used by the Catalan government to promote its referendum[12]. This shows that one of the applications of IPFS can be in overcoming censorship, since denying access to a gateway only prevents the use of a browser; it is harder to prevent someone from using an IPFS client.

[10]https://blog.ipfs.io/2020-02-14-improved-bitswap-for-container-distribution/
[11]https://hub.docker.com/
[12]https://www.abc.es/espana/catalunya/abci-jueza-ordena-desactivar-otras-tres-webs-sobre-referendum-201709231941_noticia.html

3.3 Final remarks

IPFS was designed taking the best ideas from protocols and softwares such as BitTorrent, Kademlia or Git, and combining them into a modern peer-to-peer file system. It can achieve high performances in *traditional* networks consisting of datacenters, servers and fiber optic links, as seen in the previous section with Netflix. These networks have a stable topology and connectivity, but how would IPFS perform in a scenario where this is not the case? In a scenario where, instead of cables connecting servers, there were intermittent wireless links connecting moving vehicles? Could it be used as a content distribution protocol in a VANET?

The following chapters discuss how IPFS was deployed to real OBUs - at first only in lab but later in a VANET - and the results that derived from those tests.

Chapter 4

IPFS Extensions to the Vehicular Network

This chapter presents the first IPFS tests which, although in the lab, resort to hardware similar to the one present in the real vehicular network. The main difference is that, in this case, mobility is not involved.

The tests and their results show how IPFS can behave in a small vehicular network. These results are then used to understand the adaptations that need to be performed to IPFS to make it work with high efficiency in a vehicular network. This chapter then describes these adaptations and extensions.

4.1 Hardware

The hardware initially used to run IPFS are NetRiderV4 boards, shown in Figure 4.1. These are the 4th generation of Veniam's custom boards, running OpenWRT 16.02[1]. They have a single core ARMv7l 32bit processor running at 1GHz and 256MB of RAM. The storage consists of Flash memory with a 256MB capacity, and a 16GB SD card.

The boards have a WiFi interface and an Ethernet port. The WiFi is Ad-Hoc 802.11ng, with a 40 MHz channel. iPerf[2] is used to measure the bit rate between two boards: 70 Mbps (8.75 MB/s) with UDP, 44 Mpbs (5.5 MB/s) with 24 TCP streams, and 21 Mbps (2.63 MB/s) with a single TCP stream. The Ethernet port is not relevant as everything was done wirelessly using exclusively the WiFi interface.

4.2 Initial testing

The first task of this dissertation was to test IPFS in real hardware in a controlled lab environment in order to understand its operation and performance, while studying how to deploy it and potential issues that might arise.

[1]OpenWRT - openwrt.org
[2]iPerf - iperf.fr

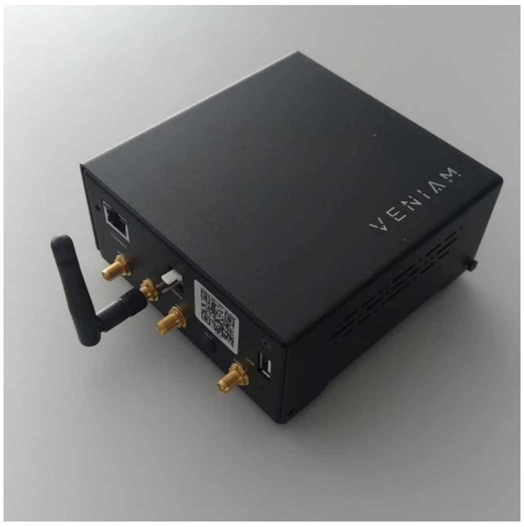


Figure 4.1: NetriderV4.

4.2.1 IPFS configurations and notes

The objective of this section is to group IPFS-related commands, configurations and considerations in order to allow the following sections to better focus on their topics:

- The default block/chunk size in IPFS is $256 * 1024$bytes, so the 52MB file used in the tests was split into $54277586/(256 * 1024) = 207.05 = 208$ chunks. Because IPFS uses a Merkle DAG, two blocks were created to reference the 208 data blocks, as they are not directly under the root CID, so the total number of blocks that IPFS created for this file was 210.
- Before running IPFS, it is mandatory to create the repository in IPFS_PATH (directory used by IPFS to store the blocks, private key, configuration files, ...) using the command `ipfs init`. A previously generated private key *swarm.key* was copied to this directory. A node's swarm will only contain other nodes that have that same *swarm.key*.
- IPFS daemon is the process that manages the node. It can be run on the foreground, but before a test starts, it is launched in the background using the command `ipfs daemon &`. The daemon supports several flags, one of them being important for disabling transport encryption: `-disable-transport-encryption`.
- Files were added to IPFS using the command `ipfs add <filename>`. Once added, they become available for any node in the swarm to download it.
- There are two commands to download a file, `ipfs get` and `ipfs cat`. The former downloads an IPFS object to disk, being able to output TAR and GZIP archives instead of the unpacked files. The latter displays the data contained by an IPFS object. For example, running `ipfs get -o /path/to/file hash_of_ipfs_object` is equivalent to running `ipfs cat hash_of_ipfs_object > /path/to/file`.
- When running a private IPFS network, bootstrap nodes (public servers that help speed up connectivity and discovery of other nodes online) are not required because the node

discovery is local and continuous. For that reason they were removed (`ipfs bootstrap rm -all`).

- The MDNS discovery interval (number of seconds to wait between discovery checks) was set to 1 second using `ipfs config -json Discovery.MDNS.Interval 1`.

- The `Addresses.Announce` key was changed to only announce the V2V(WiFi) interface address, as opposed to announcing the address of every interface. This forces IPFS to only exchange data through this interface.

4.2.2 Test specification

The IPFS' latest (in late January 2020) ARM binary is used, v0.4.23[3]. It was installed on 2 boards to perform the first tests.

One board (in this section addressed as Node TX) is the *seeder*, responsible for adding a 50 MB file to IPFS. Another board (in this section addressed as Node RX) has to download that file. It should be reminded that in IPFS the contents are addressed by their hash, not their name.

Both Node TX and Node RX are only connected to each other by WiFi and are placed in a table, 30cm apart from each other without any obstacles. A private key is generated and distributed to both nodes in order to establish an encrypted IPFS network (swarm). In Node TX, the IPFS repository is created, the daemon is launched in the background and the file is added. In Node RX, the first two steps are the same, but instead of adding the file, it downloads the file.

On average, this download takes 65 seconds to complete, which corresponds to an average throughput of 6 Mbps. The boards are placed close to each other: iPerf measures up to 70 Mbps in the WiFi link. Therefore, the IPFS content distribution rate is over 10 times slower than the link's capacity, which is not ideal when performing content distribution in vehicular networks. While looking through the logs, two potential problems were identified:

- A TCP connection opens and closes for every request (checking the DHT, sending and requesting blocks, managing 'want' lists, etc.). Because of TCP's congestion control methods (namely slow-start), this can explain the issue.

- Blocks are requested sequentially and not at the same time (a block can reference more blocks, being a tree, and perhaps each *branch* is sequentially traversed, moving to the next one only after being done). This can also explain the low performance.

In order to better understand what was happening and try to solve the performance issues, six specific aspects were investigated:

- Connectivity

- Data Storage

- CPU

- Transport Encryption

[3]IPFS v0.4.23 - github.com/ipfs/go-ipfs/releases/tag/v0.4.23

- Hashing
- Connection Persistency

4.2.3 Connectivity

If IPFS has frequent disconnects, it can impact its performance. To assess this, a script checked for peers in the IPFS swarm every 0.5s. This process was combined with observing the logs for errors or any warning, and it showed that there were not any disconnections between Node RX and Node TX during each test.

4.2.4 Data Storage

The boards only have 256 MB of storage in the Flash memory, not much space to work with. But since there is a SD card with 16GB, almost fully free, the IPFS_PATH was set to the SD card.

The SD card was chosen for precaution, due to a concern that IPFS would take too much space (more than available in the Flash or RAM). However, this meant that out of the three available storage options (RAM, Flash and SD card), the slowest one was being used.

Table 4.1 compares the different write speeds (tested with `dd if=/dev/zero of=[SD, Flash, RAM] bs=8k count=10k`) between storage mediums, showing that writing to the SD card is 33 times slower than using the RAM. There is a trade-off though: it may be 33 times slower, but is 62 times larger (16GB vs 256MB), and considering that the RAM is not fully free for IPFS to use, the SD/RAM capacity ratio gets even larger. The decision of what storage media to use lies on whether the scenario in which IPFS is to be used requires high capacity or high speeds: cannot have both simultaneously in this hardware. In this particular case, because the network would only have a small amount of small files (50MB or less), RAM was the chosen storage media.

Table 4.1: Comparison of write speeds and IPFS download times between different storages.

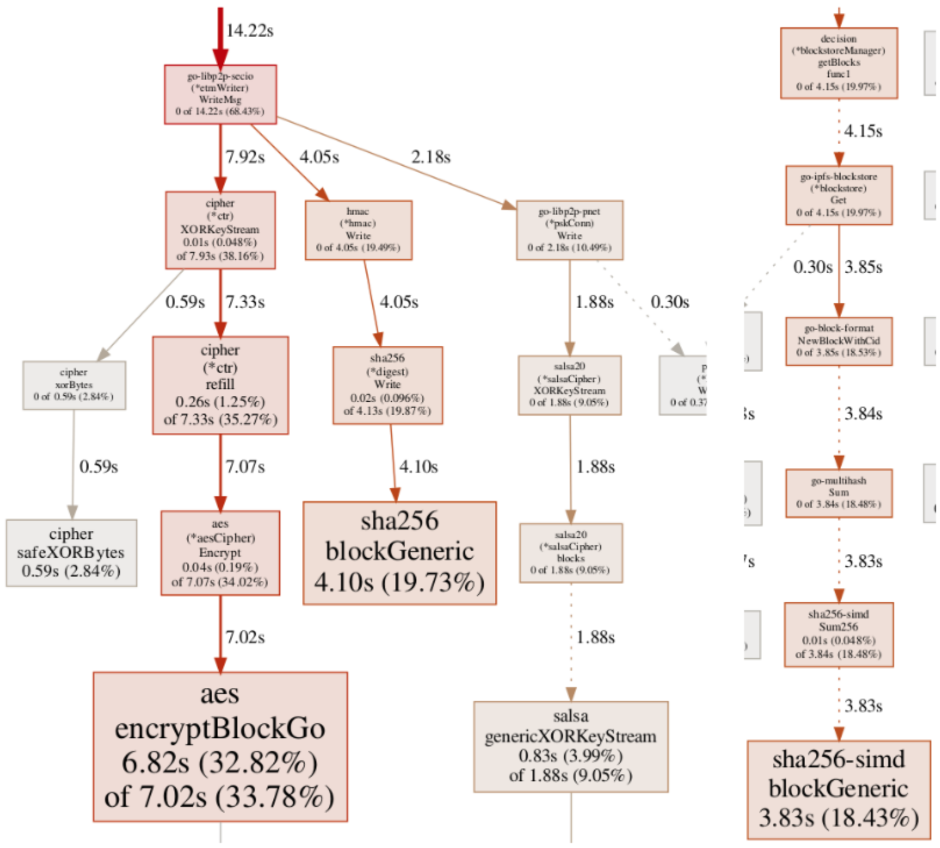
	SD	Flash	RAM (/tmp)
Write speed (MB/s)	10.3	66.6	333.3
IPFS download (s)	65	46	39

Table 4.1 also shows the time it takes for Node RX to download all the 210 blocks (average of 5 tests) of the 50MB file. Running the test with IPFS in `/tmp` was 1.7x faster than with the SD card, despite RAM being over 30x faster. This shows that, despite the storage bottleneck disappearing, something else was still slowing things down: if the issue was only storage, the IPFS download improvement would be much higher than just 1.7x.

4.2.5 CPU

Because the NetriderV4 boards only have a single 1GHz ARM core, it would be possible that IPFS would have algorithms or functionalities too complex for a low power CPU such as this one. Inspecting the running processes (`ps -a`) showed that IPFS used most of the CPU and the usage was high, but it was not at 100% or as high as it could be. Therefore, it was decided to understand how and where the CPU cycles were being spent. Fortunately, IPFS supports Go's pprof tool[4] and it can be used to render a SVG with the CPU profiling information.

Note that the percentages mentioned in Figure 4.2 and Figure 4.3 and in the following paragraph correspond to the portion of time the application spent doing each specific processing. For example, if the CPU profiling runs for 10 seconds and it shows that SHA2-256 used 20%, it means that 2 seconds were spent in SHA2-256, not that SHA2-256 was using 20% of the CPU's capacity.



14.22s
go-libp2p-secio (*etmWriter) WriteMsg 0 of 14.22s (68.43%)
7.92s
4.05s
2.18s
cipher (*ctr) XORKeyStream 0.01s (0.048%) of 7.93s (38.16%)
hmac (*hmac) Write 0 of 4.05s (19.49%)
go-libp2p-pnet (*pskConn) Write 0 of 2.18s (10.49%)
0.59s
7.33s
4.05s
1.88s
0.30s
cipher xorBytes 0 of 0.59s (2.84%)
cipher (*ctr) refill 0.26s (1.25%) of 7.33s (35.27%)
sha256 (*digest) Write 0.02s (0.096%) of 4.13s (19.87%)
salsa20 (*salsaCipher) XORKeyStream 0 of 1.88s (9.05%)
0.59s
7.07s
4.10s
1.88s
cipher safeXORBytes 0.59s (2.84%)
aes (*aesCipher) Encrypt 0.04s (0.19%) of 7.07s (34.02%)
sha256 blockGeneric 4.10s (19.73%)
salsa20 (*salsaCipher) blocks 0 of 1.88s (9.05%)
7.02s
1.88s
aes encryptBlockGo 6.82s (32.82%) of 7.02s (33.78%)
salsa genericXORKeyStream 0.83s (3.99%) of 1.88s (9.05%)
decision (*blockstoreManager) getBlocks func1 0 of 4.15s (19.97%)
4.15s
go-ipfs-blockstore (*blockstore) Get 0 of 4.15s (19.97%)
0.30s
3.85s
go-block-format NewBlockWithCid 0 of 3.85s (18.53%)
3.84s
go-multihash Sum 0 of 3.84s (18.48%)
3.83s
sha256-simd Sum256 0.01s (0.048%) of 3.84s (18.48%)
3.83s
sha256-simd blockGeneric 3.83s (18.43%)

Figure 4.2: Profile of Node TX during a test using IPFS.

[4] Go profiling with pprof - golang.org/pkg/runtime/pprof

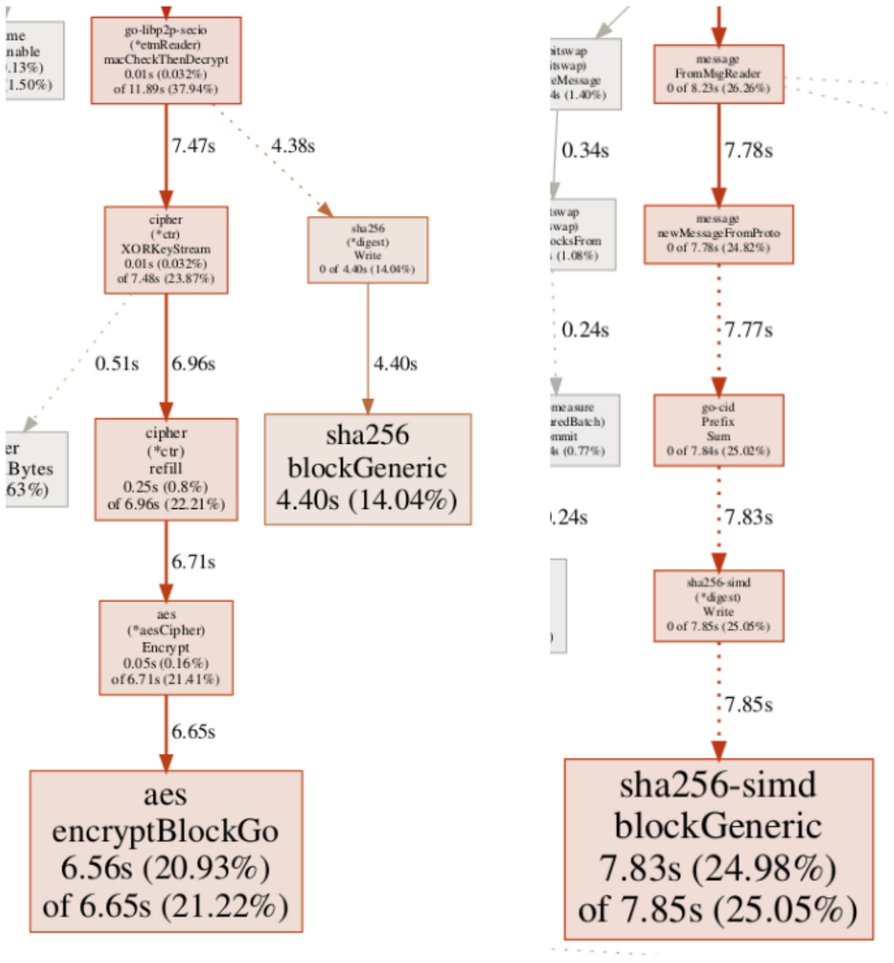

go-libp2p-secio (*etmReader) macCheckThenDecrypt 0.01s (0.032%) of 11.89s (37.94%)
7.47s
4.38s
cipher (*ctr) XORKeyStream 0.01s (0.032%) of 7.48s (23.87%)
sha256 (*digest) Write 0 of 4.40s (14.04%)
0.51s
6.96s
4.40s
cipher (*ctr) refill 0.25s (0.8%) of 6.96s (22.21%)
sha256 blockGeneric 4.40s (14.04%)
6.71s
aes (*aesCipher) Encrypt 0.05s (0.16%) of 6.71s (21.41%)
6.65s
aes encryptBlockGo 6.56s (20.93%) of 6.65s (21.22%)
message FromMsgReader 0 of 8.23s (26.26%)
0.34s
7.78s
message newMessageFromProto 0 of 7.78s (24.82%)
0.24s
7.77s
go-cid Prefix Sum 0 of 7.84s (25.02%)
7.83s
sha256-simd (*digest) Write 0 of 7.85s (25.05%)
7.85s
sha256-simd blockGeneric 7.83s (24.98%) of 7.85s (25.05%)

Figure 4.3: Profile of Node RX during a test using IPFS.

Figure 4.2 and Figure 4.3 show the highlights of the CPU profiling on both nodes, TX (seeder) and RX (downloader) respectively, during an instance of the test detailed in Section 4.2.2. It shows that both nodes spend over 50% of CPU time in AES (encryption) and SHA2-256 (hashing). This does not necessarily mean that the CPU is at fault for IPFS performing poorly, but it does show that more time is being spent performing cryptographic operations than on IPFS-specific functions.

Taking into consideration that modern CPUs often have co-processors for cryptographic operations (Intel introduced instructions to facilitate AES encryption and decryption in 2009, providing full hardware support for AES [67]), the lack of a cryptographic co-processor in the ARM CPU (hardware-accelerated cryptography was only introduced in the ARMv8 architecture[5]) explains why AES encryption and SHA2-256 hashing take such a large portion of the execution time.

AES encryption and SHA2-256 are further discussed in the following sections.

[5]ARMv8 Cryptography - developer.arm.com/docs/ddi0501/latest/introduction/about-the-cortex-a53-processor-cryptography-extension

4.2.6 Transport Encryption

Connections between IPFS nodes are encrypted and secured with a Diffie-Hellman handshake using a custom transport protocol called SECIO (v0.5.0 and onward switch to Transport Layer Security (TLS) [68]): this is enabled by default. On top of that, optionally a symmetric key (the same key is used in both encryption and decryption) can be generated and used to encrypt all traffic, meaning that a new node cannot join the network without having that key. Therefore, generating and sharing the same key between selected nodes allows the creation of a fully private IPFS network.

Figure 4.2 and Figure 4.3 show AES encryption taking 33% and 21% of CPU time on Node TX and Node RX, respectively. AES stands for Advanced Encryption Standard, and it is a symmetric-key algorithm that is used in this scenario to encrypt the network packets before transmission, and decrypting them upon reception.

Disabling transport encryption entirely reduces the download times to 92%, 60% and 75% when running on SD, Flash and RAM, respectively. The greatest performance improvement is observed when IPFS is using the Flash storage: if for example a download takes 10 seconds to finish, disabling transport encryption reduces the download time to 6 seconds.

However, switching from SECIO to TLS naturally does not have an impact as high as disabling transport encryption altogether. When using TLS instead of SECIO, the download times are reduced to 87%, 80% and 80% when running on SD, Flash and RAM, respectively. This performance difference is part of the reason why IPFS is moving away from their custom transport protocol and into a more established and tested one like TLS.

As discussed in Section 2.4, security can be an impactful area when it comes to the functioning of a VANET, and transport encryption greatly contributes to it, not only by exclusively allowing authorized nodes (must have the key) to be part of the network, but also maintaining confidentiality, as the network traffic is encrypted. However, this comes at a performance cost as high as 30% (disabling transport encryption on Flash vs SECIO on Flash).

TLS was used as the transport encryption protocol, but for each network and scenario/use-case it should be carefully pondered which is best: faster times or having a fully private, encrypted network.

4.2.7 Hashing

A hashing function is a unidirectional function that maps data of arbitrary size to a fixed-size value called a hash. It is not possible to retrieve the original data using its hash. Usually hashes are used to uniquely identify chunks of data (whether it is only 8 bytes or a HD video) by generating its hash, and to verify information, comparing a previously known hash against the generated one: if the hashes are different, it means that the information has changed and is not as expected.

Because of this, hashing is commonly used in P2P systems: a file is divided into several blocks, each block has its hash calculated and sent together with the block; the receiving node generates the hash of the block it received and compares it against the original hash to check if the information is correct (valid) or not (corrupted or entirely different, so it must be sent again).

IPFS has a similar behaviour: hashes are generated when adding a new file, when the blocks are sent (to validate that the block being sent is precisely what it should be) and once again when the blocks are received. By default the hashing function used is SHA2-256, but dozens of hashing functions are supported[6] (SHA1, SHA2, SHA3, SHAKE, Blake2b, Blake2s, Murmur3, MD5, and more).

In order to mitigate SHA2-256's impact, a different hash function, called Blake2b [69], was used. Before explaining what Blake2b is and how it compares to SHA2-256, it should be noted that the performance impact of disabling these hash verifications was tested, and although there was a performance improvement, it is not recommended for obvious reasons.

Blake2[7] [69] is a cryptographic hash function faster than MD5, SHA-1, SHA-2, and SHA-3; however, it is at least as secure as the latest standard SHA-3 [70]. Blake2 has two variations: Blake2b, which is optimized for 64bit platforms, including ARM and designed to be efficient on a single core CPU, and Blake2s, which is optimized for 32bit platforms and lower. The former is the one used, as it is slightly better (both were tested in the hardware and the differences were minimal). Figure 4.4 illustrates the performance difference between Blake2 and other hashing algorithms.

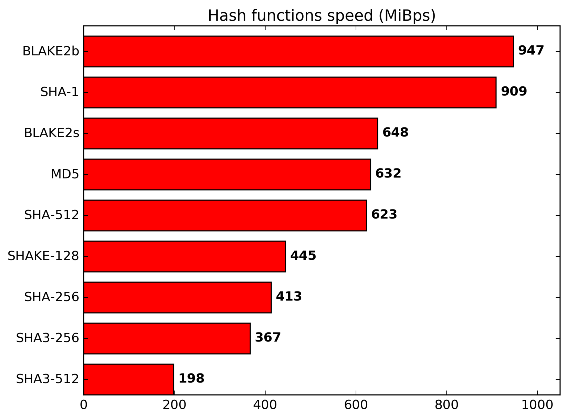


Figure 4.4: Comparison between hashing algorithms, taken from Blake2's website.[8]

A potential way to further improve performance in the future is to use Blake3[9] (announced in January 2020) instead, as it is 5x faster than Blake2b and 14x faster than SHA2-256. This will only be possible if Blake3 is available to be used in Go and if it is supported by IPFS.

4.2.8 Connection Persistency

It was previously mentioned that, when running IPFS, it looked like several TCP connections were being opened and closed for every request. Connections were indeed being opened for every request, but they were not TCP connections. As mentioned before, IPFS uses YAMUX[10] (better explained in Section 3.2.2), a streaming multiplexer that relies on a single underlying connection (such as a TCP or Unix socket) to provide bi-directional streams, flow control and keepalives. It implements the same interface as TCP, so from a software point of view opening a new stream in Yamux is the same as opening a new TCP stream: the main

[6]Multihash implementation in Go - github.com/multiformats/go-multihash
[7]Blake2 - blake2.net
[8]Blake2 - blake2.net
[9]Blake3 - github.com/BLAKE3-team/BLAKE3
[10]Yamux - github.com/libp2p/go-yamux

difference is that opening and closing streams in Yamux does not require any overhead or setup because it is based on top of an already open TCP connection.

When the connections were assumed to be TCP, it could be a possible cause for delays due to TCP's slow start mechanism (an algorithm that gradually increases the amount of data transmitted until it reaches the link's maximum capacity). However, as that was not the case and TCP connections were not being opened and closed for every request, this was not causing any delays or decreased performance.

4.3 Final remarks

Table 4.2 presents a comparison between hashing algorithms, storage options, transport encryption on/off and without hash checking[11]. This chapter was focused on discussing and understanding what could cause an IPFS transfer to only use 6 Mbps when up to 70 Mbps were available (UDP). Firstly, IPFS is overlaid on top of TCP, not UDP, so the maximum expected bandwidth of 70 Mpbs becomes 44 Mbps (24 TCP streams), which assumes that IPFS is using multiple streams. Because it uses YAMUX, it does use multiple streams, but overlaid over a single TCP stream, so the maximum expected bandwidth is reduced to 21 Mbps (1 TCP stream).

Table 4.2: Times to download a 50MB file using IPFS under varying parameters (seconds) - averages of 4 tests.

	TLS	transport off	hash checking off, transport off	
SD	46.9	40.4	35.6	blake2b
	60.0	59.4	53.5	sha256
Flash	34.3	29.2	26.1	blake2b
	37.2	33.2	28.4	sha256
RAM	26.6	21.7	17.7	blake2b
	29.4	25.0	20.4	sha256

Looking at the table, the worst result (60.0s, 6.7Mbps) occurs when using the SD card, TLS and SHA2-256. The previously observed bandwidth of 6 Mbps was when also using the SD card, SHA2-256 as the hashing algorithm, but SECIO instead of TLS (custom transport protocol, slower than TLS). Improved results can be achieved when using the RAM for storage, TLS, and Blake2b instead of SHA2-256, downloading the file in 26.6s averaging 15Mbps: this is more than double the performance, and much closer to the expected 21Mbps measured by iPerf in a single TCP stream.

The very best results (17.7s, 22.6Mbps) are achieved under conditions not recommended and prone to errors, when hash checking[12] and transport encryption are disabled.

It can be concluded that, at the beginning, the performance was so low because: **1)** the storage media used was the SD card, slowest out of the three; **2)** SECIO was used, slower than TLS; **3)** because of the low hashing performance of the CPU, SHA2-256 was slowing down all processes when compared to Blake2b (as presented in Table 4.2).

[11]Hash Checking - when creating a block from existing data and CID, IPFS checks if the given CID matches the given data by hashing it and comparing the result with the CID.

[12]See footnote 11.

Chapter 5

Network Mobility Emulator

The development of protocols for mobile networks, especially for VANETs, presents great challenges in terms of testing in real conditions. Using a production network for testing communication protocols may not be feasible, and the use of small networks does not meet the requirements for mobility and scale found in real networks. The alternative is to use simulators or emulators, but such platforms do not meet all the requirements for effective testing. Aspects closely linked to the behaviour of the network nodes (mobility, radio communication capabilities, etc.) are particularly important in mobile networks, where a delay tolerance capability is desired.

This chapter begins by presenting the motivation for the creation of a new emulator, EmuCD. The architecture and functionalities of EmuCD are then detailed, followed by a performance analysis of the emulator with different protocols and conditions.

5.1 Motivation

Despite recent technological advances with regard to vehicular networks, it is still very difficult to develop routing and content dissemination protocols in real scenarios [45]. Due to the risk and high cost of carrying out experiments on real vehicle networks [22], the alternative solution is the use of simulators and emulators for the early stages of the development, and executing only final tests on small testbeds. However, some aspects inherent to VANETs, such as mobility models, driver behaviour and wireless channel modelling have a significant effect on performance results. Due to these challenges, the simulation of realistic mobility models becomes essential for research in order to achieve precisely the desired results that reflect the realistic behaviour of vehicular traffic [71]. However, the channel modelling depends on many aspects, such as obstacles like trees, buildings, vehicles and even people on the streets, and this can only be measured in a real scenario.

Developing a content dissemination protocol for a VANET is not an easy task: the protocol may be good in theory and in simulations, but with low performance in a real environment. Moreover, even if it is possible to test the protocol in a real network (which is rarely the case), not only is the network different between each test run (preventing an accurate analysis and debugging), but it is also difficult to scale. Particularly in the case of a VANET, since the nodes keep moving and the network's topology is very dynamic, repeating a test under the same conditions is not an option.

This is why simulators exist: they provide reliable, deterministic results, and whether a real VANET dataset or a SUMO generated one is being used, it is possible to simulate the behaviour of a VANET although some difficulties are presented:

- **debugging** - errors can be hard to debug as the simulators are very complex; most errors probably result from issues in the user's custom protocol, but lack of documentation and support make solving them a challenge;

- **custom protocols** - testing a custom protocol in a simulator requires a custom implementation specifically for that simulator using its libraries or APIs; it is not a problem if the protocol is a simple one, but it may be very complex if it is an existing closed-source protocol that is not supported by the simulator. In Section 2.2.5 multiple VANET evaluation environments were presented, of which only OPNET[1] did not support any protocol other than the ones already built in. The only simulator that did not require a custom implementation of a protocol was NCTUns [35] (has some limitations) and perhaps GLOMOSIM [72] as well, but the latter is no longer under development and a free version is not available[2];

- **portability** - a protocol developed or ported to work with a specific simulator may not work with a different simulator/emulator and, most importantly, it is not ready to run in real hardware;

- **reliability** - simulators have to use models to represent the reality, encompassing mobility and radio propagation models. However, in a very challenging and mobile environment such as VANETs, these models usually fail to represent the reality as they cannot encompass the real obstacles and their effects in a real scenario.

From the issues mentioned above, the most prohibitive one that leads to the creation of a new emulator is the difficulty in testing different protocols. In a real in-vehicle OBU, there is a CPU, an OS, storage, RAM and one or more wireless interfaces. Software is developed taking these aspects in consideration, and it does not need to worry about the OS-related tasks, so using it in a simulation/emulation should be as effortless and transparent as running it in real hardware – making sure the required network interfaces and enough RAM are available to execute the software. Furthermore, when debugging software that is running in the simulator/emulator, the developer should only have to look at its own software, without having to know the details of the underlying simulator, its libraries or having to dive into its code.

To the best of our knowledge, there is not a solution that: **1)** offers easy integration with a custom protocol/software, especially when only the binaries are available, without the code; **2)** is as resilient and invisible as possible to a developer when debugging, so the developer can focus only on the protocol/software at hand; and **3)** provides results as close to reality as possible while offering customizable network metrics and tunable parameters.

5.2 EmuCD - a new emulator

The testing of a new protocol for content dissemination in networks using a traditional simulator requires, at least, two steps: first, it is necessary to configure the network with all

[1]https://opnetprojects.com/opnet-network-simulator/

[2]https://networksimulationtools.com/glomosim-simulator-projects/

its details (communication technology, node characteristics, interfaces, etc.); after that, it is necessary to deploy the new protocol, that is, the customisation of the protocol inside of the simulator following all the rules defined by this software. In order to do these tasks, a deep knowledge of the simulator is required, starting with an initial stage of studies to absorb the entire process for the network configuration, and then the implementation process of the new protocol for the simulator, where a knowledge of its libraries is required.

On the other hand, what is expected from a virtual environment capable of developing and testing networks and protocols is for it to be as realistic as possible. It should also offer conditions that allow transferring the work onto real physical networks, with a minimum effort regarding deployment and preparation. However, this is not offered by traditional simulators, as after the development period, a customisation period is still required for the protocols to be properly adapted to the conditions of the real physical network. A simple example is the configuration of the network and its nodes: each simulator has its own process for these definitions, usually stored in tables/files of the simulator, which do not correspond to the settings of the operating systems.

Another aspect closely linked to the realism of the simulation is what is expected from the behaviour of the network nodes. This aspect is particularly important in mobile networks where a delay tolerance capability is desired. Taking into account a VANET where the characteristics of each city, the habits of the inhabitants and the weather conditions have a great influence on the mobility of the vehicles, it is very important to use real traces taken directly from the vehicular networks installed in the cities. Such traces provide a degree of realism that simulators like SUMO cannot achieve.

5.2.1 Architecture

In EmuCD, each container acts as a real Ubuntu node with a configurable number of network interfaces, and follows the mobility provided by a dataset from a VANET. Because the nodes are not really moving, and to support everything being run locally in a single machine, the mobility aspect of the emulation is created by manipulating the routing table in the OS of each container. Each node knows its neighbours at every instant, and the routing table is updated so that the node can only reach its neighbours, as it would if it were in the VANET.

It should be noted that, due to the aimed abstraction between the emulator and the software/protocol being tested, the emulator only supports standalone software, whether it is a Python script or a standalone binary. It is the software's responsibility to perform tasks such as neighbour discovery (UDP broadcasts, network pings, etc.) or data management, as the emulator does not provide any support. As a rule of thumb: if the software/protocol can run in real hardware, it is ready to run in the emulator. This is the first time that a protocol can be tested in an emulated VANET without any modifications to the protocol itself: all that is required is a data trace, the software to be tested and the edition of some configuration files in the emulator.

An overall view of the architecture and its components is available in Figure 5.1. The nodes are Docker containers, with each one running a **daemon** and being controlled by a single **manager**. Ubuntu 16.04 is the base of the image, so each node can be considered to be a full OS with its own routing tables, network interfaces and storage.

The **manager** is decoupled from the nodes in a way that allows for multiple **managers** to exist in an emulation: for example, each **manager** coordinating a different set of containers in separate hardware, in order to better emulate realistic conditions. The coordination between all nodes is done by this component, as it is responsible for:

- Loading the emulation's parameters from a JSON file;
- Starting the emulation;
- Waiting for each container to be "booted" and ready;
- Synchronising every container;
- Building the emulation log file;
- Providing data for the dashboard (nodes' positions, CD progress, CD metrics);
- Polling the current status from each container's daemon;
- Collecting and storing CD metrics.

Each node is running a **daemon** process. The main task of the **daemon** is to update the node's IP routing table according to the neighbours in a given instant. In order to do this, two important pieces of information are required: the current node's neighbours and their IPs. The neighbours are read from the dataset (stored in a file), for the relevant node at the current timestamp. These node IDs are then converted into the respective container's IP. If, for example, one of the neighbours is the node with ID 332, its container's IP address is 172.20.3.32, so an entry in the routing table is added by running the command `ip route add 172.20.3.32 dev $IF`.

The dataset must be available prior to the emulation start, as it is included in the Docker image when being built. It consists of several files, one per node, with each file containing several entries. An entry is represented by a timestamp, latitude, longitude and a list of neighbours in the format of (nodeID, RSSI) tuples. A node is only considered to be a neighbour if its RSSI meets a configurable threshold.

Although the **manager** and **daemon** are the core of the EmuCD emulator, a web dashboard was created to assist with the visualisation of the emulator's status and progress. It periodically polls the `/status` endpoint of the manager to display the positions of every node in a map (by default, OBUs as blue dots and RSUs as orange dots), as well as the progress per node of the content's distribution (Figure 5.2). The dashboard can also show an overview of the metrics collected, like the cumulative download progress, the number of OBUs that reached 100% in each timestamp, or the average download progress across all OBUs.

5.2.2 Implementation

Both the emulation **manager** and the **daemons** are Python web servers that provide access and/or control through HTTP endpoints.

Manager endpoints and main functions

This section presents the manager's endpoints that are ready for `GET` requests (either using a web browser or a tool like *curl*) and its main functions.

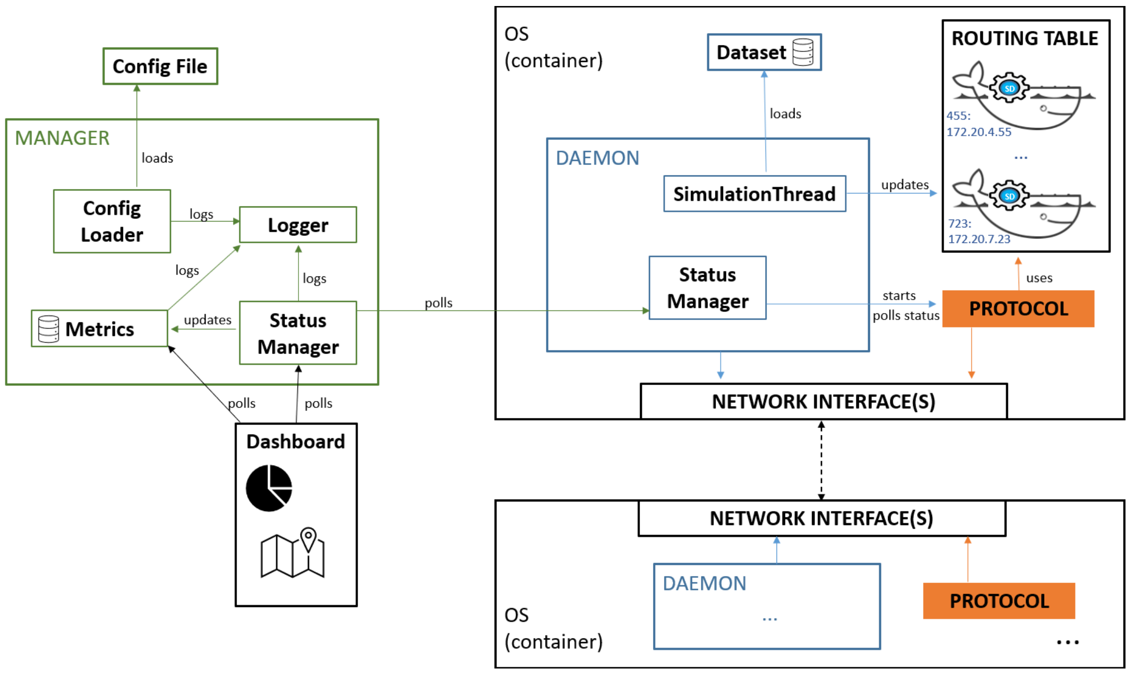


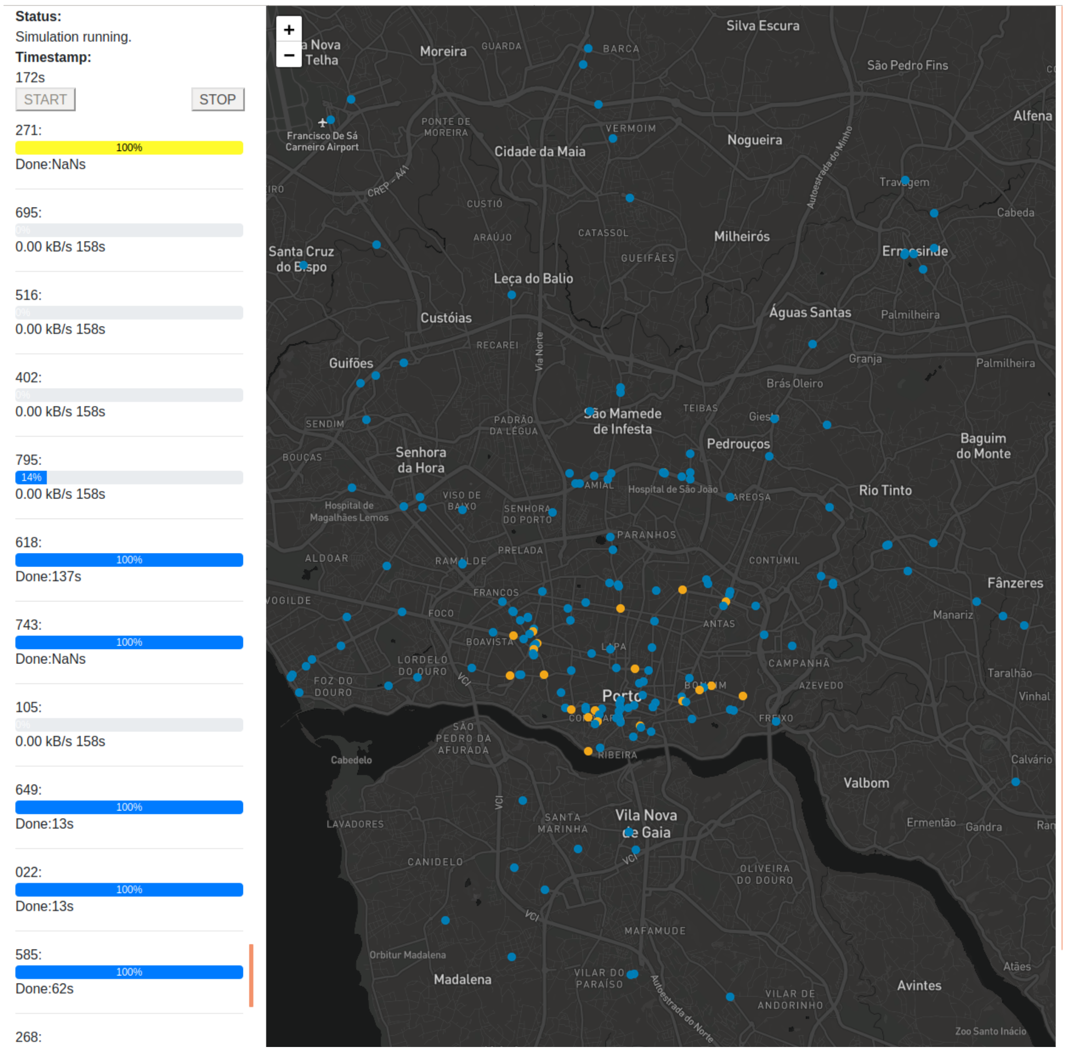
Figure 5.1: Architecture of the EmuCD Emulator.

main() In `main()` the logger is initialized (further detailed in Section 5.2.2) and the server is started on the address and port `0.0.0.0:5050`. After the server is up, the frontend's resources become available and the emulation can be started by calling `/start`, after which every other endpoint can be used.

/, /js/<path>, /img/<path> These endpoints serve the emulator's frontend dashboard, which can be seen in Figure 5.2: on the left side there is information regarding whether the emulation is already running and time elapsed; there is also information about each of the nodes, such as the total download progress, ID and if they are an OBU (represented in blue) or RSU (represented in yellow); the rest of the interface is a map where the nodes are represented in real time (provided they have latitude and longitude information).

/start This endpoint realizes the required tasks in order for the emulation to start. First, the list of nodes is built, which can either be every node if there are no limits, or it may be limited to the N most active nodes (the behaviour of which nodes to select is decided in the `allowed_nodes` function, it may be changed to only keep the N less active nodes). Afterwards, for every node on the list, a thread is launched to start a container for that node. The reason for parallelizing the containers booting is to ensure the fastest start possible (a container booting can take several seconds), only advancing when the containers are booted. Once every container is booted, each one is polled on the `/status` endpoint to ensure it is ready for operation - if the endpoint timeouts 3 times, the container is stopped, removed and restarted.

When every container is ready for the emulation to start, a start time is set to 30 seconds from the current time (it can be changed): this is done to synchronize the start in every node (the alternative is sending a message or `GET` request to every node at the same time to ask it



```
Status:
Simulation running.
Timestamp:
172s
START          STOP
271:
100%
Done:NaNs
695:
0.00 kB/s 158s
516:
0.00 kB/s 158s
402:
0.00 kB/s 158s
795:
14%
0.00 kB/s 158s
618:
100%
Done:137s
743:
100%
Done:NaNs
105:
0.00 kB/s 158s
649:
100%
Done:13s
022:
100%
Done:13s
585:
100%
Done:62s
268:
```

Figure 5.2: Frontend of the network emulator running a scenario with 200 nodes in Oporto.

to start, which is very hard to do perfectly for hundreds of nodes); time must be exactly the same on each node, which is not a problem if the emulation is running in a single machine. This time is then sent to every node's daemon through `/start?time=<ts>`.

The file where the CD metrics will be written is also created in this step.

/stop If the emulation is running, every container is stopped and the emulation ends.

/stop_count This is an endpoint for a daemon to signal that it finished. Once this endpoint is called as many times as there are nodes (meaning that every node is finished), everything is stopped, closed and the program finishes.

/status If the emulation is not running, this endpoint either returns *"Simulation is not running"* or *"Simulation is set to start in N seconds"*; otherwise it returns a JSON with the status information of every node. It is mainly used by the frontend.

Daemon endpoints and main functions

This subsection presents the daemon's endpoints that are ready for `GET` requests and its main functions.

/start If an emulation is already running, this endpoint returns *"Simulation in progress. Cannot start a new one!"*; otherwise the emulation is scheduled to start at the given time. If no start time is provided, it starts 5 seconds after it was called.

/status This endpoint returns a JSON with the current status information: timestamp, if it is an RSU, latitude, longitude, current download progress, total download size, current download speed and current neighbours.

main() The daemon starts by opening loading its data file. If the node's ID is `123`, the file `123.pickle` will be loaded as the emulation's data, stored globally.

thread_simulation() This is the most important function, started in a thread when the `/start` endpoint is called and sleeping until the starting time. Before starting the emulation loop, every route in the subnetwork is cleared (`ip route del 172.20.0.0/16`) except for the emulation manager (`ip route add 172.20.0.1 dev eth0`). If the node is an RSU, the file is prepared (taking IPFS as an example, the file is added - `ipfs add file.pdf`)

The emulation itself is a loop over the data: each iteration reads the timestamp, latitude, longitude and neighbours (list of `ID,rssi` tuples), keeping only the neighbours that are over the RSSI threshold; the status is updated with the current information, previous routes are cleared if required (neighbours that were previously in range but are not anymore) and the new nodes/neighbours are added to the routing table by running `ip route add <ip> dev eth0`. The thread then sleeps as long as needed until the timestamp of the next event.

When the emulation ends, the `/stop_count` endpoint in the manager is called.

Logging

There is logging both in the emulation manager and daemons, although only the manager's is persistent. The manager logs to a file using the `logging` Python module. Each container's daemon outputs to the standard output, available through Docker's log command (`docker logs CONTAINER`). When the emulation ends and the containers are removed, their logs are deleted as well.

Manager In the manager it is logged which containers are booted and when, when the containers are ready, emulation start, emulation end and every time the `contentMetrics` function is called.

Daemon In the daemon it is logged the start and end of the emulation and every event (timestamp, latitude, longitude and neighbours).

Metrics

The function `contentMetrics` is executed in the emulator manager with every `/status` call, and by default it aggregates the delivery rate (number of OBUs that have reached 100% in each *delta_ t*), cumulative distribution (cumulative file download progress in the network, for each *delta_ t*) and progress rate (average of download progress across all OBUs, for each *delta_ t*) of every node. It is written in a way that easily supports the addition of more metrics.

5.2.3 How to test a new protocol

Assuming that a dataset in a compatible format is already available, there are a couple of steps required in order to test a custom protocol or software in the emulator:

1. **New branch** - optional, but the creation of a new branch for the protocol is recommended, maintaining the repository's structure;

2. **Docker** - a new Docker image must be created: in `docker/build.sh` the image name must be edited, alongside any required files and changes that must be setup prior to building the Dockerfile; in `docker/Dockerfile` any software dependencies should be added, as well as the files that will be present when the container starts; the `docker/docker-startup.sh` script could also need a small customization as it is the script that runs when the container boots (used as an init script to start processes);

3. **Daemon** - the emulation's daemon will likely need changes in the `thread_simulation` function, as each protocol has different requirements. The status' update function also needs to be adapted for each protocol because of the download progress (some software updates the progress to stdout, others provide an API, etc.): it is not mandatory, but metrics will not be available unless it is done.

5.2.4 Kernel parameters

The Linux kernel's default settings do not support hundreds of Linux containers running simultaneously. The issues range from ARP tables overflowing to hitting the limit on opened sockets. In order to solve them, `sysctl` was used to change the kernel's network parameters at runtime, with each parameter's description taken from the Linux kernel's official documentation.[3] This section describes the changed settings and why they were necessary.

Sockets and TCP

`sysctl -w net.ipv4.ip_local_port_range="1024 65000"` defines the local port range that is used by TCP and UDP to choose a local port. The default values are 32768 and 60999.

`sysctl -w net.ipv4.tcp_tw_reuse=1` enables reuse of TIME-WAIT sockets for new connections when it is safe from a protocol viewpoint. The default value is 2 (enables reuse for loopback traffic only) which should be enough as everything was running in a single machine, but for precaution it was changed to be global enabled (also because there are no downsides, as the changes are not permanent and the emulator was running in a dedicated VM).

The length of time an orphaned (no longer referenced by any application) connection will remain in the FIN_WAIT_2 state before it is aborted at the local end is defined. While a

[3]https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

perfectly valid "receive only" state for an un-orphaned connection, an orphaned connection in FIN_WAIT_2 state could otherwise wait forever for the remote to close its end of the connection. With a default value of 60 seconds, it can be problematic since a node that opened the connection may suddenly loose its neighbours and remain open wasting resources. So instead it was reduced to 15 seconds by running `sysctl -w net.ipv4.tcp_fin_timeout=15` but it is a matter of fine tuning.

`sysctl -w net.ipv4.tcp_wmem="4096 65536 16777216"`: vector of 3 integers: *min*, *default*, *max*. *min* is the amount of memory reserved for send buffers for TCP sockets (4K default); *default* is the initial size of send buffer used by TCP sockets (16K default); *max* is the maximum amount of memory allowed for automatically tuned send buffers for TCP sockets (default between 64K and 5MB, depending on RAM size).

`sysctl -w net.ipv4.tcp_rmem="4096 65536 16777216"` is the same as `wmem`, but replacing "send buffer" with "receive buffer".

`sysctl -w net.ipv4.tcp_max_tw_buckets=400000`: maximal number of timewait sockets held by system simultaneously. If this number is exceeded time-wait socket is immediately destroyed and warning is printed. It was set to a high value as a precaution.

`sysctl -w net.ipv4.tcp_no_metrics_save=1`: by default, TCP saves various connection metrics in the route cache when the connection closes, so that connections established in the near future can use these to set initial conditions. Usually, this increases overall performance, but may sometimes cause performance degradation. It was set so that TCP does not cache metrics on closing connections.

`sysctl -w net.ipv4.tcp_syn_retries=2`: number of times initial SYNs for an active TCP connection attempt will be retransmitted. Default value is 6.

Connection tracking

Connection tracking is a core feature of Linux kernel's networking stack[4], allowing the kernel to keep track of all logical network connections or flows, being one of the basic features behind NAT and firewalls. In order to prevent dropped connections, which can be an issue on *loopback*, the size of the connection tracking table can be changed by running `sysctl -w net.netfilter.nf_conntrack_max=262144`.

ARP

The Address Resolution Protocol (ARP) is a protocol used in the Layer 2 level to discover a link's MAC address. Since there are dozens or even hundreds of nodes/containers running, attempting to discover each other and communicate, some settings must be updated in order to support a highly loaded docker swarm:

- *gc_thresh1* - minimum number of entries to keep. Garbage collector will not purge entries if there are fewer than this number, 128 by default. Can be changed with the command `sysctl -w net.ipv4.neigh.default.gc_thresh1=48000`;

- *gc_thresh2* - threshold when garbage collector becomes more aggressive about purging entries. Entries older than 5 seconds will be cleared when over this number, 512 by default. `sysctl -w net.ipv4.neigh.default.gc_thresh2=72000`;

[4]https://www.kernel.org/doc/Documentation/networking/nf_conntrack-sysctl.txt

- *gc_ thresh3* - maximum number of non-permanent neighbor entries allowed, 1024 by default. `sysctl -w net.ipv4.neigh.default.gc_thresh3=96000`.

These values were iteratively increased from the defaults until there were no more ARP errors (too many ARP requests, nodes not being able to find each other, etc.).

File limit

`ulimit` is used to establish system or user limits such as number of processes, CPU time or stack size. Soft and hard limits can also be set: a soft limit (`-S`) is the value that the kernel enforces for the corresponding resource; a hard limit (`-H`) acts as a ceiling for the soft limit. The most relevant limit here is the maximum number of open file descriptors, due to the potentially high number of containers and processes. The default value is 1024 but it is increased to 8192 (with 16384 as a ceiling) by running `ulimit -Sn 8192` and `ulimit -Hn 16384`.

5.3 Emulator performance

Regarding hardware, generally speaking, there are two main factors that have an impact on the number of nodes and protocols running in EmuCD: **RAM** and **CPU**.

RAM limits how many containers/nodes can be launched, largely independent of their activity (whether they have constant network traffic or not). For example, 16GB of memory may be enough to run 200 nodes with protocol A, but only 100 nodes when using protocol B. This can happen for several reasons, two of them being that A's binary is smaller than B's, or protocol B has a larger memory consumption than protocol A.

CPU, on the other hand, can usually limit the emulation in three ways: **1)** usage must be under 100% (ideally under 50%) to assure a valid parallelism between containers (e.g. if there are events and position updates every second, when the CPU is under full load, there is no guarantee that every packet transfer and operation that should be performed within each second, really happened as intended or if the CPU did not have time to do everything in that time span); **2)** high network activity causes high CPU usage (a network with 50 nodes transferring data simultaneously between each other may have the same CPU usage as 300 nodes in a sparse scenario); **3)** the amount of processing done by the protocol/software before/after sending each chunk/block influences the CPU usage (a protocol with a simple logic or algorithm allows for EmuCD to run more nodes than a very complex protocol).

Across every test and scenario presented in this document, EmuCD is running in an Ubuntu 18.04 virtual machine (VM), with 7 dedicated CPU cores at 4.6GHz and 27GB of RAM. This section presents the CPU and RAM usage of the VM while running the emulator (disk usage is not discussed, maximum observed read and write speeds during the stress tests were under 100MB/s, so this only has impact if it is an old hard disk drive). Profiling shows the emulator booting the containers and the beginning/end of the tests. The CPU and RAM graphs are not restricted to the emulator and its processes, they are for the entire VM: idling and without an emulation running, CPU averaged 0% across all cores and 1GB of RAM was in use, so the CPU graphs can be considered to be representative of the emulator and its processes; in the RAM graphs, 1GB must be subtracted from the memory in use.

Relevant changes/parameters worth noting are the `sysctl` changes (detailed in Section 5.2.4) and the network limitation of 16Mbps upload and 16Mbps download per container, to have similar restrictions to the wireless connections.

Running blank

Starting by analysing just the emulator, without any protocol running on the containers, Figure 5.3 and Figure 5.4 show the CPU and RAM usage of 100 nodes, respectively. Regarding CPU, there is a spike starting at 1:19:50PM until 1:20:20PM, which corresponds to the required Docker containers being created and booted (hence the corresponding spike in RAM); the small spike 30 seconds later is due to the emulation start, also noticed by a slight increase in RAM used. Because the containers are running "empty" there is not any CPU usage until 1:24:20PM, when the containers are stopped and removed. Memory usage during this test remained steady at 4GB, which means that 3GB were used by the emulator's containers and processes, resulting in an **average of 30MB per container**. The same test with 200 nodes showed that 6GB are used by the emulator's containers and processes.

Adding the RAM footprint of the tested protocol/software to these 30MB yields the RAM usage per container, which can then be multiplied by the number of nodes to find the required amount of RAM.

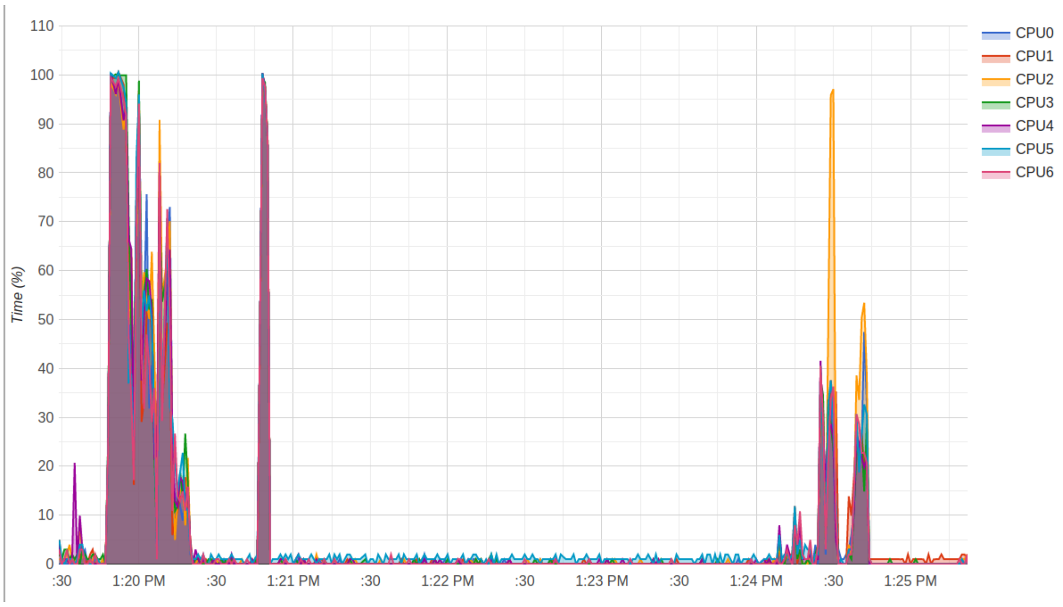


Figure 5.3: CPU usage of EmuCD without any protocol, 100 nodes.

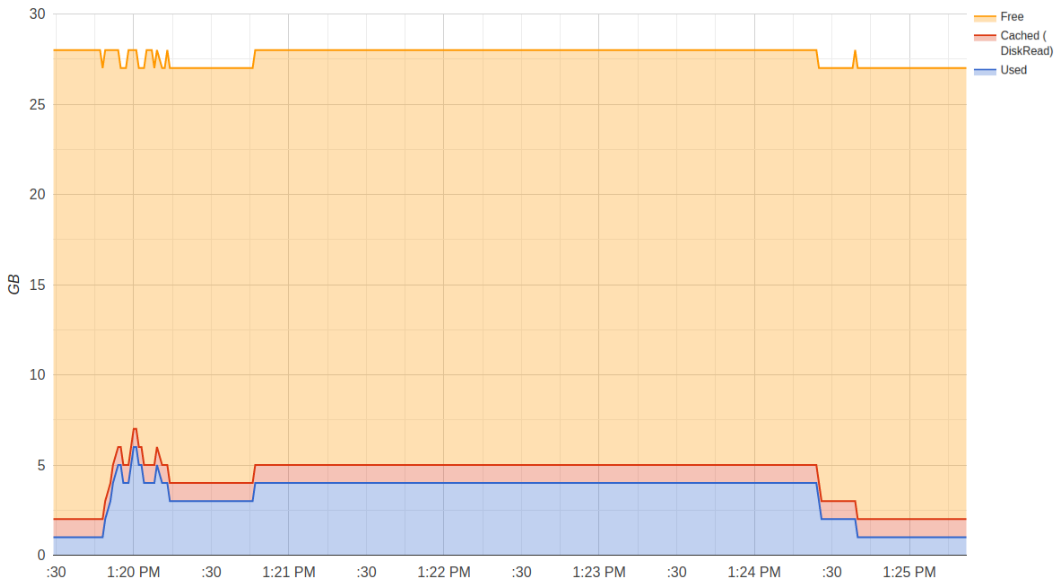


Figure 5.4: RAM usage of EmuCD without any protocol, 100 nodes.

BitTorrent

BitTorrent was used to illustrate the difference between a stress test (worst case scenario) and a regular test using real mobility data. The worst case scenario is an edge case where every node is at full activity. It is very rare to have this behaviour in a VANET, so a stress test was developed, consisting of 95 OBUs and 5 RSUs (a 50MB file starts only in the RSUs), where each OBU is connected to one of the RSUs and to every other OBU (each OBU has 95 neighbours). This is an extremely dense scenario, built to push the limits of the emulator (in fact it is so demanding that the hardware can only handle 100 nodes). Each RSU is sending the BitTorrent files to the OBUs, which then will propagate to other OBUs. On the other hand, the regular test consists of 200 nodes, 184 OBUs and 16 RSUs based on a real mobility dataset, distributing a 5MB file.

Figure 5.5 and Figure 5.6 show the CPU and RAM usage, respectively, of running 100 nodes in a regular scenario, with BitTorrent disseminating a 50MB file during a hour of real VANET data mobility. Starting with the CPU analysis, the first spike at 11:40AM corresponds to the nodes being booted and the emulation starting, whereas the last spike at 12:40PM is the emulation ending and the containers shutting down. The spikes in between are due to an increased network activity between containers, with the average CPU usage hovering around 5% during the process, naturally higher than in the previous case where the emulator was not running any protocol. Regarding RAM, BitTorrent can be a lightweight protocol, with an average 35MB used per container (meaning that around 5MB were used by the protocol).

Figures 5.7 and 5.8 show the CPU and RAM usage during the stress test, respectively. There is an increased CPU usage in the beginning, from 11:40PM until 11:41PM, due to the containers being booted. The test starts at 11:41PM and an increasing curve in the graph is observed, matching the progress of the file distribution: initially only the RSUs have the file's chunks, but as they spread, more and more simultaneous transfers between OBUs occur. At 11:45PM every container is already shutdown and the test is finished. Total RAM usage has hovered around 6GB.

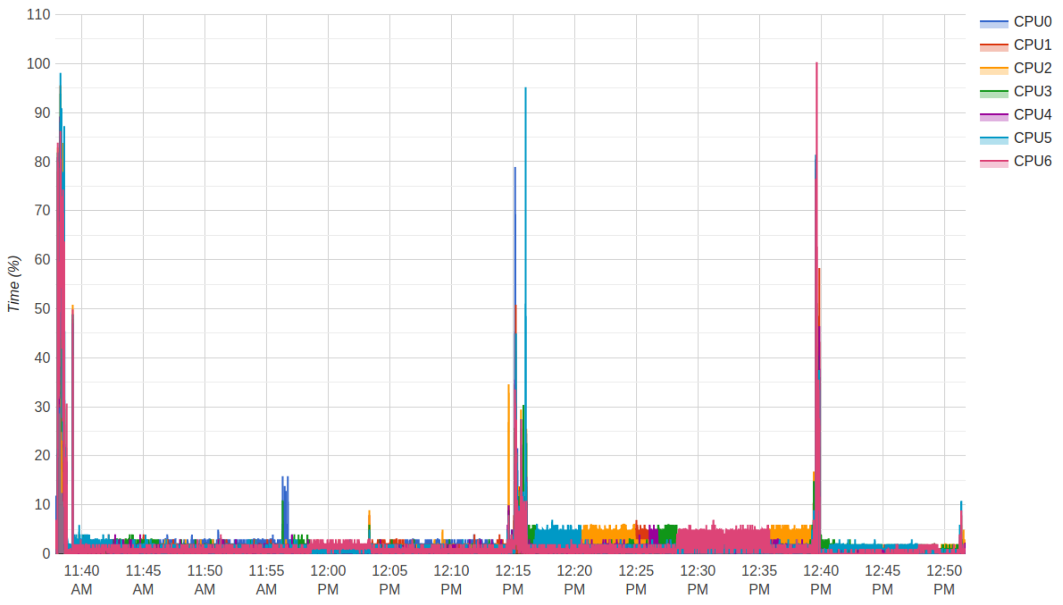


Figure 5.5: CPU usage of EmuCD running BitTorrent on 100 nodes, 50MB file.

Figure 5.5 and Figure 5.6 are the baseline of what to expect when running 100 nodes with BitTorrent under "regular" mobility in a normal VANET. Figure 5.7 and Figure 5.8 also represent 100 nodes with BitTorrent but in the stress test instead.

IPFS

IPFS is "heavier" than BitTorrent regarding both CPU and RAM usages. Figures 5.9 and 5.10 show respectively the CPU and RAM usage of 70 nodes with IPFS distributing a 50MB file. Until approximately 4:10PM there is a high CPU usage due to the Docker containers being booted and the required resources being allocated. The hardware's performance during the test is highly conditioned by the protocol and the network activity: a sparse network will result in a low resource usage because there are less contacts between nodes than in a dense network.

Then the CPU locks at 100% for the duration of the emulation: this is an indication that the hardware resources are not enough for this scenario. It is important to keep CPU usage under the maximum, as it influences results. RAM used has hovered at around 10GB.

Final remarks

These results show that, with enough hardware resources, every scenario is possible. Excluding the above mentioned edge case and focusing on the BitTorrent example, in a *normal* VANET, at 35MB per container and with the observed CPU usage, the deciding factor is the RAM. This means that, if 35GB were available to be used by the emulator, 10x more nodes could be launched and emulated: 1000 nodes instead of 100 nodes, with 35GB RAM and 50% average CPU usage, while maintaining the same CPU. With other more complex protocols, the limiting factor may not be RAM but CPU instead. The hardware requirements must be adapted according to the specific protocol and the scale of the emulation.

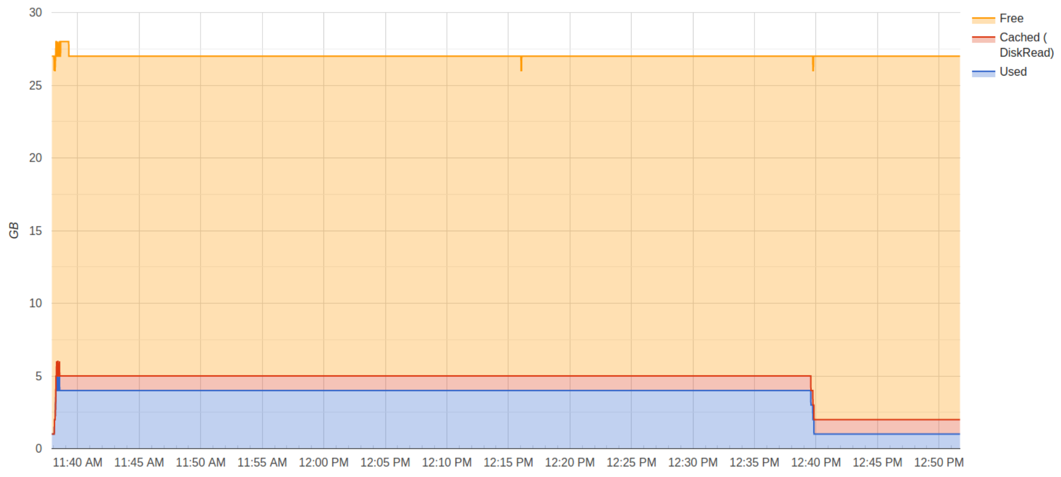


Figure 5.6: RAM usage of EmuCD running BitTorrent on 100 nodes, 50MB file.

5.4 Summary and Conclusions

In this chapter, the motivation for a different emulator was discussed, followed by presenting the new emulator's architecture and implementation details, as well as how easy it is to integrate with new protocols/software (if the protocol is ready to run in real hardware, it is ready to be run in the emulator).

The chapter ends by discussing the environment surrounding the emulator, both in the OS (some kernel parameters that had to be changed) and in the hardware that is required, depending on the software being tested. There are two important hardware factors to consider: RAM and CPU. RAM limits how many containers/nodes can be launched, generally independently of their activity (whether they have constant network traffic or not): 16GB are required to comfortably launch 200 nodes with BitTorrent but not for 70 nodes with IPFS, as expected since a single IPFS process has a much larger memory usage than a single BitTorrent client. CPU limits the emulation in three ways: 1) CPU usage must be under 100% (ideally under 50%) to consider a valid parallelism between containers; 2) high network activity causes a high CPU usage; and 3) related with the previous point, the amount of processing done by the protocol/software before sending each chunk/block naturally influences the CPU usage (a protocol with a simple logic/algorithm allows for the emulator to run more nodes than if it is a protocol with a very complex algorithm).

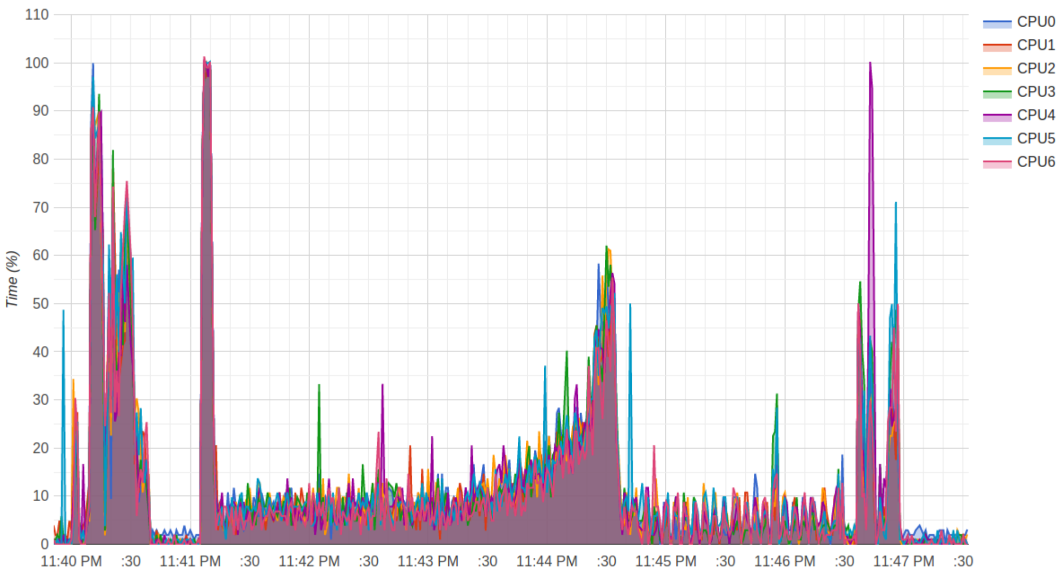


Figure 5.7: CPU usage of BitTorrent during a stress test, 100 nodes, 50MB file.

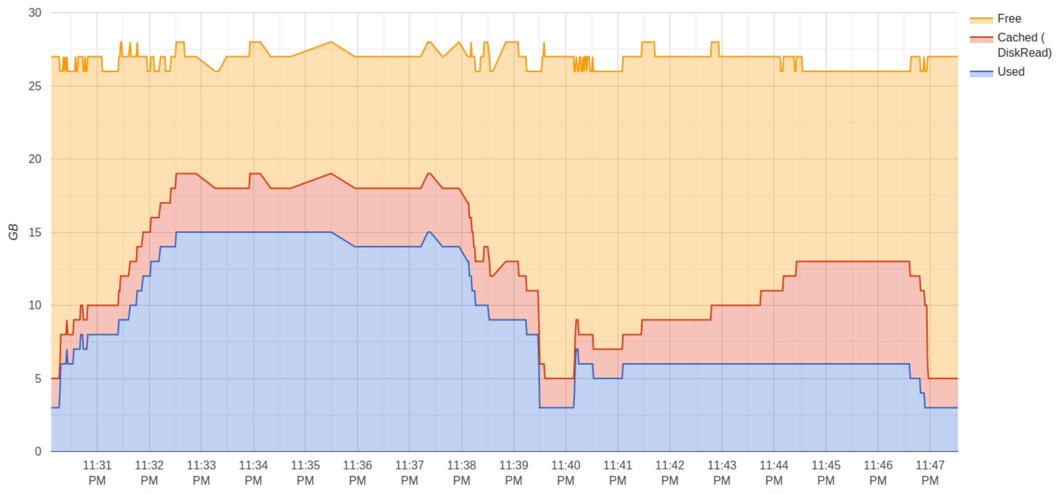


Figure 5.8: RAM usage of BitTorrent during a stress test, 100 nodes, 50MB file.

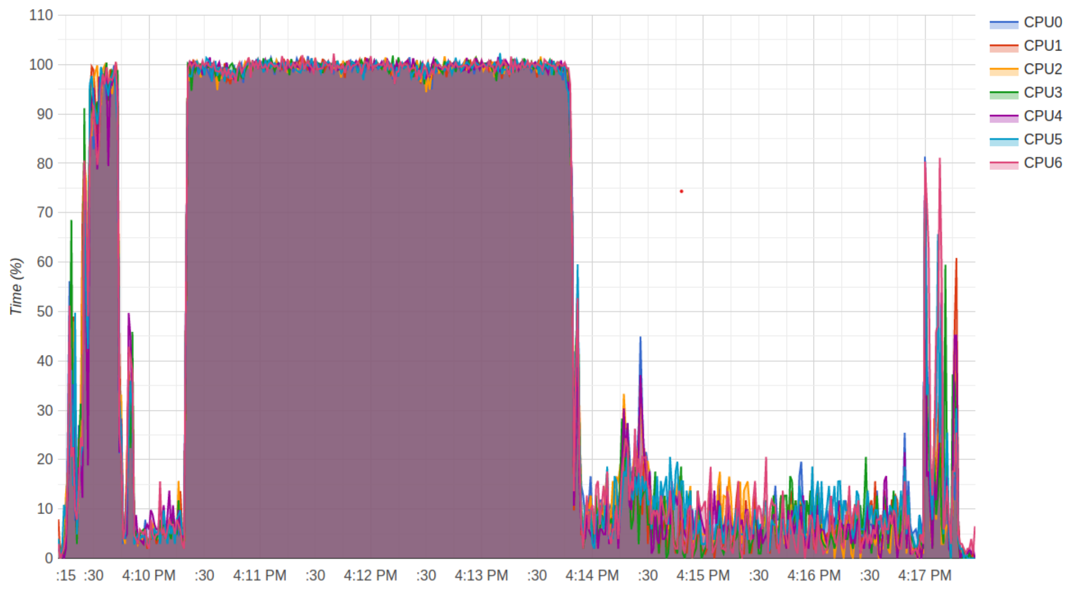


Figure 5.9: CPU usage of IPFS during a stress test, 70 nodes, 50MB file.

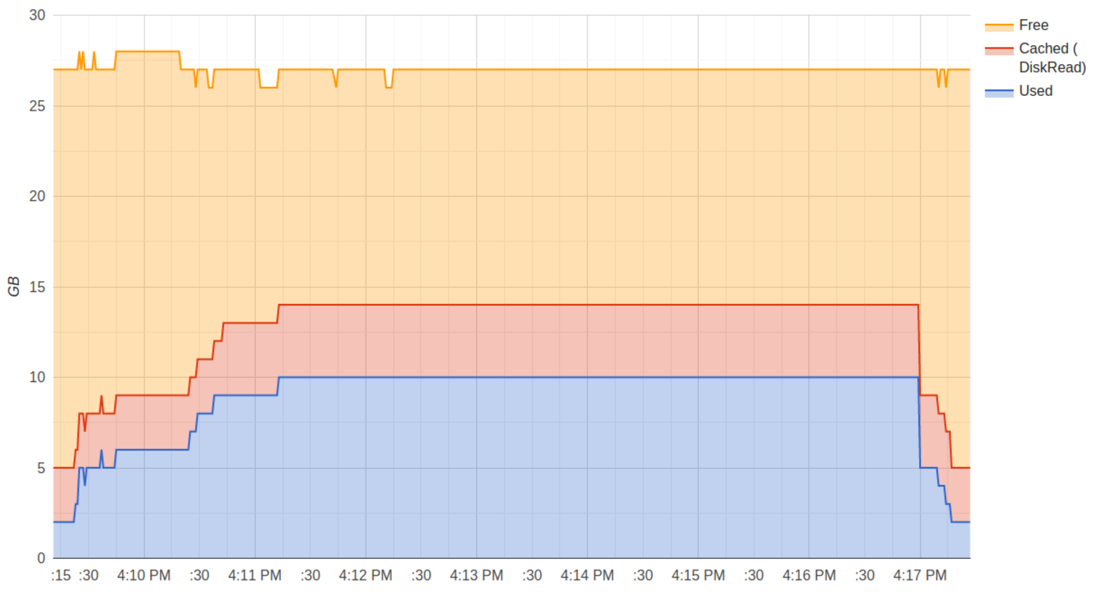


Figure 5.10: RAM usage of IPFS during a stress test, 70 nodes, 50MB file.

Chapter 6

Results - Emulated and Real

This chapter presents the results of running IPFS, BitTorrent and Sprinkler in the emulator with different scenarios (varying times, file sizes and in the case of IPFS, chunk sizes), highlighting their differences and main features. Moreover, it also presents the results of running IPFS in a real VANET. The goal of the emulator is to reliably test different protocols against one another, as it is very hard to do it in a production VANET. With this in mind, in the Veniam's VANET testbed, the goal is to validate IPFS running in real hardware with real vehicles and compare it with Veniam's current V2V protocol, Sprinkler. Notice that IPFS was never tested in VANETs and in real VANET networks.

This chapter begins by presenting the emulated scenarios and their results, moving on to the results of running IPFS and Sprinkler in Veniam's VANET testbed comparing them where/when possible. Finally, there is a comparison between some scenarios in EmuCD and the real tests, with a summary closing the chapter by discussing the most important conclusions to be taken about IPFS and its performance in a VANET.

6.1 Emulated results

For each protocol, multiple combinations of varying parameters were tested, using a dataset collected of over 300 nodes in Oporto's VANET [25], from 23:59 of May 9th, 2017 to 23:59 of May 10th, 2017. This dataset contains the mobility of the vehicles, and connectivity information between the vehicles and between the vehicles and the road side units. The following information details the parameters tested:

- **time of day** - different time intervals from the dataset were used, ranging from rush hour periods (9:00h-10:00h) to quieter periods (14:00h-15:00h or 4:00h-5:00h);
- **duration** - the majority of the time intervals tested were one hour (1h) long, but longer, seven hour (7h) time intervals, were also tested;
- **file size** - to test the impact of the amount of data being distributed (in the case of a single file) in the network, two file sizes were tested: 5MB and 50MB;
- **chunk size** - for IPFS only, the size of each block/chunk was changed in order to show the impact that this characteristic has in the overall throughput and content dissemination speed.

6.1.1 Time of day

This subsection presents 2 groups of graphs: the impact of testing different times of the day in a single protocol, and a comparison between the three protocols while varying the times of the day. Figure 6.1 presents the results of running IPFS in a 200 node network, disseminating a 50MB file from 9h-10h versus 15h-16h. The period 9h to 10h can still be considered part of the morning rush hour period, meaning that there is a higher traffic density and slower speeds than the "quieter" period of 15h-16h; slower speeds and higher traffic density result in longer connection times, as can be observed in the number of OBUs downloading 100% of the content in the first seconds when compared to the 15h period; overall, this results in a faster content dissemination in the 9h period than in the 15h period, 76% and 59% after one hour, respectively.

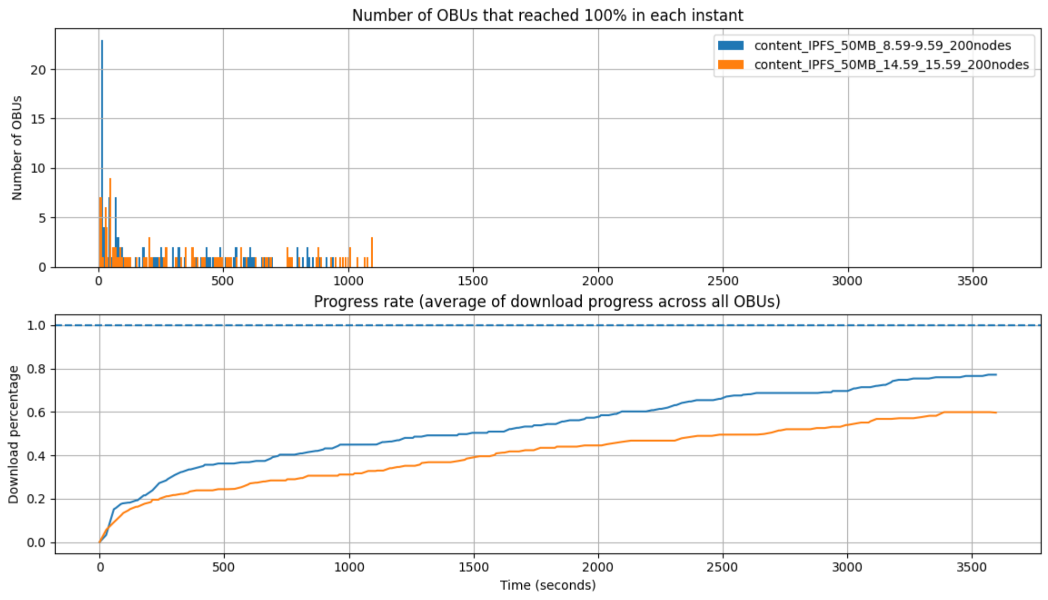


Figure 6.1: Comparison between 9h-10h and 15h-16h, using IPFS in 200 nodes, 50MB file.

This difference is noticeable across every protocol, a comparison that can be seen in Figure 6.2 (the first 20 minutes are in Figure 6.3) with IPFS in purple and orange, Sprinkler in green and brown and BitTorrent in blue and red. In every protocol, a faster content dissemination occurs in the 9h period. The most noticeable difference is with Sprinkler, where 60% of the total content is distributed in the 9h-10h period, whereas in the 15h-16h period only 35% is distributed on average.

Focusing on the first 20 minutes of the test, IPFS at 9h has the largest number of OBUs reaching 100% in the beginning; but after the first 5 minutes Sprinkler takes the lead on the content distribution. Sprinkler appears to take a higher advantage of node density when compared to IPFS, possibly because in Sprinkler chunks to be requested are chosen randomly, whereas in IPFS it is not as simple (see Section 3.2.4 for details on IPFS block transfer). IPFS at 15h, a time of the day with less traffic, performs better than Sprinkler for the duration of the test.

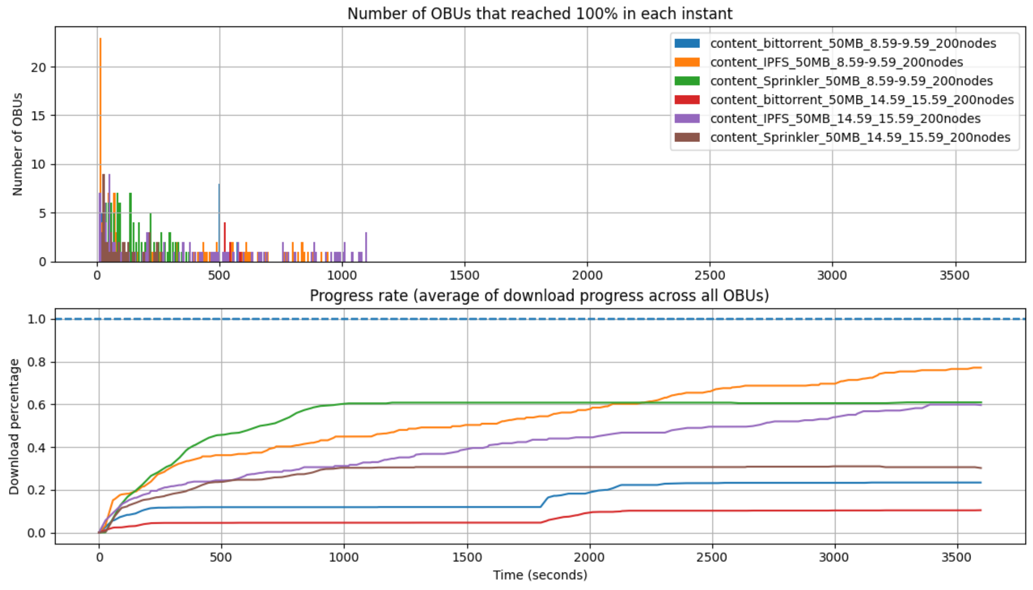


Figure 6.2: Comparison between 200 nodes, 50MB file, 9h to 10h against 15h to 16h, using IPFS, Sprinkler and BitTorrent.

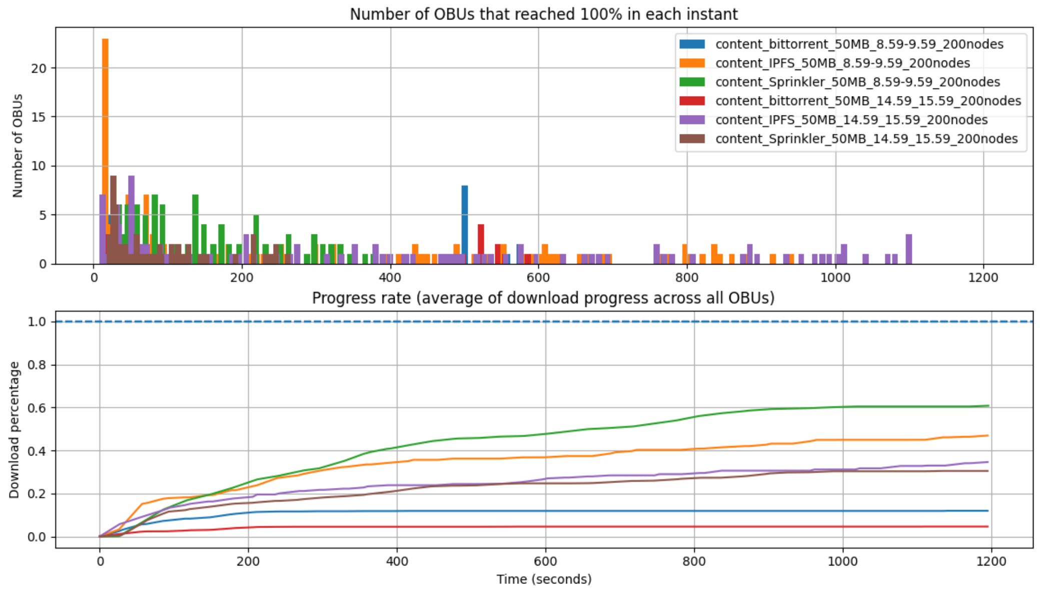


Figure 6.3: Comparison between 200 nodes, 50MB file, 9h to 10h against 15h to 16h, using IPFS, Sprinkler and BitTorrent (zoomed in the first 20 minutes).

6.1.2 Duration

As previously mentioned, the vast majority of the tests performed in the emulator had one hour in length, mainly due to the real-time aspect of the emulator, which requires a long run time. In many scenarios however, despite being enough to compare protocols and the emulation's parameters, one hour is not enough for the content to be distributed in its entirety. With this in mind, the three protocols were tested for 7 hours, from 4h to 11h in order to provide a wider view.

Figure 6.4 shows IPFS, Sprinkler and BitTorrent running on 200 nodes from 4h to 5h, acting as a "before" in the comparison between running a test for 1 hour or 7 hours. There are two behaviours worth noting. First, BitTorrent barely has any progress, not only because the protocol itself is not well suited to run in a VANET, requiring a long connectivity window (at least 20 seconds), but also because the mobility at 4h is quite sparse, especially when compared to a rush hour period (or even a quieter period but during the afternoon): previously in Figure 6.2, BitTorrent had a minimum of 10% content distribution after one hour, comparing to the 5% in this case. The second behaviour worth nothing is in Sprinkler, which after 1000 seconds does not progress any further, suggestive of an issue in the emulation, hardware or the software itself. Sprinkler's choices in packets to send or ask is very basic when compared to IPFS, and upon inspecting the logs and verifying that everything was working as intended, it was concluded that this 4h period is a worst case scenario for Sprinkler: after 1000 seconds, every OBU that came in contact with another did not have a block/chunk to offer that the other OBU did not already have.

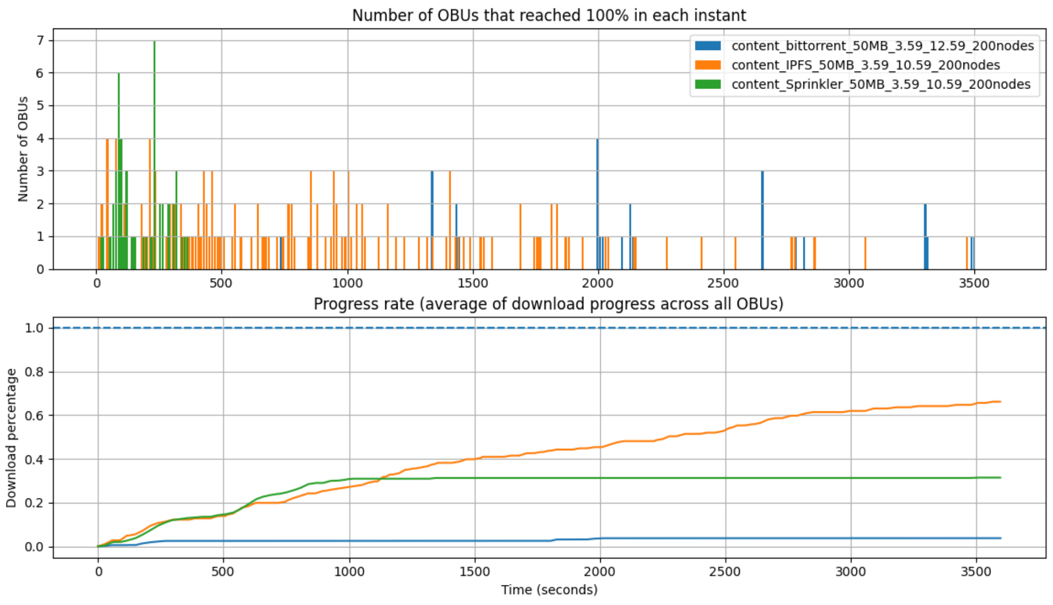


Figure 6.4: Comparison between 200 nodes, 50MB file, from 4h to 5h, using IPFS, Sprinkler and BitTorrent.

Figure 6.5 shows the results of running IPFS, Sprinkler and BitTorrent in a 200 node network, distributing a 50MB file from 4h to 11h. In this scenario, 90% of the content is distributed using IPFS after 7h, 41% with BitTorrent and 32% with Sprinkler. By increasing the duration of the test, the average download progress tends to increase: the graph shows that as time advances, the average of the download progress across all OBUs gets closer to 100%, reaching it if given enough time. This is a characteristic of a P2P download where a large amount of blocks are received in the first part of the download (when a node does not have any block, the probability of a neighbour having a new block is 100%), decreasing as more and more blocks are received (when a node is missing only 10% of the blocks, the probability of a neighbour having a new block is low).

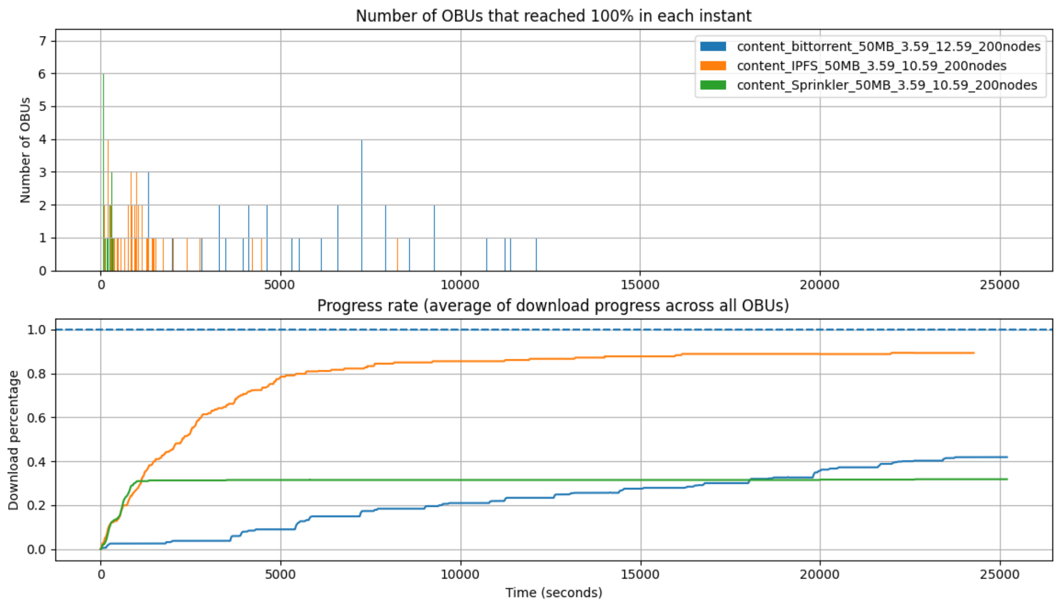


Figure 6.5: Comparison between 200 nodes, 50MB file, from 4h to 11h, using IPFS, Sprinkler and BitTorrent.

6.1.3 File size

Two different file sizes were used in the emulation runs: a 5MB file, symbolic of a small file, and a 50MB file, meant to be representative of a software update (for example the IPFS binary is close to 50MB in size). The files used were PDF files, but the file format or contents are not important as they are divided into blocks/chunks by all three protocols. The file contents could be worth mentioning only in case the file had multiple equal chunks: instead of having 208 unique chunks for the 50MB file there were only 50 unique chunks for example, as the remaining 158 were equal. For reference, the 5MB file was divided by IPFS into 19 chunks with the default size of 256KB, whereas the 50MB file was divided by IPFS into 208 256KB chunks (as previously mentioned).

Figure 6.6 shows the results of running IPFS, Sprinkler and BitTorrent from 15h to 16h in 200 nodes, distributing the 5MB file. Sprinkler is the leader until almost the end, reaching 60% after a hour, alongside IPFS; BitTorrent falls short of 20% download progress. In this 5MB scenario, the difference in distribution speed between Sprinkler and IPFS is noticeable, with Sprinkler reaching 58% after 1000s whereas IPFS and BitTorrent were still at 32% and 12%, respectively.

Distributing the 50MB file in the same time period (15h to 16h) shows a different result, which can be seen in Figure 6.7: after 1000s both IPFS and Sprinkler stand at around 30% of content dissemination and, at the end of the hour, IPFS and Sprinkler distributed 60% and 31% of the total content, respectively. This difference between the 5MB and 50MB results imply that lower file sizes are favourable for Sprinkler and overall allow a larger percentage of the content to be distributed in the same time interval (30% of the 50MB test with Sprinkler in the first 1000s, 58% of the 5MB test with Sprinkler in the first 1000s), possibly because of Sprinkler's 1MB chunk size (256KB by default on IPFS) that can be advantageous when the network is only exchanging the same 5 or 6 chunks. This advantage appears to be gone when dealing with a higher number of chunks.

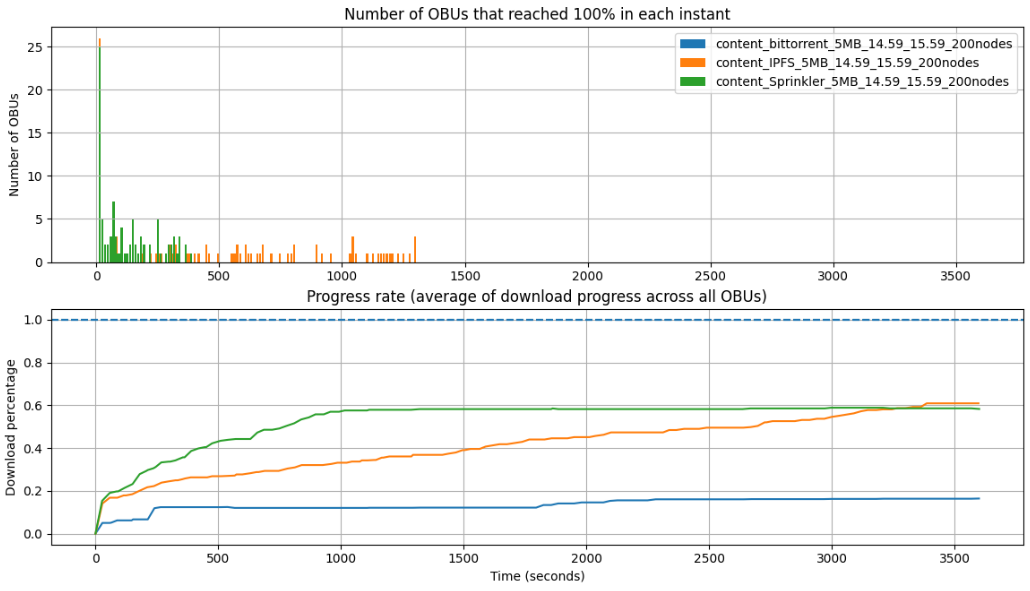


Figure 6.6: Comparison between 200 nodes, 5MB file, from 15h to 16h, using IPFS, Sprinkler and BitTorrent.

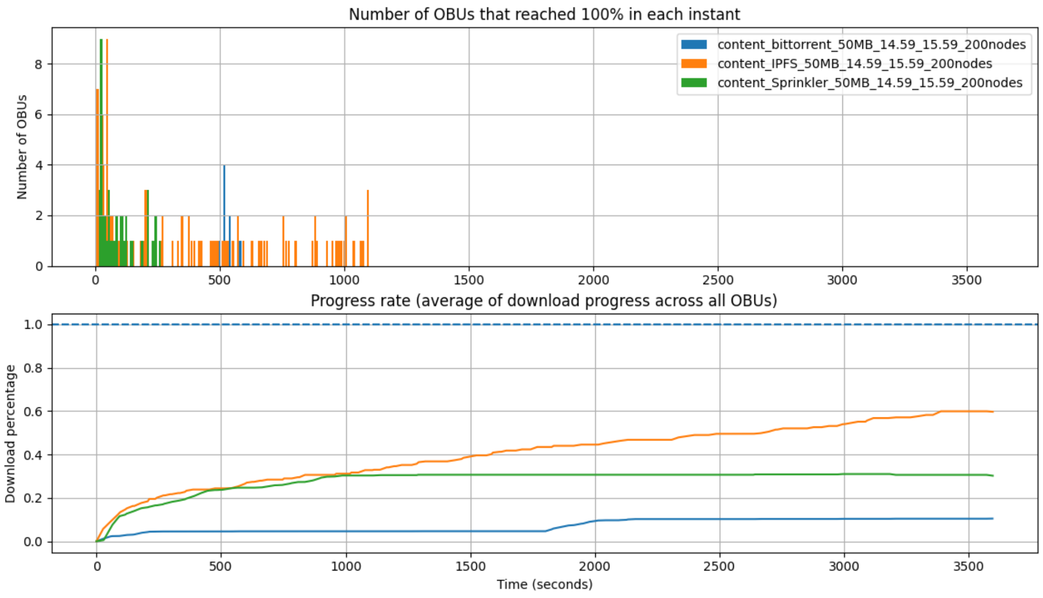


Figure 6.7: Comparison between 200 nodes, 50MB file, from 15h to 16h, using IPFS, Sprinkler and BitTorrent.

6.1.4 Chunk size (IPFS)

In IPFS, the size of the chunks can be easily changed from the default `size-262144` when adding a new file (or even using a variable chunking algorithm like Rabin-Karp), by using the flag `-chunker`. Every power of two was tested, starting from 1KB and going up until the maximum allowed value of 1MB.

Figure 6.8 shows the impacts of the different chunk sizes when running IPFS in a 200 node network, from 15h to 16h, distributing a 50MB file. The graph shows that some sizes perform much worse than others, namely 1KB and 2KB; in the opposite spectrum, among those that perform the best, there are several options close to each other, so Figure 6.9 only shows the two best, two worst and the default size for comparison, while restricting the viewing window to the first 20 minutes of the test. **Note:** *the colors of the same chunk sizes in Figure 6.8 and Figure 6.9 may be different.*

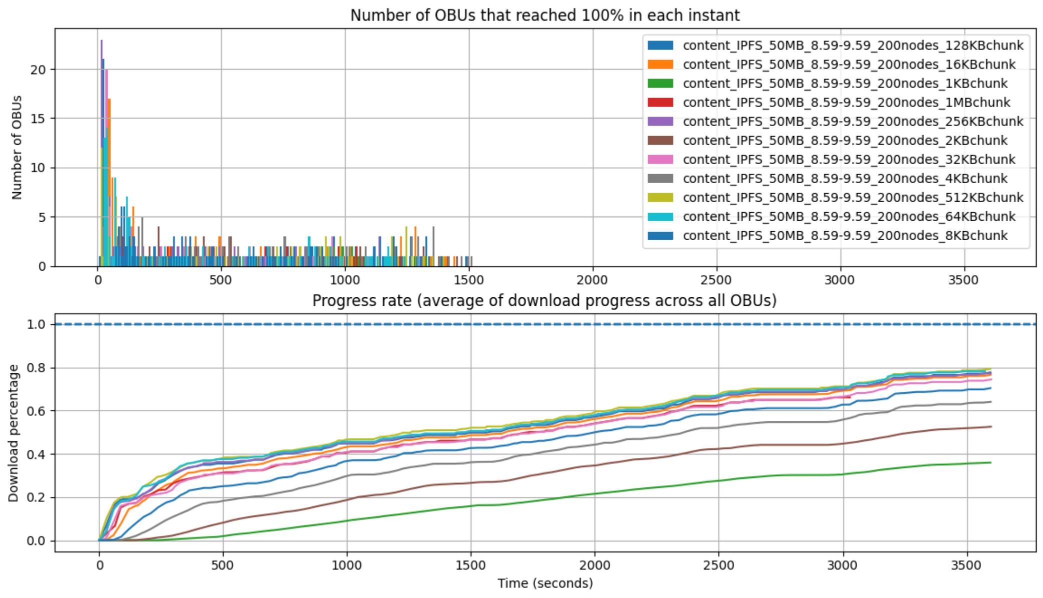


Figure 6.8: Comparison between several chunk sizes when using IPFS from 15h to 16h in a 200 node network, 50MB file.

Although 512KB is only slightly better than the default 256KB, the improvements are minimal and there is no issue in staying with the default value. However, this is an important factor and should not be modified without careful consideration, as shown by the differences in progress rate when using 1KB chunks and 256KB: after a hour, almost 80% of the content was distributed using 256KB chunks, whereas less than 40% was distributed when using 1KB chunks in the same period.

6.2 Emulator vs VANET

This section presents the validation of EmuCD against the VANET using IPFS. Despite the OBUs running a custom version of OpenWRT and EmuCD's Docker containers Ubuntu, it is exactly the same IPFS code/software running on both: what runs in the real boards is what runs in the emulator.

Veniam was responsible for deploying a VANET in Oporto that consists of over 400 public buses and over 40 RSUs. Each vehicle carries an OBU that serves the production network, providing services such as WiFi access to the passengers. In some vehicles, however, there is

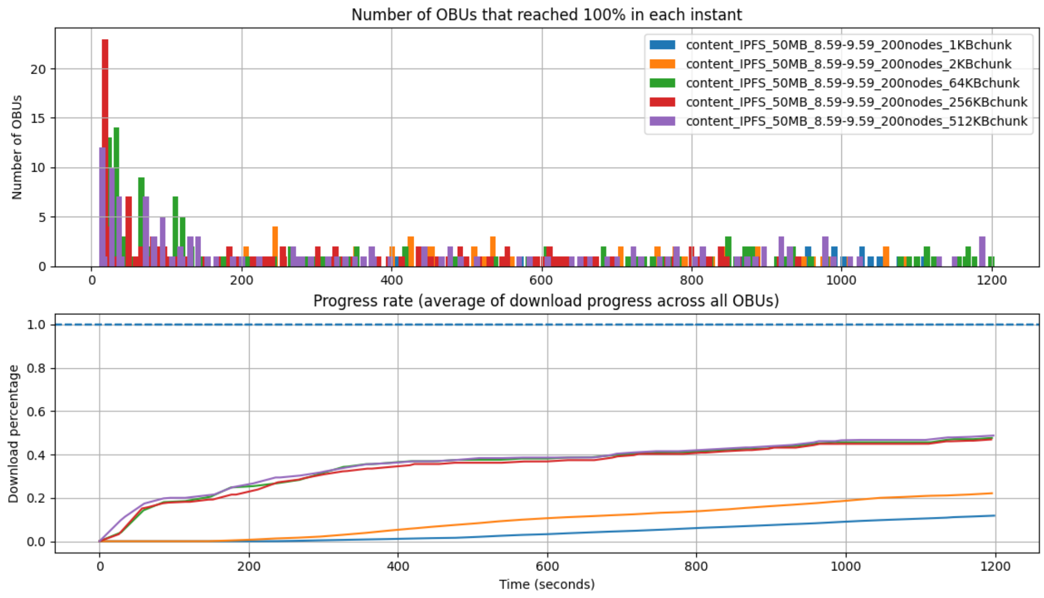


Figure 6.9: Comparison between the default, two best and two worst chunk sizes when using IPFS from 15h to 16h (only first 20 minutes) in a 200 node network, 50MB file.

another OBU for testing: the VANET comprised of these testing OBUs is the testbed network, used for validation of new software, testing and research, as is the case of the work presented here.

The RSUs are not the same type of boards as the OBUs, they are just public WiFi access points providing internet connectivity to the VANET. As such, the RSUs cannot be used in the same way the OBUs can, so whenever there is content to be transferred from the Cloud to the network it does not go to an RSU: instead, one or more OBUs are chosen as seeders that will distribute the content to the remaining OBUs.

The bus fleet is an ordinary fleet of vehicles and as such there are schedules, itineraries and vehicle rotation - not all vehicles are simultaneously on the road, nor they are locked to an itinerary (bus no. 123 can serve line A one day but line B in the following day).

For this validation the *Progress rate* is considered, which is the average of the content's download progress across all OBUs over time, when running IPFS in a real VANET and in the EmuCD. Looking at the results in the previous section, 3 different use cases can be considered for the validation with 1 seeder and 4 OBUs: **Scenario 1)** a fast content dissemination, where the nodes are close neighbours (not only spatially but also temporally) with contact times long enough for data transfers to occur; **Scenario 2)** a medium-speed content dissemination, where the nodes are occasionally in each other's neighbourhood, but contact times are not enough for the file to be fully transferred between two OBUs, requiring multiple contacts and taking potentially hours until the data is distributed across every node; **Scenario 3)** long content dissemination, where some OBUs are very far away from nodes with useful data (eg. non-overlapping bus routes), and it can take a full day or longer for the content to reach every node.

Because it is not possible to use 200 nodes in the real VANET, the number of nodes running in the emulator was reduced from 200 to 5. This constitutes a less interesting scenario regarding mobility, but allows for a comparison between the real results and the emulator's.

Figure 6.10 compares the progress rate of a IPFS file of 50MB in Scenario 1.

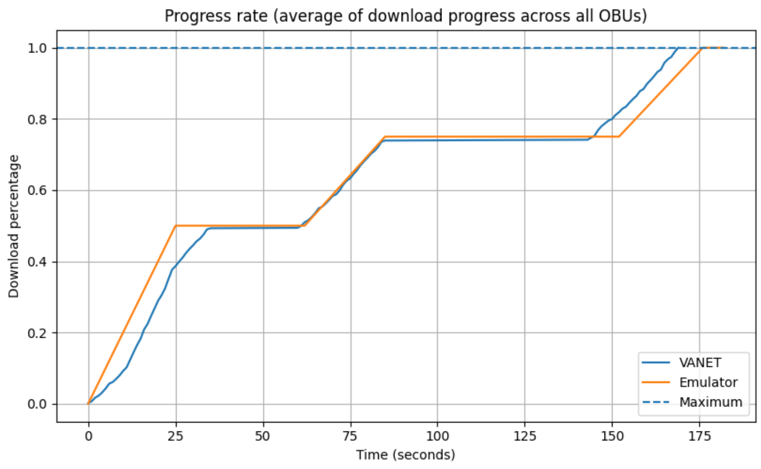


Figure 6.10: Scenario 1 - Comparison of download times of a 50MB file dissemination in a 5 node VANET using IPFS vs EmuCD using IPFS.

The graph shows that the results are similar, with only a few seconds of difference in some parts and fully distributing the content to every node within 175 seconds, but there are some important observations to be made:

- the emulator graph only shows straight lines during data transfers, whereas the real VANET's graph has more granularity and variance - in the real VANET there is more than connectivity information, such as interference, leading to an inconsistent throughput, which ranged from under 0.5MB/s up to 2.8MB/s; EmuCD sets a 2MB/s upload and download fixed limit on the wireless interfaces, which was observed to be the average from initial lab tests with the hardware;

- the emulator graph reaches 50% progress 10 seconds earlier than the real VANET, it accompanies the VANET in the second dissemination period and falls 8 seconds behind the real VANET in the final dissemination period - this can happen because of several factors, but the most probable one is the connection establishment time, which is not negligible in ad-hoc networks, where connections are not established instantly. From the moment a node is physically discovered until a connection is established, a node must select the neighbour, switch to its IBSS, send a session request, switch to the session [73, 74], just to name a few of the things happening, which can result in several seconds lost in the connection period. The emulated scenario does not replicate this, it only adds/removes IP addresses from the OS's routing table.

Regarding Scenario 2, a representative set of downloads was selected from the results of running IPFS in the real VANET and compared against a similar situation in EmuCD (the ideal would be to have the full data trace from when the tests were ran in the VANET, and use that in EmuCD). Figure 6.11 shows the comparison between IPFS distributing a 50MB file in a 5 node real VANET and in an emulated 5 node VANET in Scenario 2. The emulator has very close results when compared with the VANET albeit with lower accuracy, namely in the first 300 seconds where in the real test the transfer slows down, but in the emulator it keeps going, arriving at 50% content dissemination earlier than it should. In both cases, the content is fully distributed after 3500 seconds, taking 20 times longer than Scenario 1 due to the long periods without communication, and also because of the existence of periods where two nodes do not "see" each other for too long.

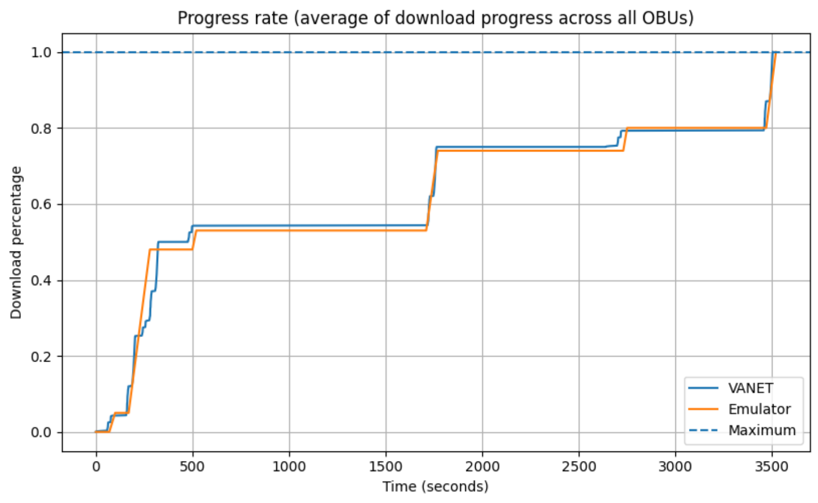


Figure 6.11: Scenario 2 - Comparison of download times of a 50MB file dissemination in a 5 node VANET using IPFS vs EmuCD using IPFS.

Finally, there is Scenario 3, happening when OBUs are either too far apart from each other or when they are close and connection times are not long enough. Figure 6.12 shows the results of Scenario 3 with IPFS disseminating a 50MB file in a 5 node VANET – both in a real network using IPFS to distribute a 50MB file, and in a similar 5-node emulated VANET – with IPFS distributing a 50MB file as well. This type of dissemination is characterised by extremely long periods without communication (more than one hour), as shown by the fact that after more than 3 hours, only 40% of the content has been distributed. As in the previous cases, EmuCD shows the same results as the real VANET, except this time in the middle transfer with the OBUs in the real scenario, probably due to an interference only observed in the real scenario, despite having connectivity (in the emulator, data was transferred as it was supposed to) the download progress did not occur as a quick continuous event. This happened while the OBUs were in range of each other, probably because of an error, causing them to not transfer enough information, miss the time frame and only cross paths 30 minutes later. It is an example of the unpredictability of the real world, where a temporary hardware/software error or an overloaded CPU/storage (due to other background tasks) can prevent the software from performing as expected.

6.3 Real Testbed Results

This section contains the testbed's results and analysis. The following subsections show a direct comparison between IPFS and Sprinkler regarding download times, throughput, duplicate packets, download progress, peers' visibility and GPS information. Both IPFS and Sprinkler ran on the testbed, although not simultaneously. Running these protocols separately means that the download times and throughputs are accurate, but the mobility is different.

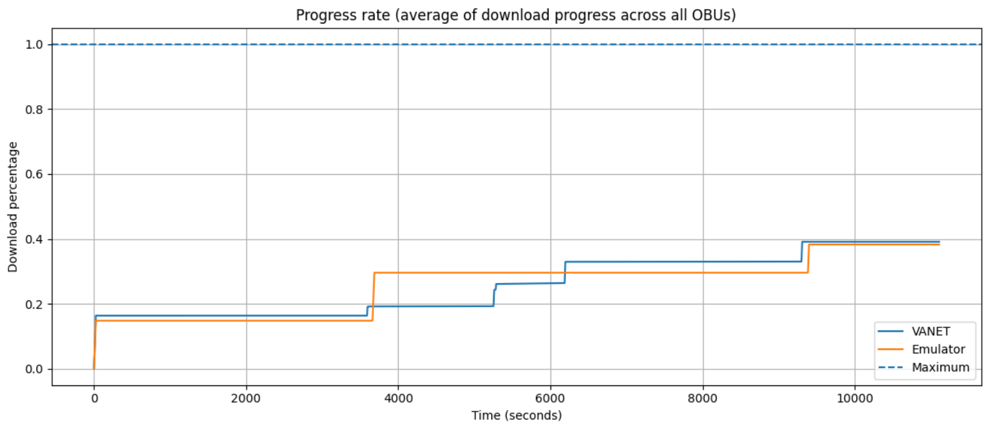


Figure 6.12: Scenario 3 - Comparison of download times of a 50MB file dissemination in a 5 node VANET using IPFS vs EmuCD using IPFS.

If they had ran at the same time, it could be seen how both protocols would perform under the same circumstances, but because the V2V interface would be shared between them, the download times would be affected. It was given priority on obtaining accurate download times for each technology, as EmuCD already compares different protocols under circumstances that are exactly the same across different runs.

6.3.1 Setup

IPFS ran in this real network for 5 days, from 17-07-2020 until 22-07-2020, and Sprinkler also ran for 5 days, from 22-07-2020 until 27-07-2020. Because running IPFS in a real VANET is different than running it in a VANET in a controlled environment, lab or emulator, there were some unforeseen issues during the first days: in the initial configuration and before the 5 days of run time, IPFS was announcing on the V2V interface every address it had, including the cell address, which meant that when OBU A entered the neighbourhood of OBU B, the IP addresses were exchanged, as well as some blocks, but when it left (and the OBUs could not longer "see" each other using V2V), the transfer would continue using the cellular IP address. It took a while to notice that content was being distributed unexpectedly fast and fix the issue. Some problems similar to this one led to the "loss" of a few testing days.

Deploying files to the OBUs and managing configurations was done by accessing the public IP address of the cell interface using SSH. This address is not static, Veniam's management dashboard was used to check if the OBUs were online and ready for access.

The IPFS test consisted of disseminating a 50MB file from 1 OBU to 4 OBUs; IPFS storage and caches were manually reset after observing that every OBU had completed the download. If after 24h the file had not been disseminated across every OBU, the IPFS storage and caches were manually reset, effectively starting a new test.

The Sprinkler test consisted of also disseminating a 50MB file from 1 OBU to 4 OBUs; Sprinkler was manually restarted after observing that every OBU had completed the download. If after 24h the file had not been disseminated across every OBU, Sprinkler was manually restarted, starting a new test.

Not every OBU had two wireless interfaces (one for V2I, another for V2V); therefore, in order to assure consistency, every board was treated as having only one wireless interface, used in V2V communication. Combined with the inability to access RSUs, as they were WiFi access points (despite not being possible to deploy the content to the RSUs, they can be used to download it from the Cloud), a OBU had to be used as a seeder, so that is why from the 5 OBUs available, only 4 were true OBUs. Further details regarding the real VANET were already presented in Section 6.2.

6.3.2 Results

This section presents the obtained results. It starts by giving some context regarding the positions of the OBUs during the tests, and some relevant metrics like throughput, bytes received, duplicates and download information. An emphasis is given on IPFS and its results, as one of the main goals of this work is to assess how it behaves in a VANET.

GPS

The GPS positions of each OBU were registered during the tests. Being the most frequent event being logged to storage (every second), the number of GPS events is the best indicator of how long each OBU was running.

Considering IPFS, the OBU that was active for the longest time had 206 hours of logs and 720,000 GPS positions, whereas the OBU that was active the least amount of time had 61 hours of logs and 210,000 GPS positions. Each OBU is inside a bus; when this bus is stopped for a long enough period of time, the OBU shuts down to prevent unnecessary usage of its battery.

A total of 2173 events were registered (blocks received), 425 when the OBU was without GPS and 1748 with GPS (because of atmospheric conditions, satellite visibility or temporary hardware issues, unfortunately it is not always possible to have constant GPS information).

Concerning Sprinkler, the OBU that was active for the longest time had 111 hours of logs and 400,000 GPS positions, whereas the OBU that was active the least amount of time had 41 hours of logs and 110,000 GPS positions. A total of 987 events were registered, 574 with GPS and 413 without GPS.

When testing both IPFS and Sprinkler, a few geographic areas with a high percentage of the downloads were identified - probably due to a higher vehicle density or slower traffic. As these GPS positions represent sensitive data, unfortunately they can not be shared here.

Download times

IPFS registered 48 individual downloads and Sprinkler registered 6. This means that, if the network only had 2 OBUs, in the IPFS tests each board downloaded the 50MB file 24 times on average (48/2 = 24), and in the Sprinkler tests each board downloaded the 50MB file 3 times on average (6/2 = 3). When looking at this difference in numbers, it should be taken into account the data from the previous section, showing that, when Sprinkler was being tested, the OBUs were offline/stopped more often.

Download times were measured as the time between the first and last blocks. Table 6.1 shows the download times of IPFS and Sprinkler, which can also be seen in Figure 6.13. The average time for an OBU to download a file from one or more OBUs is much lower in IPFS than in Sprinkler, but it is not solely representative of the protocol's performance: not only were the mobility conditions different when both Sprinkler and IPFS were being tested, but it should also be taken into account the low number of downloads registered in Sprinkler, as they are not enough to establish a definitive conclusion and verdict when comparing both protocols.

Table 6.1: Download times (s) between the first and last block (50 MB file).

	Average	Median	Std	Min	Max
IPFS	208.79	38.0	661.69	19	3430
Sprinkler	17421.6	960.0	24676.53	276	64250

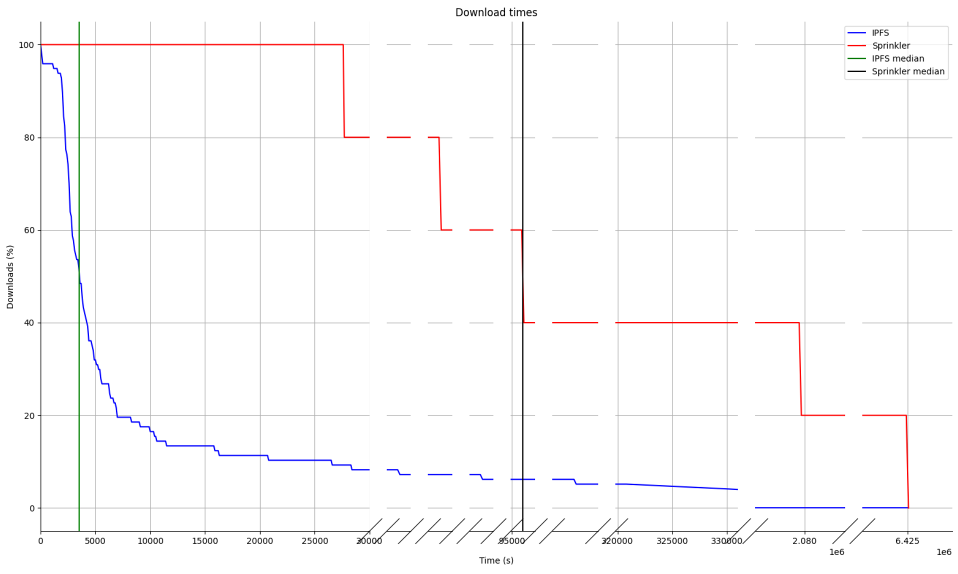


Figure 6.13: Graph of the download times (s) between the first and last block (50 MB file).

Throughput

Throughput was calculated by dividing the data received (bytes) by the download time. In Sprinkler however, because of the long download times (max throughput was 0.19 MB/s) and to better compare the data, a *fair throughput* was also calculated by considering only intervals where time between chunks is less than 4 seconds.

Table 6.2 shows a comparison between the two protocols: Sprinkler reaches higher average, median and minimum download throughput than IPFS.

Table 6.2: Download throughput, calculated dividing data received by download time (units in MB/s).

	Average	Median	Std	Min	Max
IPFS	1.57	1.51	0.85	0.02	2.82
Sprinkler (fair)	2.04	2.16	0.53	1.30	2.59

Bytes received and duplicates

In the graphs regarding bytes received, presented below, 100% represents the amount of bytes that are supposed to be received (eg. 50MB: 54277586/54277586 bytes). IPFS never had a reception ratio of exactly 100%, because there were always some duplicates received, so two main scenarios can be identified:

- An OBU is able to download everything that it needs and receives duplicate blocks (eg. 105% or 110% of the bytes);
- An OBU can only receive most of the file, but did not get the chance to download it fully. This is a frequent scenario, which is why IPFS's average percentage of bytes received is 93.3%.

Figure 6.14 shows the distribution of the percentage of bytes received, both in IPFS and Sprinkler. During the 5 days using IPFS, 2859.68 MB of data were downloaded, and using Sprinkler 258.82 MB of data were downloaded.

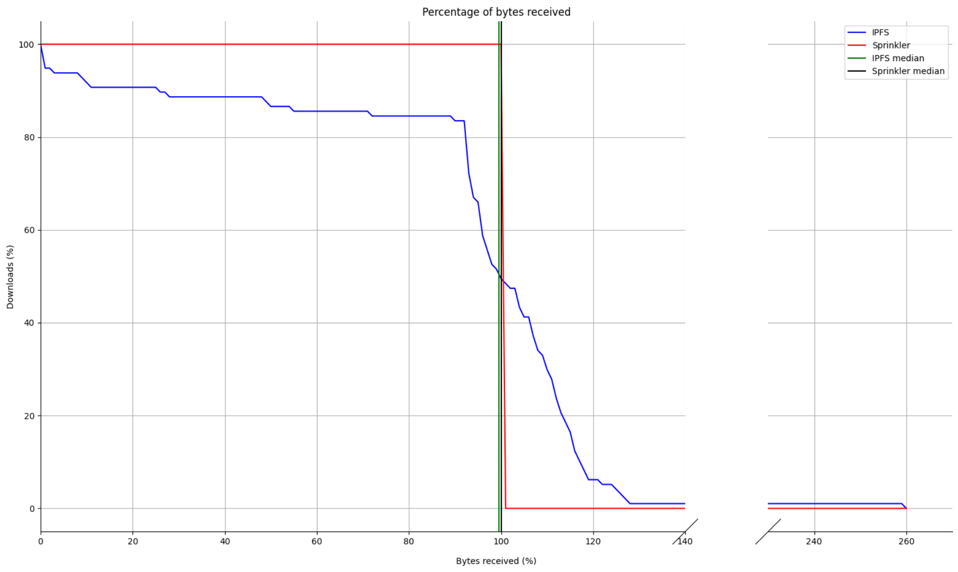


Figure 6.14: Percentage of bytes received in a download (IPFS in blue vs Sprinkler in red).

Figure 6.15 shows the percentage of duplicate blocks in the IPFS downloads. Duplicates are the extra chunks/blocks received. For example, if there are 100 unique chunks and one of them is received extra twice (3 times in total), the percentage of duplicates is 2%. Sprinkler is not in the graph: because of its architecture, there are never duplicate chunks being received, whereas in IPFS this can and does happen. The median percentage of duplicate blocks was 10%, with the graph showing that 60% of downloads had 5% or more of duplicates and 30% of downloads had 18% or more of duplicates. Another way to look at this is that 30% of downloads had less than 5% of duplicates and 70% of downloads had less than 18% of duplicates. There were some edge cases where over 90% of the blocks received were duplicates, but as it only happened once the cause was not identified.

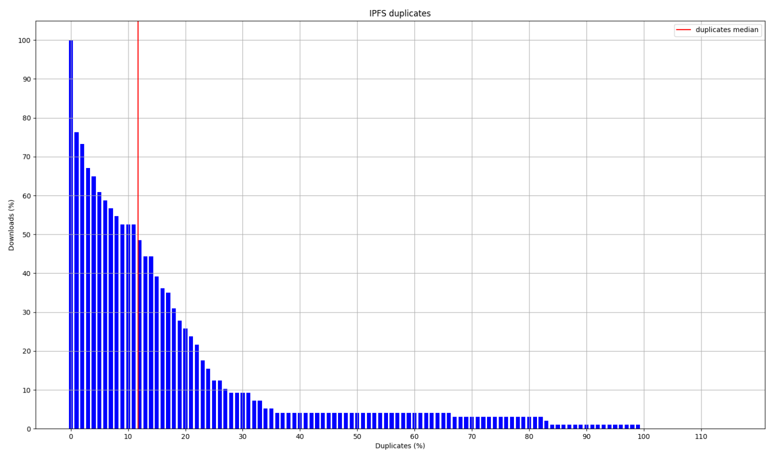


Figure 6.15: Percentage of IPFS duplicate blocks in a download.

Peers in a download

Because the network has 5 nodes, an OBU can download a file from a single OBU, from 4 neighbours over time or anything in between. Table 6.3 shows a comparison between IPFS and Sprinkler regarding the number of nodes that contributed to a download during the tests.

During the IPFS tests, on average an OBU downloads data from 2 other OBUs and in some cases, 3 OBUs contributed to an OBU download. Because it is very hard to ensure equal testing conditions in a real VANET, when Sprinkler was tested on average an OBU downloaded data from 1 other OBU, occasionally using 2 OBUs.

Table 6.3: Number of peers in a download.

	Average	Median	Std	Min	Max
IPFS	2.27	2.0	0.6	1	3
Sprinkler	1.4	1.0	0.49	1	2

6.4 Final remarks

This section compared IPFS against Sprinkler and BitTorrent in an emulated context, based on datatraces from a real VANET, discussing the impact that the time of day, content size and chunk size can have on the time it takes for content to be distributed to every node in the network. Then, IPFS was tested in the real world in a 5 node VANET against Sprinkler, Veniam's V2V protocol, showing that IPFS has good performance in a vehicular network, by not only looking at metrics like throughput (1.5 MB/s of throughput on average, going as high as 2.8 MB/s) but also comparing it to a reference V2V protocol, which is something that has never been done before.

IPFS is then used to compare EmuCD with the real VANET, in similar scenarios taken from the tests, showing that the emulator has the potential to be a great platform for developing and testing protocols for use in VANETs

The tests also showed a difference in performance between the two protocols, which can be attributed to the differences in network topology: IPFS ran from Friday (17-07) to Wednesday (22-07) and Sprinkler ran from Wednesday (22-07) to Monday (27-07), which despite having Saturday, Sunday and Monday in common, the vehicles' schedules and itineraries may not be the same in both tests. It should also be noted that in IPFS one OBU had a chance to receive blocks from 3 OBUs when downloading a file (not necessarily simultaneously), whereas Sprinkler did not, so that can also have an impact.

Chapter 7

Conclusions and future work

7.1 Conclusions

This work began by explaining concepts relevant in understanding IPFS and its context, from the simpler content distribution with a single server, all the way to CDNs and P2P. Then VANETs were introduced, presenting important works along the years regarding content distribution and the challenges this type of network encompasses, such as high mobility and dynamic network topology, limited bandwidth or frequent disconnections. Some characteristics, like anonymity, can be both an advantage (maintains the privacy of the nodes) and a disadvantage (susceptible to attacks such as Denial of Service).

As it is difficult to test and evaluate new protocols using real VANETs (inconsistent mobility and behaviour, difficult setup, impossible to repeat a test under exactly the same conditions), researchers have been using simulators instead, since they provide reliable, deterministic results when testing a network. Several existing solutions were mentioned in this work in order to better understand why none of them were suitable to test IPFS (or other software) in a VANET, and the need for a new emulator, EmuCD.

IPFS was then presented, a peer-to-peer distributed file system that provides a high throughput content-addressed block storage, heavily inspired by BitTorrent and Git. Nodes of this P2P network start by storing IPFS objects locally, but later connect to each other, while using a DHT for content routing, and knowing who has what. This decentralization allows IPFS to be used, not only as a typical filesystem, but also as a CDN, as a communications' platform or a versioned package manager for software, for example.

The goal of this work was to test IPFS in a vehicular network, check its performance, make adaptations if required/possible, and determine whether it can be used or not as a content distribution protocol in a vehicular scenario. Firstly, IPFS was deployed in OBUs in a lab context, without any movement, in order to understand what can be expected in a best-case scenario. It was determined that one aspect that has a great impact on its performance is the data storage medium used: a 50MB file takes 65 seconds to be downloaded if IPFS is running on the SD card, but just 39 seconds if IPFS is running on RAM, 40% faster. It was also determined that hashing can have a huge impact: using Blake2b instead of the default SHA2-256 can be up to 34% faster. Using TLS instead of SECIO, the default transport encryption protocol, also shows improvements (later versions use TLS by default). With a very powerful CPU, transport or hashing issues are no longer bottlenecks, only the network link; but, as often is the case, an OBU has energy and size constraints preventing it to use a desktop-grade

CPU, which is why these changes were made.

Then, the developed network emulator, EmuCD, was explained, adding further details to the motivation and explaining its architecture. The base impact of EmuCD, without any protocol, is an average of 30MB of RAM per container; this value goes up depending on the memory consumption of the software running on each container (if a protocol uses on average 50MB of RAM, each container will then use 80MB of RAM). Regarding CPU, results also show that it is capable of running hundreds of nodes (up to 200 were tested), depending on the hardware available, the protocol's load and the behaviour of the network. For a typical VANET topology, this limitation is fine, but it becomes troublesome if, for example, an OBU needs to send content simultaneously to 90 other OBUs.

EmuCD was compared to the real tests in scenarios as similar as possible (a perfect comparison would only be possible if the full GPS history of the real tests were available, which unfortunately was not the case), showing that EmuCD has performance close to reality if the dataset is accurate regarding the mobility, but lacks the finer details caused by wireless communications, since in the emulator, content is transferred at a fixed rate whenever there is connectivity.

IPFS was tested in EmuCD and compared against a BitTorrent client and Sprinkler, Veniam's proprietary V2V content distribution protocol. Parameters such as time of day, testing duration, file size and the chunk size of IPFS were changed: regarding the time of day, running a test in the 9h-10h period is different that running a test in the 15h-16h period, as there is more traffic density in the former time interval. This is shown in the results, since in a 200 node network using IPFS, the average of the download progress across all OBUs after 1h is 80% in the 9h-10h period, but only 60% in the 15h-16h period.

Regarding the testing duration, the longer the test, the more content is distributed (in every protocol), with IPFS reaching 90% of an average download progress across all 200 nodes after 7h, compared to 41% with BitTorrent and 32% with Sprinkler. Regarding file size, as expected the larger the file, the longer it takes for it to be distributed. In the 15h-16h interval, IPFS and Sprinkler distribute 60% of a 5MB file, but changing to a 50MB file shows IPFS also hovering around 60%, but Sprinkler falling down to 30%, very likely because of Sprinkler's larger chunk sizes (1MB vs 256KB on IPFS).

The impact of varying the chunk size was studied in IPFS, showing that the default 256KB value is acceptable (only 512KB is better, but by a small margin): under the same conditions, 80% of the content was distributed when using 256KB chunks, but only 40% in the same time when using 1KB chunks.

Finally, IPFS was validated in a real VANET. Download times, throughput, bytes received, duplicates and peers in a download were the evaluated metrics. The time for an OBU to download a 50MB file was, on average, 208 seconds with a maximum observed time of 57 minutes. As most of the metrics, this is highly dependent on the mobility of the VANET, not only regarding the position of the nodes, but also their speed, as slower speeds result in longer connections. The average throughput across all downloads using IPFS was 1.57 MB/s, with a maximum of 2.82 MB/s. Regarding bytes received and duplicates, during the 5 days of testing, IPFS was used to download 2.8 GB of data, and 30% of downloads had less than 5% of duplicates and 70% of downloads had less than 18% of duplicates. There were some edge cases where over 90% of the blocks received were duplicates, but as it only happened once, the cause was not identified. The final metric is the number of peers that contributed to a download: on average an OBU downloaded data from 2 other OBUs (not necessarily at the same time).

It should be noted that because IPFS was only tested using 5 nodes and for 5 days (not enough to gather conclusive and detailed results), the results should be interpreted not as a full in-depth analysis of IPFS (this requires much more data) but as an indication that the protocol works in a VANET and appears to be suitable for use.

Taking everything into account and after some adaptations, IPFS can be considered a good protocol for content distribution in a VANET, in the sense that: **1)** data transfers happen at the link's full capacity without noticeable bottlenecks (if it is a 20 Mbps wireless interface, data can be transferred at 20 Mbps); **2)** it provides an usable file system and DHT allowing for interesting applications and services to be developed on top of IPFS; **3)** it is not specific to a transport or communication technology, which can be good and versatile, but also prevents it from taking advantage of characteristics that could allow for faster wireless connections, in the case of a technology like 802.11p; **4)** it takes security in consideration, using TLS for transport encryption and allowing for private networks to be deployed.

7.2 Future Work

When it comes to VANETs, emulation strategies or content dissemination protocols, there is always room for further improvements. Some of them are discussed here:

- **IPFS hashing** - Blake2b was the new hashing algorithm used, but further improvements can still be obtained by using Blake3, when ready. This will enable not only adding content faster to IPFS, but also faster downloads as hashing is performed for every received block's validation.

- **EmuCD: improvements** - the emulator facilities can be expanded by automating the creation of containers and managing the execution of emulations; a more detailed, real-time dashboard can be developed. Moreover, the inclusion of new protocols can be further facilitated and documented.

- **EmuCD: validation** - the absolute validation of EmuCD's results can only be done after running a protocol in a VANET, registering the complete datatrace with every event and GPS position, and using it to emulate the protocol under exactly the same conditions and comparing the results. The tests performed can be used to establish initial impressions and conclusions regarding IPFS and EmuCD's performance, but the ideal is to test for a period of time much longer than just 5 days and preferably in a VANET larger than 5 nodes, as large as possible.

- **IPFS in a production VANET** - of course the final goal would be to have IPFS running in a real VANET permanently, facing the challenges and hurdles that this type of scenario offers, studying IPFS's performance in long periods of time.

- **IPFS applications** - IPFS was only tested with some files being exchanged between nodes. A better testing scenario would be to deploy into a VANET IPFS-based applications, such as a video streaming service, a software updating service or use IPFS for data collection from sensors in OBUs and RSUs, just to name a few interesting scenarios. Perhaps even better would be to use libp2p's PubSub protocol, rather than just *simple* file transfers.

- **Improvements to IPFS** - although IPFS is a protocol under active development and always getting better, some changes could be made to improve the download logic (and CID requests) or, for example, experiment with data compression in order to reduce the load on the network. This is a particular relevant aspect in VANETs, where the time connected is more valuable than spending a bit of extra CPU time decompressing the received chunks.

Bibliography

[1] N. Liu, M. Liu, G. Chen, and J. Cao. The sharing at roadside: Vehicular content distribution using parked vehicles. In *2012 Proceedings IEEE INFOCOM*, pages 2641–2645, March 2012.

[2] Xiaowen Chu and Yixin Jiang. Random Linear Network Coding for Peer-to-Peer Applications. *IEEE Network*, 24:35–39, 07 2010.

[3] Joao Gonçalves Filho, Ahmed Patel, Bruno Lopes Alcantara Batista, and Joaquim Celestino. A systematic technical survey of DTN and VDTN routing protocols. *Computer Standards & Interfaces*, 48:139 – 159, 2016. Special Issue on Information System in Distributed Environment.

[4] Meisong Wang, Prem Prakash Jayaraman, Rajiv Ranjan, Karan Mitra, Miranda Zhang, Eddie Li, Samee Khan, Mukkaddim Pathan, and Dimitrios Georgeakopoulos. *An Overview of Cloud Based Content Delivery Networks: Research Dimensions and State-of-the-Art*, pages 131–158. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[5] Manoj Parameswaran, Anjana Susarla, and Andrew Whinston. P2P networking: An information-sharing alternative. *Computer*, 34:31 – 38, 08 2001.

[6] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System. *CoRR*, abs/1407.3561, 2014.

[7] I. Baumgart and S. Mies. S/Kademlia: A practicable approach towards secure key-based routing. In *2007 International Conference on Parallel and Distributed Systems*, pages 1–8, 2007.

[8] D. Recharte, A. Aguiar, and H. Cabral. Cooperative Content Dissemination on Vehicular Networks. In *2018 IEEE Vehicular Networking Conference (VNC)*, pages 1–8, Dec 2018.

[9] R. Chaves, C. Senna, M. Luís, S. Sargento, A. Moreira, D. Recharte, and R. Matos. EmuCD: An Emulator for Content Dissemination Protocols in Vehicular Networks. *Future Internet*, 12(12):234, 2020.

[10] Sipat Triukose, Zhihua Wen, and Michael Rabinovich. Measuring a Commercial Content Delivery Network. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 467—-476, New York, NY, USA, 2011. Association for Computing Machinery.

[11] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up Data in P2P Systems. *Commun. ACM*, 46(2):43–48, February 2003.

[12] L. Wang and J. Kangasharju. Measuring large-scale distributed systems: case of BitTorrent Mainline DHT. In *IEEE P2P 2013 Proceedings*, pages 1–10, 2013.

[13] P. R. Pereira, A. Casaca, J. J. P. C. Rodrigues, V. N. G. J. Soares, J. Triay, and C. Cervello-Pastor. From Delay-Tolerant Networks to Vehicular Delay-Tolerant Networks. *IEEE Communications Surveys Tutorials*, 14(4):1166–1182, 2012.

[14] D. Jiang and L. Delgrossi. IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments. In *VTC Spring 2008 - IEEE Vehicular Technology Conference*, pages 2036–2040, 2008.

[15] Bilal Erman Bilgin and Vehbi Cagri Gungor. Performance comparison of IEEE 802.11 p and IEEE 802.11 b for vehicle-to-vehicle communications in highway, rural, and urban areas. *International Journal of Vehicular Technology*, 2013, 2013.

[16] E. C. Eze, S. Zhang, and E. Liu. Vehicular ad hoc networks (VANETs): Current state, challenges, potentials and way forward. In *2014 20th International Conference on Automation and Computing*, pages 176–181, 2014.

[17] Valerian Mannoni, Vincent Berg, Stefania Sesia, and Eric Perraud. A comparison of the V2X communication systems: ITS-G5 and C-V2X. In *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*, pages 1–5. IEEE, 2019.

[18] Alessandro Bazzi, Giammarco Cecchini, Michele Menarini, Barbara M Masini, and Alberto Zanella. Survey and perspectives of vehicular Wi-Fi versus sidelink cellular-V2X in the 5G era. *Future Internet*, 11(6):122, 2019.

[19] Mohamed Nidhal Mejri, Jalel Ben-Othman, and Mohamed Hamdi. Survey on VANET security challenges and possible cryptographic solutions. *Vehicular Communications*, 1(2):53 – 66, 2014.

[20] Saif Al-Sultan, Moath Al-Doori, and Ali Al-Bayatti. A comprehensive survey on vehicular Ad Hoc network. *Journal of Network and Computer Applications*, 37:380—-392, 01 2014.

[21] Rakesh Kumar, Mayank Dave, et al. A review of various VANET data dissemination protocols. *International Journal of u-and e-Service, Science and Technology*, 5(3):27–44, 2012.

[22] G. Chengetanai and G. B. O'Reilly. Survey on simulation tools for wireless mobile ad hoc networks. In *2015 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, pages 1–7, 2015.

[23] Narendra Mohan Mittal and Savita Choudhary. Comparative Study of Simulators for Vehicular Ad-hoc Networks (VANETs). *IEEE International Journal of Emerging Technology and Advanced Engineering*, 4, 2014.

[24] Helder Fontes, Rui Campos, and Manuel Ricardo. A Trace-Based Ns-3 Simulation Approach for Perpetuating Real-World Experiments. In *Proceedings of the Workshop on Ns-3*, WNS3 '17, pages 118–124, New York, NY, USA, 2017. Association for Computing Machinery.

[25] P. M. Santos, J. G. P. Rodrigues, S. B. Cruz, T. Lourenco, P. M. d'Orey, Y. Luis, C. Rocha, S. Sousa, S. Crisóstomo, C. Queirós, S. Sargento, A. Aguiar, and J. Barros. PortoLivingLab: An IoT-Based Sensing Platform for Smart Cities. *IEEE Internet of Things Journal*, 5(2):523–532, 2018.

[26] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic Traffic Simulation using SUMO. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018.

[27] Jerome Harri, Marco Fiore, Fethi Filali, and Christian Bonnet. Vehicular mobility simulation with VanetMobiSim. *SIMULATION*, 87(4):275–300, 2011.

[28] V. D. Khairnar and S. N. Pradhan. Mobility models for Vehicular Ad-hoc Network simulation. In *2011 IEEE Symposium on Computers Informatics*, pages 460–465, 2011.

[29] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[30] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.

[31] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P Systems Scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418, 2003.

[32] Ari Keränen, Jörg Ott, and Teemu Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. In *SIMUTools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, New York, NY, USA, 2009. ICST.

[33] R. Mangharam, D. Weller, R. Rajkumar, P. Mudalige, and F. Bai. GrooveNet: A Hybrid Simulator for Vehicle-to-Vehicle Networks. In *2006 Third Annual International Conference on Mobile and Ubiquitous Systems: Networking Services*, pages 1–8, 2006.

[34] M. Piórkowski, M. Raya, A. Lezama Lugo, P. Papadimitratos, M. Grossglauser, and J.-P. Hubaux. TraNS: Realistic Joint Traffic and Network Simulator for VANETs. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12(1):31–33, 2008.

[35] S. Y. Wang and C. L. Chou. NCTUns 5.0 Network Simulator for Advanced Wireless Vehicular Network Researches. In *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, pages 375–376, 2009.

[36] S.Y. Wang, C.L. Chou, C.H. Huang, C.C. Hwang, Z.M. Yang, C.C. Chiou, and C.C. Lin. The design and implementation of the NCTUns 1.0 network simulator. *Computer Networks*, 42(2):175 – 197, 2003.

[37] Kusum Dalal, Prachi Chaudhary, and Pawan Dahiya. Performance evaluation of TCP and UDP protocols in VANET scenarios using NCTUns-6.0 simulation tool. *International Journal of Computer Applications*, 36(6):6–9, 2011.

[38] Parul Tyagi and Deepak Dembla. Performance analysis and implementation of proposed mechanism for detection and prevention of security attacks in routing protocols of vehicular ad-hoc network (VANET). *Egyptian Informatics Journal*, 18(2):133 – 139, 2017.

[39] C. Sommer, R. German, and F. Dressler. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, 2011.

[40] Pedro Cirne, André Zúquete, and Susana Sargento. Loop - A Trace-based Emulator for Vehicular Ad Hoc Networks. In *8th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2018)*, SIMULTECH 2018, page 391–402, Setubal, PRT, 2018. SCITEPRESS - Science and Technology Publications, Lda.

[41] G. Pessoa, R. Dias, T. Condeixa, J. Azevedo, L. Guardalben, and S. Sargento. Content distribution emulation for vehicular networks. In *2017 Wireless Days*, pages 208–211, 2017.

[42] Goncalo Pessoa, Miguel Luis, Lucas Guardalben, and Susana Sargento. On the Analysis of Content Dissemination through Real Vehicular Boards. In *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*, pages 1–7, 06 2018.

[43] Gonçalo Pessoa, Lucas Guardalben, Miguel Luís, Carlos Senna, and Susana Sargento. Evaluation of Content Dissemination Strategies in Urban Vehicular Networks. *Information*, 11(3):163, 2020.

[44] A. . Kermarrec, L. Massoulie, and A. J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, 2003.

[45] J. Conde, C. Senna, and S. Sargento. Content Distribution Optimization Algorithms in Vehicular Networks. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00871–00877, 2018.

[46] Y. Wang, Y. Liu, J. Zhang, H. Ye, and Z. Tan. Cooperative StoreCarryForward Scheme for Intermittently Connected Vehicular Networks. *IEEE Transactions on Vehicular Technology*, 66(1):777–784, Jan 2017.

[47] Wei Huang and Liangmin Wang. ECDS: Efficient collaborative downloading scheme for popular content distribution in urban vehicular networks. *Computer Networks*, 101:90 – 103, 2016. Industrial Technologies and Applications for the Internet of Things.

[48] T. Wang, L. Song, Z. Han, and B. Jiao. Dynamic Popular Content Distribution in Vehicular Networks using Coalition Formation Games. *IEEE Journal on Selected Areas in Communications*, 31(9):538–547, 2013.

[49] M. Li, Z. Yang, and W. Lou. CodeOn: Cooperative Popular Content Distribution for Vehicular Networks using Symbol Level Network Coding. *IEEE Journal on Selected Areas in Communications*, 29(1):223–235, January 2011.

[50] Stefan Saroiu, Krishna P. Gummadi, and Steven Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. *Proc SPIE*, 03 2002.

[51] A. Nandan, S. Das, G. Pau, M. Gerla, and M. Y. Sanadidi. Co-operative downloading in vehicular ad-hoc wireless networks. In *Second Annual Conference on Wireless On-demand Network Systems and Services*, pages 32–41, 2005.

[52] Uichin Lee, Joon-Sang Park, Joseph Yeh, Giovanni Pau, and Mario Gerla. Code Torrent: Content Distribution Using Network Coding in VANET. In *Proceedings of the 1st International Workshop on Decentralized Resource Sharing in Mobile Computing and Networking*, MobiShare '06, pages 1—-5, New York, NY, USA, 2006. Association for Computing Machinery.

[53] X. Wang and Y. Li. Content Delivery Based on Vehicular Cloud. *IEEE Transactions on Vehicular Technology*, 69(2):2105–2113, 2020.

[54] Victor Ortega and Jose F. Monserrat. Semantic Distributed Data for Vehicular Networks Using the Inter-Planetary File System. *Sensors*, 20(22):6404, Nov 2020.

[55] T. Mishra, D. Garg, and M. M. Gore. A Publish/Subscribe Communication Infrastructure for VANET Applications. In *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, pages 442–446, 2011.

[56] R. Ahlswede, Ning Cai, S. . R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, 2000.

[57] Christos Gkantsidis, John Miller, and Pablo Rodriguez. Anatomy of a P2P Content Distribution system with Network Coding. In *International Workshop on Peer-to-Peer Systems*, volume 6, pages 1–6. Citeseer, 2006.

[58] Guanjun Ma, Yinlong Xu, Minghong Lin, and Ying Xuan. A content distribution system based on sparse linear network coding. In *Third Workshop on Network Coding (Netcod 2007)*. Citeseer, 2007.

[59] Suparna DasGupta, Rituparna Chaki, and Sankhayan Choudhury. TruVAL: Trusted Vehicle Authentication Logic for VANET. In Srija Unnikrishnan, Sunil Surve, and Deepak Bhoir, editors, *Advances in Computing, Communication, and Control*, pages 309–322, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[60] Hong Zhong, Shunshun Han, Jie Cui, Jing Zhang, and Yan Xu. Privacy-Preserving Authentication Scheme with Full Aggregation in VANET. *Information Sciences*, 476, 10 2018.

[61] J. Sun, C. Zhang, Y. Zhang, and Y. Fang. An Identity-Based Security System for User Privacy in Vehicular Ad Hoc Networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(9):1227–1239, Sep. 2010.

[62] M. Shabbir, M. A. Khan, U. S. Khan, and N. A. Saqib. Detection and Prevention of Distributed Denial of Service Attacks in VANETs. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 970–974, 2016.

[63] David David Folkman Mazières. *Self-certifying file system.* PhD thesis, Massachusetts Institute of Technology, 2000.

[64] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing Content Publication with Coral. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, page 18, USA, 2004. USENIX Association.

[65] Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[66] Mubariz Rehman, Zahoor Ali Khan, Muhammad Umar Javed, Muhammad Zohaib Iftikhar, Usman Majeed, Imam Bux, and Nadeem Javaid. A Blockchain Based Distributed Vehicular Network Architecture for Smart Cities. In Leonard Barolli, Flora Amato, Francesco Moscato, Tomoya Enokido, and Makoto Takizawa, editors, *Web, Artificial Intelligence and Network Applications*, pages 320–331, Cham, 2020. Springer International Publishing.

[67] Shay Gueron. Intel's New AES Instructions for Enhanced Performance and Security. In *Fast Software Encryption*, pages 51–66. Springer Berlin Heidelberg, 2009.

[68] Sean Turner. Transport layer security. *IEEE Internet Computing*, 18(6):60–63, 2014.

[69] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[70] Morris J Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Technical report, 2015.

[71] S. A. Ben Mussa, M. Manaf, K. Z. Ghafoor, and Z. Doukha. Simulation tools for Vehicular Ad Hoc Networks: A comparison study and future perspectives. In *2015 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pages 1–8, 2015.

[72] Rajive Bagrodia and Mario Gerla. A Modular and Scalable Simulation Tool for Large Wireless Networks. In Ramon Puigjaner, Nunzio N. Savino, and Bartomeu Serra, editors, *Computer Performance Evaluation*, pages 1–14, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[73] Shanshan Lu, S. Shere, Yanliang Liu, and Yonghe Liu. Device discovery and connection establishment approach using Ad-Hoc Wi-Fi for opportunistic networks. In *2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 461–466, 2011.

[74] C. Funai, C. Tapparello, and W. Heinzelman. Enabling multi-hop ad hoc networks through WiFi Direct multi-group networking. In *2017 International Conference on Computing, Networking and Communications (ICNC)*, pages 491–497, 2017.