INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

Otimização da entrega over-the-air em veículos conectados

ANTÓNIO FERNANDO DIONÍSIO DA SILVA FESTAS BARBOSA

Setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR
DE ENGENHARIA DO PORTO

Otimização da entrega *over-the-air* em veículos conectados

António Fernando Dionísio da Silva Festas Barbosa

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Rui Filipe Marques

Porto, setembro de 2025

Declaração de Integridade

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, 21 de setembro de 2025

Dedicatória

Mestre em “já experimentou desligar e ligar?”

Resumo

A evolução dos veículos conectados trouxe novos desafios às comunicações *over-the*-air (OTA), que exigem soluções fiáveis, escaláveis e eficientes. Esta dissertação, elaborada no âmbito da Unidade Curricular de Dissertação do Mestrado em Engenharia Informática do Instituto Superior de Engenharia do Porto teve como objetivo estudar e otimizar a entrega de conteúdo OTA através da análise de protocolos de comunicação, algoritmos diferenciais e estratégias de compressão e de *cache*, focando-se na minimização da latência e do tamanho dos ficheiros enviados (*payload*).

Foi desenvolvida uma arquitetura experimental composta por um *backend* em Java 21 com Spring Boot e um cliente Raspberry Pi 5, permitindo simular cenários realistas de atualização de *blobs* de mapas. Foram implementados e avaliados algoritmos diferenciais (*bsdiff* e *xdelta3*), combinados com compressão (*gzip*) e com o conceito de *hot zones* em *cache*. As métricas analisadas incluíram tempo de resposta *end-to-end* e tamanho de *payload* transferido.

Os resultados demonstraram que o uso de pacotes de atualizações (*patches*) diferencias reduziram substancialmente a latência (até 75%) e o volume de dados transmitidos (cerca de 90%), em comparação com o envio de *blobs* completos. Verificou-se que *xdelta3* é mais eficiente em tempo, enquanto *bsdiff* gera *patches* ligeiramente menores. A compressão com *gzip* apenas se mostrou benéfica em artefactos de maior dimensão, sendo contraproducente em *patches* pequenos. A estratégia de *hot zones* mostrou-se essencial para garantir escalabilidade, reduzindo em cerca de 50% o tempo médio de resposta em cenários de elevada reutilização de conteúdos.

Este estudo conclui que a combinação de algoritmos diferenciais, compressão seletiva e *cache* baseada em *hot zones* constitui uma abordagem eficaz para otimizar comunicações OTA. São ainda propostas perspetivas de evolução que incluem a integração de outros protocolos (MQTT, QUIC), simulação de redes adversas e políticas adaptativas de compressão e envio.

Palavras-chave: *Over-the-air*, veículos conectados, V2X, algoritmos diferenciais, compressão, *caching*

Abstract

The evolution of connected vehicles has introduced new challenges to over-the-air (OTA) communications, which require reliable, scalable, and energy-efficient solutions. This dissertation, developed within the Master's in Computer Engineering at *Instituto Superior de Engenharia do Porto*, aimed to study and optimize OTA content delivery by analyzing communication protocols, differential algorithms, and strategies for compression and caching.

An experimental architecture was developed, consisting of a backend built in Java 21 with Spring Boot and a client running on a Raspberry Pi 5, enabling the simulation of realistic map blob update scenarios. Differential algorithms (bsdiff and xdelta3) were implemented and evaluated, in combination with compression (gzip) and a caching strategy based on hot zones. The main metrics considered were end-to-end response time and payload size.

The results showed that using differential patches significantly reduces latency (up to 75%) and data volume (around 90%) compared to full blob delivery. It was observed that xdelta3 achieves better performance in terms of response time, while bsdiff generates slightly smaller patches. gzip compression proved beneficial only for larger artefacts, being counterproductive for small patches. The hot zone caching strategy was essential to scalability, reducing average response times by approximately 50% in scenarios with high content reuse.

This study concludes that the combination of differential algorithms, selective compression, and hot zone caching provides an effective approach to optimizing OTA communications. Future work should explore the integration of additional protocols (MQTT, QUIC), the simulation of adverse network conditions, and adaptive policies for compression and transmission.

Keywords: Over-the-air, connected vehicles, V2X, differential algorithms, compression, caching

Agradecimentos

À minha família, a minha mãe Constança e às minhas irmãs Mariana e Constança, porto de abrigo.

À minha madrinha Helena, luz e exemplo para sempre.

À minha namorada, Inês, pela paciência e apoio ao longo desta jornada.

Ao ISEP, casa durante estes 6 anos “desde o primeiro dia”.

Ao meu orientador, Prof. Rui Marques, pelo apoio ao longo do desenvolvimento deste documento.

À Critical Techworks pelo apoio no desenvolvimento deste documento e pela aprendizagem, à minha equipa Krokotiles, pelo apoio e compreensão, e ao meu orientador João Paulo Peixoto.

Índice

1 Introdução ... 1

1.1 Enquadramento ... 1

1.2 Contextualização ... 1

1.3 Descrição do problema ... 2

1.4 Estrutura do documento ... 3

2 Planeamento ... 5

2.1 Abordagem ... 5
- 2.1.1 Metodologia sistemática – PRISMA ... 5
- 2.1.2 Metodologia *Agile* ... 6

2.2 Estrutura do projeto ... 7
- 2.2.1 Objetivos ... 7
- 2.2.2 Benefícios ... 7
- 2.2.3 Metodologia do planeamento ... 7
- 2.2.4 Gestão do projeto ... 9
- 2.2.5 Considerações finais ... 9

3 Estado da Arte ... 11

3.1 Veículos conectados ... 11

3.2 Protocolos de Comunicação ... 13
- 3.2.1 TCP vs UDP ... 13
- 3.2.2 HTTP ... 15
- 3.2.3 MQTT ... 16
- 3.2.4 QUIC ... 17
- 3.2.5 Comparação entre protocolos ... 18

3.3 *Payload* – compressão e encriptação eficiente ... 19
- 3.3.1 Compressão ... 19
- 3.3.2 Compressão diferencial – *delta* ... 21

3.4 PRISMA ... 22
- 3.4.1 Perguntas de pesquisa ... 24
- 3.4.2 Discussão ... 25
- 3.4.3 Conclusão ... 26

4 Desenho da Solução ... 27

4.1 Proposta de Arquitetura ... 27
- 4.1.1 Backend ... 27
- 4.1.2 Veículo conectado ... 28

4.2 Protocolos de comunicação (HTTP/MQTT) ... 29

4.3 Algoritmos de compressão e diferenciais (gzip, bsdiff, xdelta3) ... 30

4.4 Estratégia de *caching* e conceito de *hot zones* ... 31

4.5 Métricas e critérios de avaliação 31
4.6 Alternativas de *design* 32
4.6.1 Alternativas *backend* 32
4.6.2 Alternativas simulação veículo 33
4.6.3 Protocolos adicionais 34
4.6.4 Comparação de arquiteturas 34
4.6.5 Conclusão 35
5 Implementação 37
5.1 Ambiente de testes 38
5.1.1 Backend 38
5.1.2 Simulação cliente OTA 41
5.2 Ferramentas utilizadas 42
5.3 Configuração dos testes 43
5.3.1 Justificação na escolha dos parâmetros 43
5.3.2 Execução sequencial vs concorrente 44
5.4 Testes unitários no *backend* 45
5.4.1 Estratégias de testes unitários 45
5.5 Estratégia de CI/CD 47
5.6 Automação da recolha de métricas 49
5.7 Métodos de captura e análise 50
5.8 Considerações finais sobre a implementação 50
6 Análise de Resultados 53
6.1 *Baseline* de resultados 53
6.2 Resultados com cache (*hot zones*) 53
6.2.1 Visão global e comparação com a *baseline* 53
6.2.2 Impacto da probabilidade de acerto na *cache* (p) 55
6.2.3 Impacto da dimensão da *hot zone* 56
6.3 Resultados sem *cache* no *backend* 58
6.3.1 Comparação com a solução com *cache* no *backend* 59
6.4 Comparação entre algoritmos 60
6.4.1 Estabilidade face a p e k 60
6.5 Impacto da compressão 60
6.6 Discussão dos resultados 61
6.6.1 Principais constatações 61
6.6.2 Implicações operacionais para a arquitetura 61
6.6.3 Ameaças à validade e generalização 62
6.6.4 Perspetivas de evolução 62
6.7 Resultados sob perspetiva de escalabilidade 62
7 Conclusões 65

7.1 Conclusões 65
7.1.1 Eficácia das *hot zones* 65
7.1.2 Algoritmos diferenciais (*bsdiff* vs *xdelta3*) 65
7.1.3 Compressão (*gzip*) em *blobs* e *patches* 66
7.1.4 Estratégia de execução e validade dos resultados 66
7.1.5 Implicações práticas 66
7.1.6 Estado dos protocolos 66
7.1.7 Resumo 66

7.2 Limitações técnicas 67

7.3 Sugestões de trabalhos futuros 67

8 Referências 69

Lista de Figuras

Figura 1 - Timeline do Jira para gestão do projeto 8
Figura 2 - Arquitetura de comunicação V2X [15]........ 12
Figura 3 - Camadas OSI [19]........ 13
Figura 4 - Protocolo MQTT [25] 16
Figura 5 - Comparação entre HTTP e QUIC [29] 18
Figura 6 - Comparação entre algoritmos de compressão [32] 21
Figura 7 - Flow de um update delta [32] 21
Figura 8 - Comparação do rácio de compressão de algoritmos *delta* [37] 22
Figura 9 - Organização artigos PRISMA 23
Figura 10 - Diagrama de componentes OTA Backend & Veículo........ 28
Figura 11 - Código em C para Arduino........ 34
Figura 12 - Diagrama de interação carro-*backend-cloud* 37
Figura 13 - Código de geração de blobs aleatórios........ 38
Figura 14 - Execução do algoritmo diferencial através de ProcessBuilder 39
Figura 15 - Interface *PatchService* 39
Figura 16 - Utilização de Factory para distinguir entre serviços por algoritmo diferencial........ 40
Figura 17 - Exemplo REST API de produção de *patches* 40
Figura 18 - Diagrama de sequência da interação do Raspberry Pi com o *backend* em cenários de *hot zones* 42
Figura 19 - Exemplo utilização *Mockito*........ 46
Figura 20 - Exemplo utilização MockMVC 46
Figura 21 - Exemplo *pipeline* validação 48
Figura 22 - Exemplo de código *Python* para guardar os dados em CSV 49
Figura 23 – Comparação de tempo de resposta médio entre combinação de algoritmo de compressão e algoritmo diferencial........ 54
Figura 24 – Comparação de tamanho de payload médio entre combinação de algoritmo de compressão e algoritmo diferencial........ 54
Figura 25 – Comparação de tempo de resposta médio entre variação da probabilidade de pertencer à hot zone, por combinação de algoritmo de compressão e algoritmo diferencial . 55
Figura 26 - Comparação do tamanho médio do *payload* entre variação da probabilidade de pertencer à hot zone, por combinação de algoritmo de compressão e algoritmo diferencial . 56
Figura 27 - Comparação de tempo de resposta médio entre variação de k - tamanho da hot zone, por combinação de algoritmo de compressão e algoritmo diferencial 57
Figura 28 – Comparação do tamanho médio do *payload* entre variação de k - tamanho da hot zone, por combinação de algoritmo de compressão e algoritmo diferencial 58
Figura 29 - Comparação de tempo de resposta médio entre combinação de algoritmo de compressão e algoritmo diferencial sem cache 58

Orientador: Rui Filipe Marques

Porto, setembro de 2025

Figura 30 - Comparação tamanho de payload médio entre combinação de algoritmo de compressão e algoritmo diferencial sem cache 59
Figura 31 - Escalabilidade em Kubernetes 63
Figura 32 - Arquitetura Kubernetes 64

Lista de Tabelas

Tabela 1 - Critérios de inclusão e exclusão PRISMA 6
Tabela 2 - Comparação entre protocolos de comunicação 18
Tabela 3 – Exemplo de ficheiro de métricas 49

Acrónimos e Símbolos

Lista de Acrónimos

CI/CD	*Continuous Integration/Continuous Delivery*
CTW	Critical Techworks
CoAP	*Constrained Application Protocol*
CSV	*Comma-Separated Values*
DSR	*Design Science Research*
ECU	Electronic Control Units
GPS	*Global Positioning System*
IoT	*Internet of Things*
MQTT	*Message Queueing Telemetry Transport*
OTA	*Over-The-Air*
PRISMA	*Preferred Reporting Items for Systematic Reviews and Meta-Analyses*
RSU	*Remote Software Update*
RTTI	*Real-time Traffic Information*
TCP	*Transmission Control Protocol*
TLS	*Transport Layer Security*
UDP	*User Datagram Protocol*
V2X	*Vehicle-to-everything*
WBS	*Work Breakdown Structure*

1 Introdução

O conceito de veículos conectados refere-se a aplicações, serviços e tecnologias que conectam um veículo ao que o rodeia e/ou outras aplicações, redes e serviços fora da esfera do veículo. [1]. Esta dissertação tem como principal objetivo analisar os desafios destas comunicações, estudando a otimização dos sistemas de entrega OTA (*over-the-air*) em veículos conectados.

Mais concretamente, irão ser estudados diferentes protocolos de comunicação e a forma como a combinação destes protocolos com diversas técnicas de compressão e algoritmia podem tornar a comunicação entre o veículo e o *backend* mais eficiente e fluída, zelando pela qualidade, velocidade e reação à falha dos mesmos. Por fim, será analisado como é que a compressão e particionamento dos dados poderá colaborar para esta eficiência.

1.1 Enquadramento

Este documento foi desenvolvido com base num projeto enquadrado no âmbito da unidade curricular de DIMEI (Dissertação do Mestrado em Engenharia Informática) do Mestrado em Engenharia de Software do Instituto Superior de Engenharia do Porto – ISEP. Este projeto foi realizado na Critical Techworks, *joint venture* da Critical Software com o *BMW Group*, com o foco na otimização das comunicações entre o veículo e os servidores, tema transversal a muitos projetos desenvolvidos na empresa. O projeto desenvolvido permitiu aplicar conhecimentos adquiridos ao longo da licenciatura e do mestrado, bem como validar hipóteses de desenvolvimento para veículos com entregas OTA, facilitando este processo e tornando-o mais eficiente.

1.2 Contextualização

Com os desenvolvimentos tecnológicos das últimas duas décadas, associados às capacidades de comunicação em tempo real dos dispositivos, os veículos conectados estão na vanguarda da

indústria automotiva, transformando os automóveis em sistemas inteligentes, capazes de melhorar a experiência, segurança e qualidade da condução. [2] Deste modo, é necessário que estes dispositivos comuniquem continuamente com serviços, de modo a obterem toda a informação necessária. A comunicação OTA (*Over-the-air*) é fulcral para o bom funcionamento destes dispositivos, uma vez que garante que a informação presente nos mesmos é a mais atualizada e segura. Assim, a troca contínua de dados é essencial para manter as funcionalidades desses veículos, de forma a melhorar a experiência do utilizador, permitindo decisões em tempo real e veículos autónomos e semiautónomos [3].

Apesar do potencial transformador, a entrega OTA apresenta vários desafios que têm de ser endereçados de forma a assegurar eficiência e fiabilidade. Primeiro, as limitações de largura de banda e disponibilidade intermitente de sinal de rede podem levar a atrasos e/ou falhas na entrega dos dados, particularmente em regiões do globo com baixa cobertura de rede [4]. Além disso, a energia gasta no processo de transmissão é um problema crítico, uma vez que estes veículos se baseiam em fontes de energia limitadas, cuja eficiência deve ser gerida de forma correta, de forma a não comprometer outras funcionalidades. Finalmente, a escolha de protocolos de comunicação e estratégias de gestão da informação enviada – *payload* – impacta a velocidade e fiabilidade da transmissão de informação, sendo necessário uma avaliação cuidada de forma a atingir os requisitos que este tipo de entrega preveem.

A adoção de uma arquitetura orientada a eventos surge como uma abordagem promissora às operações de otimização de entrega de conteúdo OTA, uma vez que através da sua alavancagem, os sistemas do *backend* podem responder a eventos em tempo real, aumentando a escalabilidade e adaptação dinâmica a variações de condições de rede e demanda dos veículos [5]. Para além disso, técnicas de pré-processamento, como compressão do *payload*, entregas *batch-driven* e entregas de *deltas* de informação podem reduzir o volume e complexidade de dados transmitidos, de forma a mitigar restrições de largura de banda e consumo de energia.
Neste contexto, o foco deste estudo será a análise comparativa de protocolos de comunicação, como por exemplo MQTT, HTTP, QUIC, aliado à avaliação das estratégias de compressão, de forma a minimizar o volume de dados, preservando a integridade. Através desta otimização, esta tese tem como objetivo contribuir para o desenvolvimento de soluções escaláveis, confiáveis e eficientes, que suportam o futuro dos transportes conectados.

1.3 Descrição do problema

Com o aumento da prevalência dos veículos conectados, a necessidade de entrega de conteúdo em tempo real de forma eficiente aumentou exponencialmente. No entanto, as abordagens atuais tendem a ter problemas a balancear todos os requisitos de minimização da latência, eficiência energética e integridade dos dados. Para além disso, não existe um protocolo universal de comunicação para todos os cenários de comunicação OTA, pelo que é necessário existir uma abordagem dinâmica para vários cenários, onde o tamanho da *payload* e a urgência da informação podem ser critérios diferenciadores. Finalmente, a ausência de soluções robustas de pré-processamento contribuem para a complexidade da gestão de dados por parte

do veículo. Sem mecanismos de compressão eficiente, verificação de integridade ou *batching*, o sistema torna-se suscetível a ineficiências que comprometem a performance e fiabilidade. Estes desafios mostram a necessidade de uma solução holística, que integra otimização de protocolos, gestão de *payload* e pré-processamento no *backend*, de forma a atingir entregas OTA fiáveis e escaláveis a veículos conectados.

1.4 Estrutura do documento

Este documento será composto por 8 capítulos, para garantir que a leitura do mesmo seja fácil e eficiente. Assim, cada capítulo será dividido por secções, onde uma área específica será abordada em maior detalhe.

Assim, o capítulo da **Introdução** dá ao leitor um contexto geral do projeto, com a apresentação da problemática, o enquadramento do projeto, a metodologia de trabalho e pesquisa, bem como outras nuances relevantes para o desenvolvimento do projeto. O **Estado da Arte** é onde a pesquisa cujos métodos foram definidos no capítulo anterior tem lugar. Seguidamente, será apresentado o **Desenho da Solução** proposta e a sua **Implementação**, que permitirão entender qual o caminho tomado para a execução da mesma. Finalmente, serão analisados os resultados e serão tiradas conclusões, tendo em conta trabalhos futuros e possíveis melhorias. O documento conta ainda com uma bibliografia, que serve de suporte bibliográfico ao trabalho produzido.

2 Planeamento

2.1 Abordagem

De forma a garantir contexto compreensivo e desenvolvimento robusto desta tese, duas abordagens complementares foram utilizadas: PRISMA para revisão sistemática de literatura e metodologias *Agile* para o processo iterativo e adaptativo da pesquisa. O método PRISMA foi utilizado para estabelecer um método estruturado e transparente para identificar, avaliar e sintetizar estudos relevantes, de forma a assegurar uma base de alta qualidade para a pesquisa. As metodologias *Agile* foram utilizadas para permitir flexibilidade e refinamentos iterativos, acomodando a natureza dinâmica da exploração de novas tecnologias e soluções.

2.1.1 Metodologia sistemática – PRISMA

PRISMA, *Preferred Reporting Items for Systematic Reviews and Meta-Analyses*, é uma *framework* amplamente reconhecida que foi desenhada para melhorar a clareza e transparência das revisões sistemáticas de literatura [6]. Esta metodologia é bastante valiosa para sintetizar a literatura existente sobre um determinado tópico, pois permite a identificação, seleção e avaliação de estudos relevantes, minimizando o viés. Este processo garante que a revisão do estado da arte é compreensiva o suficiente, estruturada e alinhada com os *standards* académicos.

De forma a fazer o estudo baseado neste método, foi necessário definir uma metodologia de trabalho. Assim, em primeiro lugar foi definido o objetivo da pesquisa que, neste caso foi:

“Identificar, avaliar e analisar as metodologias chaves, protocolos e estratégias para a otimização do conteúdo OTA em veículos conectados, com o foco na eficiência, escalabilidade, fiabilidade e integridade da informação”

Depois, definiram-se três perguntas de pesquisa que abrangiam os temas que esta tese aborda. As perguntas foram:

- Quais são os protocolos mais eficientes para a entrega de conteúdo em veículos conectados?
- Como é que diferentes protocolos de comunicação (e.g., MQTT, HTTP/2, QUIC) se comparam em termos de latência, utilização da largura de banda e fiabilidade em entregas a veículos conectados?
- Quais são os algoritmos de compressão mais eficientes para minimizar o tamanho da carga sem comprometer a integridade dos dados durante as entregas de informação?

Com as perguntas definidas, foi necessário decidir quais as bases de dados onde a pesquisa iria ser efetuada, bem como os critérios de inclusão e exclusão. As bases de dados utilizadas para a pesquisa foram a IEEE Xplore [7], Google Scholar [8] e ACM Digital Library [9]. A Tabela 1 mostra os critérios de inclusão e exclusão utilizados para esta pesquisa.

Tabela 1 - Critérios de inclusão e exclusão PRISMA

Inclusão	Exclusão
Artigos de jornais *peer reviewed*, *papers* de conferências e relatórios técnicos	Estudos que não tenham sido *peer reviewed* ou *blogs*
Publicações dos últimos 10-15 anos, com um foco maior nos avanços dos últimos 5	Estudos cujo foco não seja a indústria automotiva
Estudos em inglês	Artigos que não possuam métricas de performance ou análise comparativa
Estudos que discutam protocolos de comunicação, arquitetura orientada a eventos, gestão de *payload*, entrega OTA em veículos conectados	Estudos que não sejam em inglês
	Estudos duplicados ou com dados incompletos

2.1.2 Metodologia *Agile*

Agile é uma metodologia de trabalho flexível e iterativa para gestão de projetos e desenvolvimento de *software*, com ênfase em colaboração, *feedback* do cliente e adaptabilidade. Foca-se na entrega de pequenos incrementos a um produto, permitindo que as equipas respondam de forma rápida a mudanças nos requisitos ou condições do mercado. O *Agile* encoraja a existência de trabalho em equipa *cross-functonial*, comunicação frequente e um comprometimento para uma melhoria contínua, almejando um ambiente onde o valor é entregue de forma eficiente e a satisfação do cliente é priorizada. *Scrum* e *Kanban* são duas das *frameworks* mais populares que suportam as práticas de *Agile*, permitindo métodos estruturados para implementar os princípios [10].

As metodologias *Agile* foram adotadas para gerir a pesquisa durante o desenvolvimento desta tese. A natureza iterativa de *Agile* permitiu que fossem feitas avaliações periódicas e refinamento do processo de pesquisa, permitindo a integração de novas descobertas e desenvolvimentos tecnológicos.

De forma a seguir a metodologia *Agile*, o processo de desenvolvimento foi organizado por *sprints*, onde cada *sprint* focou-se em objetivos específicos, como por exemplo a revisão de literatura, o desenho arquitetural, o *setup* da simulação e a avaliação de performance [10].

2.2 Estrutura do projeto

2.2.1 Objetivos

O objetivo principal desta tese é comparar as várias nuances da entrega de conteúdo OTA a veículos conectados, de forma a avaliar uma solução o mais otimizada possível. Assim, a análise vai ser feita por:

- Avaliar os diversos protocolos de comunicação utilizados para entrega de dados *over-the-air* para veículos conectados;
- Avaliar os vários métodos de encriptação e compactação de *payload* no que toca a integridade de informação e redução do volume de dados transferido;
- Avaliar uma arquitetura orientada a eventos no consumo e produção de conteúdo para veículos conectados;
- Simular entregas de conteúdo de larga escala, com uma frota de veículos virtuais, de forma a avaliar a escalabilidade;
- Utilizar diferentes protocolos e *payloads* adaptativos;
- Analisar o consumo energético do veículo em cada estratégia distinta;
- Medir métricas e analisar;
- Desenhar uma solução final o mais otimizada possível para a entrega de conteúdo para updates *over-the-air*.

2.2.2 Benefícios

Com os objetivos traçados para esta tese, os principais benefícios vão ser a melhoria da eficiência na entrega de conteúdo a veículos conectados, reduzindo a latência e o tamanho da *payload*. Também irá ser benéfico para entender que conjunto de protocolo + compressão de *payload* + pré-processamento poderão ser mais eficazes para cada cenário de entrega de conteúdo. Finalmente, o conhecimento adquirido será benéfico para compreender melhor o método de funcionamento de cada protocolo de comunicação, bem como das arquiteturas orientadas a eventos.

2.2.3 Metodologia do planeamento

De forma a planear e estruturar o projeto desenvolvido, foi utilizada a metodologia *Design Science Research* (DSR), uma abordagem de pesquisa metódica que tem como principal foco a criação e avaliação de artefactos inovadores de forma a resolver problemas complexos do mundo real. O desenvolvimento iterativo previsto pela DSR, com fundamentação na engenharia, incentiva à criação de soluções como por exemplo modelos, *frameworks*, sistemas ou algoritmos. O processo de DSR prevê a identificação do problema, desenho dos artefactos,

avaliação e comunicação das soluções encontradas, de modo a melhorar o campo de estudos em que o projeto se enquadra [11].

O processo de DSR é composto por sete passos fundamentais:

- **Identificação do problema**: Compreensão do desafio que será abordado.
- **Definição de objetivos para a solução**: Estabelecimento de metas específicas para a criação da solução.
- **Design e desenvolvimento do artefacto**: Criação de modelos, metodologias de trabalho, sistemas ou algoritmos.
- **Demonstração**: Aplicação da solução para resolver o problema identificado.
- **Avaliação**: Análise do desempenho e da eficácia da solução em resolver o problema.
- **Comunicação**: Documentação e disseminação dos resultados para a comunidade científica e prática.
- **Iteração contínua**: Melhoria incremental baseada no *feedback* das etapas anteriores.

No processo de desenvolvimento deste projeto, foi seguida uma metodologia de trabalho ágil. Assim, a utilização de SCRUM foi fulcral para que fosse possível organizar o tempo, tarefas e prioridades.

Deste modo, foi utilizado o Jira para gerir o tempo e as tarefas a realizar. Criaram-se 16 épicos, cada um com *user stories* que correspondem ao fracionamento desses épicos. Inicialmente foram criadas *sprints* de duas semanas, mas com o passar do tempo foi necessário criar *sprints* de 4 semanas, permitindo um desenvolvimento mais conciso das tarefas. A Figura 1 mostra a *board* do Jira, ilustrando no topo as várias *sprints* e a distribuição da realização dos épicos ao longo das mesmas.

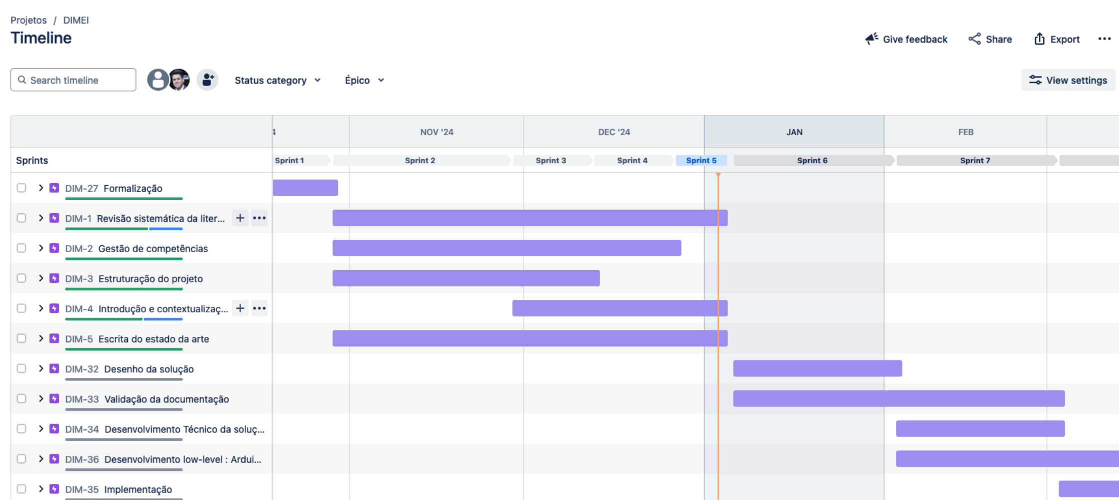


Figura 1 - Timeline do Jira para gestão do projeto

Com a estratégia de gestão de tempo definida, foi necessário definir qual o método científico a ser utilizado no desenvolvimento da tese, bem como na pesquisa e desenvolvimento técnico.

Assim, foi necessário inicialmente definir quais os tópicos a avaliar em cada uma das soluções pretendidas, bem como o que definia o sucesso e o que definia o insucesso de cada solução.

Na comunicação *over-the-air* com veículos conectados, foram definidos 2 fatores preponderantes de medição de sucesso:

- **Latência**: Um dos fatores mais importantes na comunicação com veículos conectados é se a comunicação foi feita no menor tempo possível, daí ser importante medir esta informação;
- **Tamanho do *payload***: Uma vez que o consumo da largura de banda é essencial, foi necessário medir o tamanho do *payload*, de modo a exercer o menor *stress* possível sobre a rede;

2.2.4 Gestão do projeto

2.2.4.1 Ferramentas a utilizar

Para a execução do projeto, foi necessária a utilização de determinadas ferramentas e tecnologias de desenvolvimento, teste e gestão do código e operações.

- IDEs
  - **Jetbrains IntelliJ IDEA**: Ambiente de desenvolvimento do código em Java cuja facilidade de configuração e ferramentas de *debug* facilitam o desenvolvimento
- Gestão de código e controlo de versões
  - **GitHub**: Gestão e manutenção do código-fonte, bem como integração com CI/CD através das GitHub Actions
- Gestão de tarefas
  - **Jira**: Gestão de *sprints* e de tarefas a desenvolver, acompanhamento do progresso e do estado das taregas
- Hospedagem e gestão de rede
  - **Raspberry Pi 5**: Utilização de um dispositivo Raspberry Pi 5, capaz de hospedar o sistema utilizado, bem como gerir a rede onde acedem os vários Arduino
  - **Servidores do DEI**: Utilização dos servidores do Departamento de Engenharia Informática para hospedagem, se necessário.

2.2.5 Considerações finais

As metodologias e competências analisadas neste capítulo fundamentam o sucesso desta tese. Através da integração de abordagens sistemáticas e iterativas, como o PRISMA (para a revisão de literatura) e metodologias Agile, o estudo garante um balanço entre rigor e flexibilidade. Esta combinação é fulcral para analisar os desafios proporcionados pela entrega de conteúdo *over-the-air* a veículos conectados.

Para além disso, o diagnóstico de competências e respetivo plano de desenvolvimento reiteram a importância de alinhar o crescimento pessoal e profissional com os requisitos do projeto. Este alinhamento reforça a qualidade da pesquisa e também prepara o pesquisador para desafios futuros, reforçando o valor de uma metodologia estruturada, porém adaptável, com foco na melhoria contínua.

3 Estado da Arte

3.1 Veículos conectados

O conceito de veículos conectados foi abordado, inicialmente, pela General Motors em parceria com a Motorola Automotive em 1996 com o *OnStar* [12], onde os veículos se conectavam a uma central em caso de emergência ou de ativação do *airbag*. Este tipo de sistemas surgiu pela necessidade de garantir uma conexão estável e segura, numa era onde os sistemas de comunicação não eram tão abrangentes e fiáveis. [13]

A integração destes sistemas foi antecedida por dispositivos como os acessórios de sistemas de navegação com GPS, que eram colocados no campo de visão do condutor e que permitiam que o mesmo obtivesse informações de localização em tempo real para conseguir estabelecer rotas entre o local onde se encontrava e o destino. A criação destes dispositivos foi um grande avanço tecnológico, mas que rapidamente se tornou obsoleta, pois as fabricantes dos veículos viram nestes produtos uma ótima ideia para ser integrada nos carros nativamente, tornando a experiência para o utilizador muito mais fácil, sem existir a necessidade de aquisição de outros dispositivos. [14]

Assim, existiu a necessidade de criar uma rede onde o veículo pede o conteúdo que precisa, podendo basear esse pedido em determinadas informações (por exemplo, a sua localização ou dados dos sensores). Esta rede sofreu bastantes alterações nos últimos anos, que acompanhou o desenvolvimento tecnológico. Foram feitos esforços no âmbito da *standartização* da comunicação entre o veículo e todas as partes envolvidas no processo. Essa comunicação é designada de *Vehicular-to-Everything* (V2X) e subentende a comunicação sem fio entre um veículo e uma entidade que afeta, pode afetar ou poderá ser afetada pelo veículo [15].

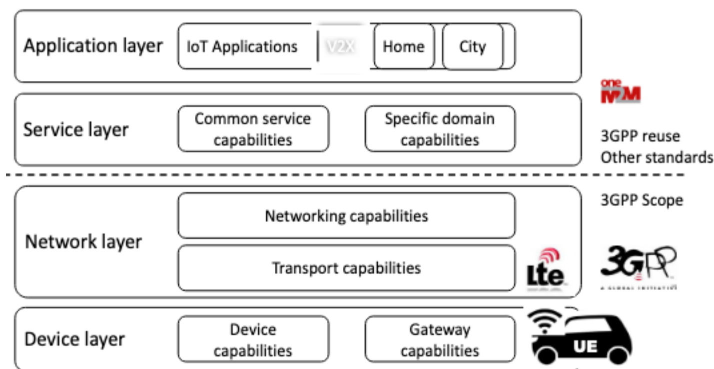


Figura 2 - Arquitetura de comunicação V2X [15]

No entanto, a comunicação pode não ser sempre no sentido veículo > servidor, mas também servidor > veículo, baseando-se no princípio de atualizações remotas, quando existe necessidade para tal. Por exemplo, se o sistema instalado na *head unit* do carro tem algum *software* onde foi encontrada alguma vulnerabilidade, pode existir a necessidade de a marca fazer um *Remote Software Update* (RSU), de modo a garantir a segurança informática do veículo, mitigando possíveis ataques ou acessos indevidos [16]. Também pode existir uma entrega de conteúdo do servidor para o veículo no *Infotainment* – entretenimento a bordo – aquando de, por exemplo, uma mensagem personalizada de Feliz Natal, para aparecer na viatura quando a mesma é ligada.

Na ótica dos veículos conectados, também é necessário que os servidores onde estes vão buscar informação sejam robustos o suficiente para balancear a carga e giram os recursos de forma eficiente, de forma a não gastar recursos a mais ou não existir recursos suficientes. Assim, a utilização de balanceadores de carga (*load balancers*), a distribuição de instâncias do mesmo serviço, paralelismo e *stream* de informação são alguns dos mecanismos que permitem otimizar o processo de gestão de recursos no *backend* [17].

A codificação da informação e compressão da mesma também tem um papel preponderante no envio da mesma, uma vez que garante a integridade e pode reduzir a carga no sistema. Assim, sempre que a comunicação é estabelecida, deve existir a verificação de integridade, com recurso a *checksum*, de forma a validar que o conteúdo recebido é igual ao conteúdo que foi enviado. No caso de não ser, é necessário entender como é que o sistema distribuído será capaz de lidar com a falha, repetindo (ou não) o pedido, na sua integra ou só na parte que necessita.

Assim, é necessário que o protocolo de comunicação seja resistente a falhas e instabilidades de rede, bem como possua mecanismos de gestão de falhas, para que a robustez do processo comunicativo seja garantida. Esse é, também, um dos fatores mais importantes a ter em consideração ao analisar vários protocolos de comunicação, pois os veículos conectados estão constantemente a ser expostos a possíveis falhas na rede, como por exemplo a entrada num túnel, uma zona com fraca cobertura de sinal, ou mesmo uma zona com muitos veículos e pessoas, onde a pressão sobre a rede é mais elevada. [18]

3.2 Protocolos de Comunicação

Com o desenvolvimento tecnológico e a noção de rede computacional, a comunicação entre dispositivos tornou-se um dos tópicos basilares do desenvolvimento de software e hardware. Assim, foram estabelecidas convenções que permitem que todos os dispositivos, independentemente do fabricante ou país, sejam capazes de comunicar. [19]

O modelo OSI (*Open Systems Interconnection*) é uma *framework* conceptual que standartiza comunicações de rede em 7 camadas, como indicado na Figura 3.

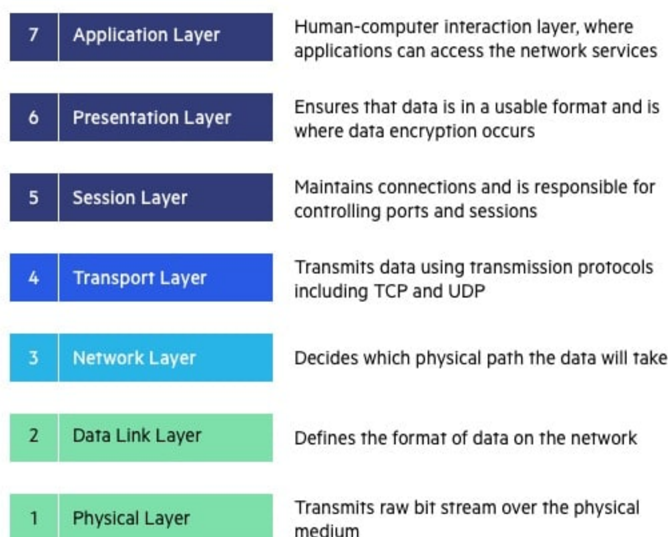


Figura 3 - Camadas OSI [19]

3.2.1 TCP vs UDP

3.2.1.1 TCP

Transmission Control Protocol (TCP) é um protocolo de comunicação orientado a conexão, desenhado para comunicações fiáveis. Opera através do estabelecimento de uma conexão entre um transmissor e um recetor antes do início de troca de dados [20]. Esta conexão garante que os pacotes de dados enviados são entregues de forma correta, respeitando a ordem, tornando o TCP um protocolo ideal para cenários onde a fiabilidade é critica, como por exemplo atualizações de *firmware* ou ficheiros de configuração. Este protocolo utiliza pacotes de reconhecimento, de modo a confirmar a receção da mensagem e retransmite pacotes corrompidos ou perdidos no processo, sendo fulcral a integridade dos dados [21].

Uma das principais vantagens de TCP é a robustez na gestão de questões de rede. Caso um pacote seja perdido durante o processo de transmissão devido a fatores como a fraca qualidade de rede, os mecanismos inerentes ao TCP detetam e corrigem a falha, enviando novamente os

pacotes em falta [5]. Assim, o processo de entrega torna-se mais fácil e sem erros associados, embora com algum atraso.

Apesar das vantagens, a fiabilidade traz custos associados: a gestão de conexões e reconhecimentos pode atrasar a transmissão de dados e aumentar a latência, o que pode trazer problemas a aplicações onde o tempo é fulcral, como por exemplo telemetria em tempo real ou comunicações entre veículos.

Em veículos conectados, o protocolo TCP torna-se mais eficaz para cenários onde a fiabilidade é mais importante que a velocidade de transmissão. Atualizações ao *software* ou operações críticas podem alavancar a utilização de TCP de modo a garantir uma entrega completa e correta, mesmo em condições de rede adversas [21].

Outra desvantagem do protocolo TCP é o mecanismo de controlo de congestão, que ajusta a percentagem de dados transmitidos baseado nas condições de rede. Apesar de prevenir congestionamento de rede, reduz a velocidade de transmissão, principalmente em situações de tráfego elevado. No ecossistema dos veículos conectados, apesar de ser vantajoso na estabilização de conexões estáveis, pode trazer atrasos nas horas de ponta.

Apesar das limitações, a ampla adoção do protocolo TCP e a compatibilidade com várias estruturas de rede faz com que seja uma opção bastante confiável para os desenvolvedores, devido à consistência de dados e ferramentas de gestão de erros [20].

3.2.1.2 UDP

User Datagram Protocol (UDP) é um protocolo de comunicação *connectionless* que valoriza velocidade e simplicidade, em detrimento da fiabilidade. Ao contrário de TCP, UDP não estabelece uma conexão ou faz reconhecimento de mensagens enviadas e recebidas. Os pacotes de dados são enviados para o recetor, sem a garantia de que os mesmos irão chegar ou que a ordem dos mesmos é mantida [2]. Este facto faz do UDP altamente eficiente para aplicações onde a latência baixa é essencial, como por exemplo em *streaming* de vídeo ou atualizações de localização em tempo real.

O *overhead* mínimo deste protocolo torna-o uma opção bastante viável para cenários onde a velocidade é mais importante do que a exatidão. Por exemplo, na transmissão de dados em tempo real, como por exemplo coordenadas de GPS ou atualizações de tráfego, a velocidade de entrega é fulcral, mesmo que alguns pacotes sejam perdidos pelo caminho [22]. No entanto, a simplicidade apresentada pelo UDP traz desvantagens, como por exemplo a falta de correção de erros, o que quer dizer que pacotes perdidos ou corrompidos não são enviados novamente, tornando-o menos adequado para aplicações cuja exatidão da informação é importante.

No contexto dos veículos conectados, a natureza leve do protocolo UDP faz com que seja um forte candidato para operações onde a velocidade de entrega é importante, no entanto é necessário que os mecanismos de reação à falha estejam do lado da aplicação, de forma a reenviar informação perdida no processo [20].

Outra vantagem do protocolo UDP é o suporto para comunicação *multicast* e *broadcast*, permitindo que a informação seja enviada para vários recetores ao mesmo tempo. No entanto, pode tornar-se complicado de gerir o congestionamento de tráfego, sendo que é necessário gerir corretamente a rede de modo que os limites das redes não sejam atingidos.

3.2.1.3 Comparação

No contexto da entrega de conteúdo *over-the-air* a veículos conectados, a escolha entre TCP e UDP depende do caso de uso em questão. Como analisado anteriormente, o protocolo TCP é primordial em cenários onde a fiabilidade e integridade de dados são critérios não negociáveis [5]. No exemplo da entrega de software critico a uma frota específica de veículos, obriga a que os mecanismos de entrega sejam fiáveis o suficiente para que não existam falhas. Por outro lado, o protocolo UDP é vantajoso em situações onde a latência deve ser tão reduzida quanto possível, como na transmissão de informações de telemetria ou alertas de tráfego [20].

Num ecossistema de veículos conectados, uma abordagem híbrida pode trazer os melhores resultados. Por exemplo, atualizações críticas podem ser feitas através de protocolos de comunicação baseados em TCP, garantindo a fiabilidade, enquanto operações que não sejam críticas, mas cuja velocidade seja fulcral podem ser asseguradas através de UDP. A alavancagem da utilização dos dois protocolos traz uma abordagem otimizada da entrega de conteúdo, tendo em conta a fiabilidade e a performance.

3.2.2 HTTP

HTTP (*Hypertext Transfer* Protocol) é um protocolo de comunicação que opera na camada aplicacional do modelo OSI criado em 1999 pela World Wide Web Consortium para transmitir *hypermedia* (texto, imagens, vídeos) através da internet. É uma das fundações base da *World Wide Web*. [23] e a comunicação é feita utilizando como base 15 métodos [24]. Funciona no modelo de pedido/resposta no modelo computacional cliente/servidor. Este protocolo é baseado em TCP e funciona com a encriptação clássica TLS/SSL. [3]

A utilização do protocolo HTTP/HTTPS na implementação OTA traz vantagens na facilidade de utilização e alta compatibilidade, pois é um protocolo utilizado amplamente nas comunicações *web*, no entanto traz algumas desvantagens como *overheads* elevados e eficiência baixa, devido à sua natureza desconectada. No caso específico de HTTPS, embora melhore a segurança, é necessária uma camada de transporte separada (TLS) para cada pedido, aumentando o processamento para cada pedido e, consequentemente, a resposta [25]. Este tipo de problema pode ser mitigado com a utilização de sessões, onde cada elemento tem a sua sessão única e esta é validada pelo serviço que provisiona os dados, tendo um *time-to-live* específico, garantido segurança e velocidade do processo.

3.2.2.1 CoAP

CoAP (*Constrained Application Protocol*) é um protocolo ao nível aplicacional para IoT (adaptação do protocolo HTTP para IoT) baseado em UDP. É um protocolo binário que permite facilidade de transmissão através de HTTP. Este protocolo permite troca de mensagens entre pontos, mas também suporta um modo de *Broadcast*. As mensagens em CoAP podem ser confirmáveis ou não confirmáveis, de acordo com o caso de uso e é baseado no modelo de pedido/resposta. [3]

3.2.3 MQTT

MQTT (*Message Queueing Telemetry Transport*) é um protocolo de comunicação na camada aplicacional que funciona baseado tanto em transporte TCP/IP e UDP que utiliza TLS/SSL para segurança. MQTT baseia-se no modelo de publicação-subscrição, onde todas as mensagens vêm através de um *broker* MQTT [3].

A utilização de MQTT em *updates* OTA traz vantagens por ser um protocolo leve e utilizado amplamente no ambiente de IoT, pois é desenhado para trocas de mensagens contínuas e periódicas, que garante a segurança através de um único *handshake* TLS e que garante a continuidade do processo de forma segura, com *overhead* baixo [25]. A Figura 4 ilustra o processo de publicação/subscrição utilizado pelo protocolo MQTT.

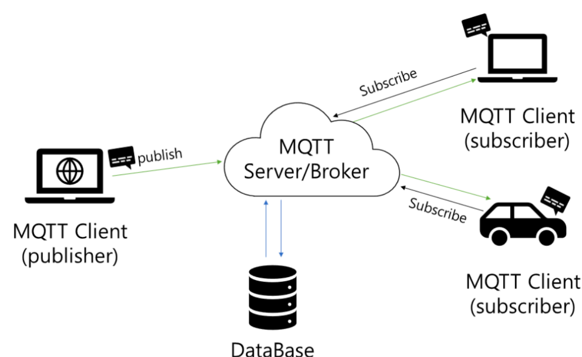


Figura 4 - Protocolo MQTT [25]

Este protocolo é bastante utilizado em aplicações IoT, onde os dispositivos com recursos limitados requerem uma forma eficiente e fiável de comunicar. O facto do pacote de transmissão de dados ser pequeno, suporte para níveis de qualidade do serviço (QoS) e capacidades de persistência de sessões faz dele um candidato ideal para veículos conectados [25]. O modelo publicação/subscrição traz também vantagens no que toca ao suporte de comunicação assíncrona, permitindo que os veículos recebam *updates* sem terem que estar constantemente conectados à rede, podendo apenas conectar em áreas onde a cobertura de sinal é superior [26].

3.2.4 QUIC

QUIC é um protocolo de rede da camada de transporte desenvolvido pela Google, projetado para melhorar a performance da comunicação na *internet*. Utiliza UDP como base para a conexão e tem como vantagem a velocidade alta, latência baixa e segurança integrada, semelhante ao TLS/SSL [27]. O QUIC foi criado para superar limitações de protocolos baseados em TCP, mais concretamente o atraso no estabelecimento de conexões e ineficiência no tratamento de perdas de pacotes. É utilizado por bastantes serviços recentes, como o Chrome e por plataformas de *streaming*, com a grande vantagem da fluidez para os utilizadores em redes com instabilidade e latência alta. Em redes de tempo real, QUIC aproxima o desempenho de UDP/DTLS, embora algumas interações com a camada de acesso possam degradar resultados. [28]

Em comunicação V2X, o QUIC traz vantagens, nomeadamente no que toca a transferências rápidas e eficientes e integração de segurança TLS 1.3, bem como a capacidade de lidar com mudanças frequentes de rede, como a troca entre redes Wi-Fi, 4G ou 5G, sem a necessidade de renegociar a conexão, graças ao suporte a identificadores de conexão persistentes. Essa característica é crucial para veículos em movimento, onde a qualidade da rede pode variar constantemente.

Além disso, o suporte de *multiplex* e a redução de *overhead* no controlo de congestionamento tornam o QUIC ideal para lidar com grandes volumes de dados, otimizando a entrega de atualizações e conteúdos para frotas inteiras de veículos conectados de maneira eficiente e confiável. [5]

Comparativamente com HTTP, que gere pedidos e respostas entre clientes e servidores, o QUIC é responsável pelo transporte da informação de forma segura e eficiente. Para além disso, o QUIC utiliza UDP em vez de TCP, que mitiga parte da latência no estabelecimento de conexões e problemas de *head-of-line blocking*.

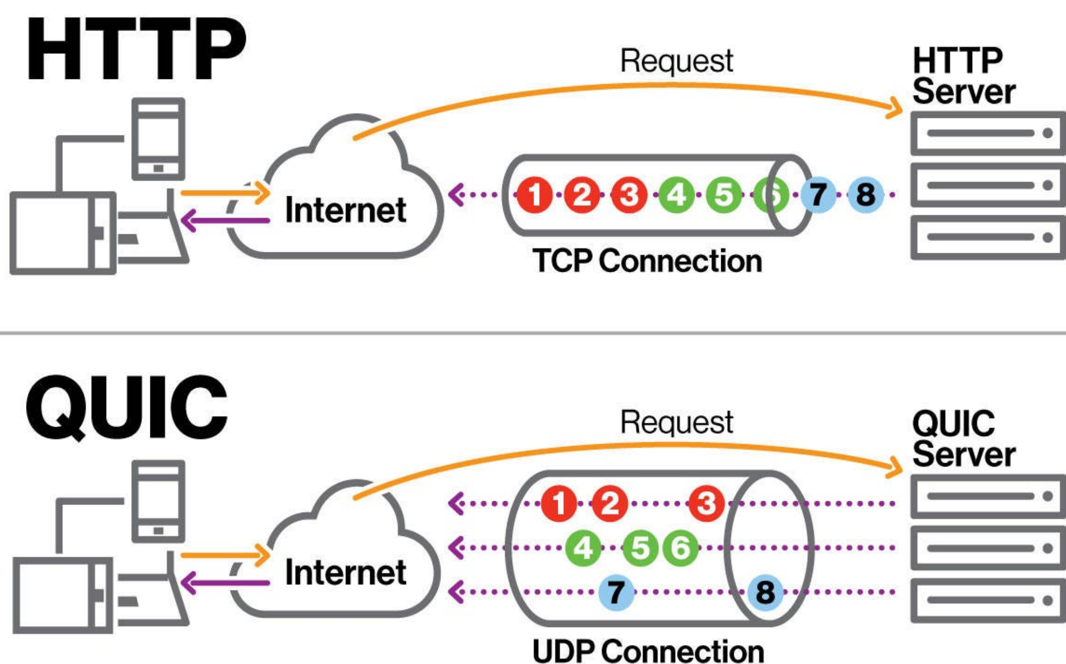


Figura 5 - Comparação entre HTTP e QUIC [29]

3.2.5 Comparação entre protocolos

De modo a comparar os vários protocolos de comunicação apresentados, a Tabela 2 apresenta as principais diferenças entre estes protocolos, tendo em conta as principais funcionalidades dos mesmos.

Tabela 2 - Comparação entre protocolos de comunicação

Funcionalidade	HTTP/2	MQTT	QUIC
Modelo de comunicação	Pedido/Resposta	Publicação/Subscrição	Pedido/Resposta
Protocolo de transporte	TCP	TCP (ou WebSocket)	UDP
Caso de uso primário	Serviços *web*, APIs, updates OTA	IoT, telemetria, dados orientados a eventos	Serviços *web*, *streaming*
Overhead das mensagens	Moderado	Baixo	Baixo
Latência	Baixo	Muito baixo	Muito baixo
Fiabilidade	Elevada (TCP assegura a entrega)	Elevada (Níveis de QoS disponíveis)	Elevada (retransmissão incorporada)
Segurança	TLS/SSL	TLS/SSL	Encriptação incorporada (TLS)
Eficiência energética	Moderada	Baixa	Elevada
Vantagens em *updates* OTA	Eficiente para entregas de dados estruturadas e APIs	Eficiente para telemetria e *updates* pequenos e frequentes	Eficiente para transferências de ficheiros grandes e

			updates de latência reduzida
Utilizações em veículos conectados	Ideal para APIs e *updates* OTA	Ideal para telemetria, controlo remoto e *updates* pequenos	Ideal para *updates* OTA grades e casos de latência baixa.

3.3 *Payload* – compressão e encriptação eficiente

Na comunicação entre veículos conectados e os diversos elementos envolvidos no processo, a informação que transita do ponto A ao ponto B é chamada de *payload*. Este conceito engloba qualquer tipo de carga de dados, como conteúdos multimédia, informações de assistência à condução, telemetria, dados de posicionamento geoespacial e outros tipos de informações essenciais para a operação do veículo ou para melhorar a experiência do utilizador.

A *payload* pode ser transportada em diferentes formatos, como binário, JSON, ou até mesmo conteúdos multimédia, dependendo da finalidade e da necessidade de processamento dos dados. É crucial que o transporte dessa informação assegure altos padrões de segurança e velocidade, uma vez que falhas podem ter consequências graves. Casos reais de *hacking* a veículos conectados evidenciam essa vulnerabilidade: num incidente, a exploração de uma falha na comunicação via rádio comprometeu a segurança de 1,5 milhões de veículos, forçando uma revisão em massa para mitigar os riscos. Este exemplo destaca a importância de implementar protocolos robustos e práticas avançadas de segurança para proteger os dados e garantir a integridade do sistema. [2].

De forma a melhorar a velocidade de transmissão e a utilização da largura de banda, a compressão da *payload* é uma das opções mais comuns, pois o processo de compressão reduz o tamanho final dos dados a serem transmitidos. No entanto, é necessário ter em consideração o tempo e esforço computacional necessário para a compressão ser feita relativamente ao ganho no tamanho da *payload*.

3.3.1 Compressão

Na comunicação entre o carro e o servidor, é necessário que o conteúdo transmitido seja o mais conciso possível, de forma a diminuir a utilização da largura de banda. Assim, é comum que o conteúdo que circula entre o veículo e o servidor seja comprimido, de forma que o tamanho da *payload* seja menor. Em dispositivos microcontrolados, a compressão pode reduzir o consumo energético total de transmissão, dependendo do tamanho e tipo de dados [30].

É necessário que os algoritmos de compressão sejam *lossless* pois reduzem o tamanho dos dados transmitidos, sem comprometimento da informação, garantindo que o conteúdo original pode ser reconstruído no momento da descompressão. Há também tipos de ficheiros que não

podem ser convertidos ou cuja conversão pode trazer deterioramento dos dados, daí haver uma necessidade elevada de verificar a integridade dos mesmos depois da descompressão [31].

3.3.1.1 gzip (*deflate*)

gzip (GNU zip) é um algoritmo de compressão baseado no encoding LZ77 e Huffman que combina o algoritmo de compressão *deflate* com metadata identificadora do ficheiro, respetiva integridade e atributos [32]. É bastante utilizado, pois é a forma mais comum de comprimir e descomprimir ficheiros em sistemas baseados em Unix. O custo computacional da compressão e da descompressão no gzip é relativamente moderado, sendo que é um algoritmo otimizado para velocidade e compressão moderadas, sendo ideal para aplicações como *web servers*, onde reduz o tempo de transferência com a compressão *on-the-fly*.

3.3.1.2 bzip2

Bzip2 é uma ferramenta de compressão que utiliza *encoding BWT* (Burrows-Wheeler Transform) e *Huffman* para atingir razões de compressão maiores que o gzip, com tempos de compressão e descompressão menores. Apesar disso, o poder computacional que requer é relativamente elevado. Devido à metodologia de *block-sorting* utilizada por este algoritmo, é bastante eficiente para dados mais pesados e é utilizado em cenários onde o tamanho de armazenamento é crucial [33].

3.3.1.3 LZMA2

LZMA2 é uma versão melhorada do LZMA (algoritmo em cadeia de Lempel-Ziv Markov) com elevados rácios de compressão e utilização de recursos configurável. Está por trás de formatos de compressão como .7z e XZ, permitindo uma elevada performance para ficheiros grandes. Ainda assim, é ideal para cenários onde a performance é secundária, devido ao uso substancial de memória e poder computacional. Este tipo de performance é atingido através da divisão dos dados em *chunks* para compressão paralela, permitindo um controlo refinado sobre a memória e utilização do CPU, sendo melhor para ambientes computacionais modernos, sendo necessária a compressão escalável [34].

3.3.1.4 BPE

BPE (*Byte Pair Encoding*) é um algoritmo de compressão de dados que substitui os pares de *bytes* adjacentes com um *byte* único não utilizado, reduzindo o tamanho dos dados de forma iterativa.

Este estudo [32] compara vários métodos de compressão, na memória utilizada e a razão de compressão da mesma.

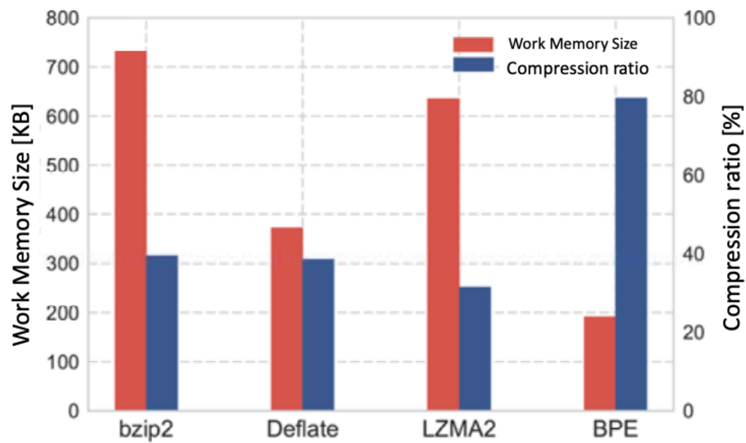


Figura 6 - Comparação entre algoritmos de compressão [32]

3.3.2 Compressão diferencial – *delta*

De forma a reduzir o tamanho do conteúdo entregue ao carro, podem ser feitos *updates delta*, que contém apenas a parte do software que foi alterada. Para isso, são utilizados algoritmos de compressão diferencial que, ao contrário dos algoritmos de compressão normais, encontram pontos de repetição entre dois ficheiros e criam um ficheiro *delta* com as diferenças [35]. Este tipo de atualizações prevê uma redução da quantidade de dados transmitidos em até 90% [36].

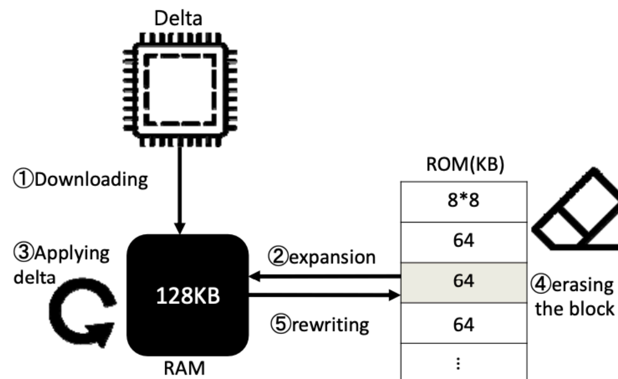


Figura 7 - Flow de um update delta [32]

3.3.2.1 bsdiff

bsdiff é uma ferramenta utilizada para compressão diferencial em ficheiros binários e utilizada para *delta updates*. Este tipo de ferramenta calcula as diferenças binárias entre dois ficheiros, sendo particularmente eficiente em ficheiros grandes com pequenas diferenças, como por exemplo, *patches* para *updates* de *software*. Utiliza a transformação de Burrows-Wheeler para maximizar a deteção de similaridades e atingir razões elevadas de compressão, que podem ser

utilizadas para reconstruir o ficheiro original [35]. Devido à utilização da ordenação de sufixo, torna-se um algoritmo dispendioso no que toca a poder computacional [1].

3.3.2.2 Xdelta

Xdelta é uma ferramenta de *diff* e *patch* desenhada para funcionar de forma eficiente tanto em dados binários como estruturados. Gera ficheiros *delta* através da identificação de mudanças entre dois *inputs* e codifica-os num formato compacto [37]. Ao contrário de ferramentas habituais de *diff*, o *xdelta* suporta tipos de dados mais complexos e oferece funcionalidades como correção de erros e gestão eficiente de ficheiros cujo tamanho é mais elevado, bem como gestão eficiente de recursos computacionais [38].

3.3.2.3 rsync

Rsync é uma ferramenta utilizada amplamente para a sincronização e transferência de ficheiros, alavancando a transferência *delta* para minimizar a quantidade de dados enviados pela rede. Funciona através da comparação dos ficheiros de origem e destino, transferindo apenas as diferenças entre os mesmos, em vez do ficheiro inteiro. É eficiente, versátil e tem suporte a ferramentas de encriptação, compressão e backups incrementais [31].

Neste estudo [37] é feita a comparação entre três protocolos de compressão: *rsync*, *xdelta* e *bsdiff*.

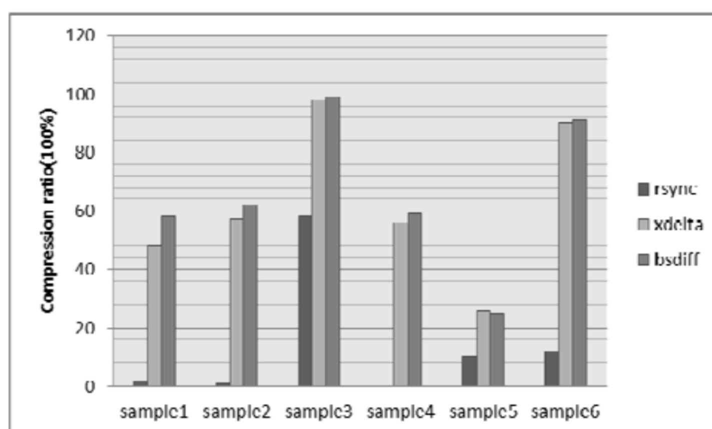


Figura 8 - Comparação do rácio de compressão de algoritmos *delta* [37]

3.4 PRISMA

A metodologia de revisão de literatura sistemática PRISMA foi utilizada de forma a encontrar literatura relevante relativamente ao problema proposto por esta tese. Assim, como explicado no capítulo anterior, a pesquisa foi feita em três bases de dados: *IEEE xplore*, *Google Scholar* e *ACM Digital Library*.

As pesquisas nestas bases de dados foram feitas com recurso a *queries* de pesquisa, onde os temas-chave foram incluídos, de acordo com a sintaxe que cada base de dados requere. Este tipo de pesquisa mostra-se mais eficiente no processo de pesquisa de artigos, pois filtra inicialmente uma grande percentagem de artigos que não são relevantes para o contexto da pesquisa.

De forma a facilitar o processo de escolha dos artigos, bem como a sustentação dos mesmos para determinadas perguntas, foi criada uma folha de *Excel*, onde os mesmos foram organizados e seriados. O resultado dessa seriação consta na Figura 9.

						Responde à pergunta...			
Título	Autor	Tipo	Publicação	Ano	É peer reviewd	1	2	3	Notas
Secure Over-the-Air Software Updates for Autonomous Vehicle Operating Systems	Eugene Ndego	Artigo de um Journal	DLBASR	2024	Sim	Sim	Sim	Sim	
MQTree: Secure OTA Protocol Using MQTT and MerkleTree	Yunje Shin e Sanghoon Jeon	Artigo de um Journal	MDPI - Sensors	2024	Sim	Sim	Sim	Sim	
Vertically Autoscaling Monolithic Applications with CaaSPER: Scalable Container-as-a-Service Performance Enhanced Resizing Algorithm for the Cloud	Anna Pavlenko et al	Artigo	SIGMOD-Companion	2024	Sim (?)	Excluído devido aos critérios de exclusão: Não fala do contexto de carro, apenas fala de scaling de monolitos			
Trajectory Planning of Automated Vehicles Using Real-Time Map Updates	MÁTYÁS SZÁNTÓ et al	Artigo de pesquisa		2023	Sim	Sim	Sim	Não	
A Big Data Architecture for Automotive Applications: PSA Group Deployment Experience	Amir Haroun, Ahmed Mostefaoui, François Dessables	Artigo de pesquisa	International Workshop on Distributed Big Data Management	2017	Sim	Sim	Sim	Não	Grupo Peugeot
QoSHVCP: Hybrid Vehicular Communications Protocol with QoS Prioritization for Safety Applications	Ahmad Mostafa et al	Artigo de pesquisa	ISRN Communications and Networking	2012		Não	kinda	Não	
OPC UA Publish-Subscribe and VSOME/IP Notify-Subscribe Based Gateway Application in the Context of Car to Infrastructure Communication	Alexandru Ioana	Artigo de um Journal	MDPI - Sensors	2020	Sim	Sim	Não	kinda	
A Hybrid Vehicle Hardware-in-the-Loop System with Integrated Connectivity for eHorizon Functions Validation	Lorenzo Brunell et al	Artigo	Alma Mater Studiorum Università di Bologna Archivio istituzionale della ricerca	2021	Sim	Não	Sim	Não	
An Event-Driven Inter-Vehicle Communication Protocol to Attenuate Vehicular Shock Waves	Markus Forster, Raphael Frank, Thomas Engel	Artigo de conferência	2014 International Conference on Connected Vehicles and Expo (ICCVE)	2014	Sim	kinda	Não	Não	
Multi-armed Bandit-Aided Near-Optimal Over-The-Air Updates in Multi-Band V2X Systems	Sherief Hashima et al	Artigo de conferência	ICCCI 2023 5th International Conference on Computer Communication and the Internet (ICCCI)	2023	Sim	Sim	Sim	Não	
Efficient delta based updates for read-only filesystem images	ELLINOR WESTERBERG	Tese de Mestrado	KTH Stockholm	2021	Sim	Sim	Não	Sim	Grupo BMW
Secure OTA Software Updates in Connected Vehicles: A Survey	Subir Halder et al	Artigo		2020	Sim	Sim	Não	Sim	
State of the art and trends of Vehicle Communication: Overview	Nadeza Yakusheva et al	Artigo de conferência	27th Telecommunications forum TELFOR 2019	2019	Sim	Sim	Sim	Não	

Figura 9 - Organização artigos PRISMA

Após a seriação de artigos através da metodologia PRISMA, os artigos foram analisados, lendo o conteúdo mais relevante, que permitiu a escrita da revisão de literatura do estado da arte. Finalmente, analisou-se o resultado, de forma a responder às perguntas de pesquisa previstas inicialmente.

Este método de pesquisa mostrou-se bastante relevante e fácil de implementar, uma vez que poupa tempo no processo de pesquisa e escrita, valorizando os tópicos mais importantes para a pesquisa em questão. A sua natureza metódica permite ao investigador compreender de

forma mais imersiva o tópico que está a ser investigado, permitindo que este seja abordado de forma mais consciente e completa.

3.4.1 Perguntas de pesquisa

3.4.1.1 Quais são os protocolos mais eficientes para a entrega de conteúdo em veículos conectados?

Os protocolos de comunicação mais eficientes para entrega de conteúdo *over-the-air* em veículos conectados são aqueles que balanceiam melhor a latência baixa, fiabilidade elevada e utilização eficiente da largura de banda, enquanto mitigam desafios típicos da indústria automotiva.

O protocolo MQTT, baseado no modelo de publicação/subscrição é uma escolha popular para este tipo de aplicações devido à natureza leve do mesmo, que permite execuções assíncronas e reduz a largura de banda consumida. É particularmente eficiente em cenários de que requerem *low overhead*, como por exemplo para atualizações em tempo real [25].

HTTP/2 e QUIC são também protocolos adequados para entregas *over-the-air*, oferecendo *streams multiplexed*, reduzindo a latência e mecanismos robustos de correção de erros [39]. HTTP/2 suporta compressão dos *headers* e prioriza as *streams* de dados, permitindo a entrega de cargas maiores, mantendo a eficiência. QUIC integra melhorais na camada de transporte, reduzindo os tempos de *handshake* e resiliência à perda de pacotes. A latência reduzida e a manutenção de conexões tornam-no ideal para veículos cuja rede varie durante as atualizações [4].

A escolha de protocolo pode depender do requisito, onde um modelo de publicação/subscrição possa ser mais eficiente para transmissão de informação em tempo real, no entanto para *updates* maiores, HTTP/2 ou QUIC podem ser mais eficientes [40].

3.4.1.2 Como é que diferentes protocolos de comunicação (e.g., MQTT, HTTP/2, QUIC) se comparam em termos de latência, utilização da largura de banda e fiabilidade em entregas a veículos conectados?

No que toca à latência, o QUIC é um dos protocolos mais rápidos, devido ao seu *design* baseado em UDP e *overhead* associado ao *handshake* reduzido [4]. Ao contrário dos protocolos baseados em TCP, QUIC estabelece uma conexão segura de forma mais rápida, reduzindo o tempo que o início da transmissão demora. HTTP/2, embora baseado em TCP, oferece latência baixa devido ao *multiplexing* e reutilização de conexões. A latência de MQTT é ligeiramente mais alta devido ao foco na fiabilidade e persistência, não deixando, por isto, de ser competitivo em cenários onde a *payload* é reduzida [25].

Quanto à utilização de largura de banda, MQTT destaca-se devido ao tamanho compacto da mensagem e inexistência de *headers* adicionais, sendo ideal para redes constrangidas [3].

HTTP/2 reduzi a largura de banda através da compressão dos *headers*. QUIC otimiza o transporte de dados de forma eficiente, reduzindo o *overhead* através de *multiplexing*.

Finalmente, em termos de fiabilidade, MQTT destaca-se substancialmente devido aos três níveis de qualidade de serviço, permitindo às aplicações a escolha entre uma entrega que garante a entrega da mensagem ou uma entrega onde o maior esforço vai ser feito para a entrega com reconhecimento da receção da mensagem [1]. HTTP/2 e QUIC oferecem mecanismos robusto de recuperação de erros, embora QUIC lide melhor com a perda de pactos, devido à funcionalidade de isolação de *streams* [3].

3.4.1.3 Quais são os algoritmos de compressão mais eficientes para minimizar o tamanho da carga sem comprometer a integridade dos dados durante as entregas de informação?

No que toca à eficiência do algoritmo de compressão, gzip destaca-se devido ao seu balanço entre velocidade e razão de compressão, onde a redução do ficheiro é elevada sem *overhead* de processamento associado [32].

LZMA2 é uma opção eficiente, particularmente para *payloads* elevados, como *firmware* ou *updates* de mapas. A capacidade de entregar rácios de compressão mais elevados torna-o ideal para a minimização de custos e transferência de dados. No entanto, as velocidades de compressão e descompressão mais lentas fazem com que não seja ideal para casos de atualizações urgentes [37].

A utilização de algoritmos de compressão diferencial, onde apenas a parte do ficheiro que foi utilizada é enviada é também uma solução bastante capaz de transmitir a informação de forma eficiente. [1]Assim, o algoritmo bsdiff destaca-se por calcular as diferenças binárias entre dois ficheiros, sendo particularmente eficiente em ficheiros grandes com pequenas diferenças, elevando as razões de compressão. Estes tipos de algoritmos têm como desvantagem poder ser apenas utilizado em ficheiros com semelhanças, como é o caso de ficheiros de mapas ou de *updates* de *software* [37].

3.4.2 Discussão

A escolha de protocolos e algoritmos de compressão para a entrega *over-the-air* em veículos conectados requer um balanço cuidadoso entre eficiência, fiabilidade e adaptabilidade para os desafios do setor automotivo. Protocolos como HTTP/2, MQTT e QUIC apresentam vantagens distintas, com MQTT a destacar-se na telemetria leve e cenários de comando e HTTP/2 e QUIC ao apresentarem soluções robustas para *payloads* grandes e *updates* mais rápidos. A mobilidade inerente aos veículos, a transição entre várias redes e/ou interrupções de sinal reiteram a utilização de protocolos como QUIC, onde a manutenção da sessão e a perda de pacotes é gerida de forma eficiente. Para além disso, a alavancagem de protocolos com

qualidade de serviço ajustável, como é o caso de MQTT, assegura que vários requisitos de *updates* são atingidos, quer no ponto de vista de segurança, quer no ponto de vista de fiabilidade.

Igualmente, a aplicação de algoritmos de compressão melhora a entrega *over-the-air*, garantindo a integridade dos dados aquando da minimização do tamanho. A utilização de *deltas* traz também uma vantagem relativamente grande pois envia para o veículo apenas a informação necessária, poupando recursos computacionais e de rede.

3.4.3 Conclusão

A entrega eficiente de conteúdo *over-the-air* a veículos conectados é um elemento fundamental dos ecossistemas modernos de veículos conectados, permitindo atualizações contínuas para aprimorar a funcionalidade, a segurança e a experiência do condutor e demais utilizadores. Este estudo destaca a importância de selecionar protocolos de comunicação e algoritmos de compressão adequados para atender aos requisitos dinâmicos das atualizações OTA. Protocolos como MQTT, HTTP/2 e QUIC oferecem opções diversificadas para lidar com diferentes necessidades de latência, largura de banda e fiabilidade, enquanto algoritmos de compressão *lossless*, como gzip, otimizam a transferência de dados ao reduzir os tamanhos de *payload* sem comprometer a integridade.

A integração de sistemas adaptativos que escolhem dinamicamente protocolos e métodos de compressão com base no tipo de informação a transmitir é fulcral na entrega de conteúdo a veículos conectados. O estudo feito neste capítulo permite entender quais são as vantagens e desvantagens de cada uma das abordagens, permitindo que uma abordagem híbrida seja desenvolvida, de forma atingir um cenário otimizado de comunicação entre o veículo e a rede que o rodeia.

4 Desenho da Solução

4.1 Proposta de Arquitetura

A solução desenhada tem como objetivo avaliar diferentes estratégias de comunicação entre veículos conectados e o *backend*, simulando o processo de entrega de conteúdo *over-the-air*. Para tal, foi criada uma arquitetura modular que permite testar múltiplas combinações de protocolos de comunicação, algoritmos diferenciais (*deltas*), mecanismos de compressão e estratégias de *caching*. Utilizou-se *blobs*, estruturas representativas de porções de um mapa que compõe a zona onde o carro está a circular. Estes mapas, sendo versionados, permitem que se faça *patches* entre a versão anterior de um *blob* e a atual.

4.1.1 Backend

O *backend* produzido para esta solução visa ser robusto e capaz de ser adaptável para os vários casos de teste a validar. De forma a conseguir simular um cenário produtivo de um *backend* capaz de entregar conteúdo a milhares de veículos, foi necessário montar uma *stack* robusta. Assim, decidiu-se que a melhor solução para várias instâncias acederem aos ficheiros seria uma solução de *cloud*, responsável pela disponibilização dos ficheiros em tempo real. Visto que o preço de soluções de *cloud* seria demasiado elevado para o cenário de teste em questão, decidiu-se utilizar uma solução que emulasse as interações à *cloud*.

Entre as várias soluções de *cloud*, devido a restrição tecnológica da empresa, selecionou-se a solução da *cloud* da Amazon, AWS [41]. Para a simular sem custo, utilizou-se a solução de *Localstack* [42], que é uma estrutura que permite através de máquinas virtuais – *containers*. Esses *containers* são criados através da plataforma Docker [43], onde estes *containers* permitem correr aplicações num ambiente controlado.

A solução é constituída por:

- ***Protocol Adapter***: expõe *endpoints* HTTP a serem consumidos pelo veículo e integra cliente/servidor MQTT (através do *broker*);
- ***Content Registry***: estrutura de armazenamento dos *blobs*, com a utilização de versões para cada *blob* (eg: *10/blob1.bin, 11/blob1.bin*). O armazenamento é feito através da utilização de *Localstack*, hosteado em Docker;
- **Serviço de *Patch***: responsável por encapsular os algoritmos de *patching*;
- **Serviço de compressão**: responsável por aplicar os algoritmos de compressão;
- **Serviço de *cache***: responsável por guardar em *cache* (*Spring simple cache*) os patches gerados, através da utilização do nome do *blob* antigo e do *blob* novo (artefactos necessários à criação e unicidade do *patch*).

4.1.2 Veículo conectado

O veículo conectado será simulado, neste cenário, através da utilização de um Raspberry Pi 5, que emula as várias unidades do veículo.

O Raspberry Pi [44] foi escolhido pela facilidade de configuração e ambiente de teste, que permite uma quantidade considerável de configurações, bem como semelhanças técnicas com o tipo de controladores presentes em veículos. Para executar os pedidos ao *backend*, utilizou-se Python, pela facilidade de configuração e de forma a parametrizar os testes, de acordo com os vários cenários propostos, recorreu-se a *shell scripting*, também por ser simples de configurar e do baixo gasto de memória.

É constituído por:

- ***Network Adapter***: cliente MQTT ou HTTP que consome/pede a informação necessária;
- ***In-memory store***: responsável por guardar os *patches* e *blobs* já descarregados do servidor e gerir as respetivas versões;
- **Cliente OTA**: responsável por fazer os pedidos, validar as respostas e retirar métricas (tamanho, latência);
- ***Decompressor applier***: aplica o algoritmo de descompressão necessário;
- ***Patch applier***: aplica o algoritmo de diferenciação utilizado, de forma a fazer *restore* ao *blob*, utilizando o *blob* que tem em memória.

A Figura 10 mostra o diagrama de componentes da arquitetura proposta, com destaque para a adaptação de protocolos, serviços de criação de *patches*, compressão e mecanismo de hot zones.

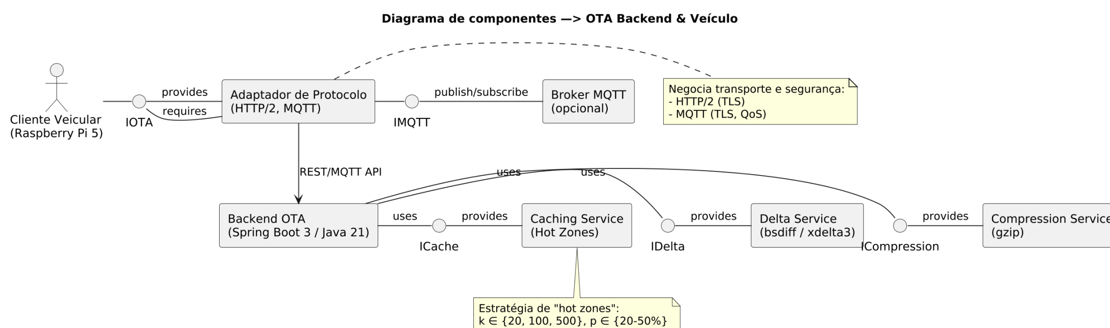


Figura 10 - Diagrama de componentes OTA *Backend* & Veículo

4.2 Protocolos de comunicação (HTTP/MQTT)

Na solução proposta foram implementados dos métodos distintos de comunicação entre o *backend* e o veículo, de forma a conseguir avaliar qual dos dois métodos produz o melhor resultado, de acordo com os vários cenários propostos, sendo estes HTTP e MQTT.

No caso específico de HTTP, o *backend* disponibiliza uma REST API, através da qual o veículo pode solicitar *blobs*. Esta API expõe três *endpoints*:

```
GET /ota/blob?blobId=123&version=10&encoding=gzip
```

Este *endpoint* é responsável por entregar ao carro um *blob* e possui os seguintes parâmetros:

- *blobId* (obrigatório): Identifica o ID do *blob* que se está a pedir ao *backend*;
- *version* (opcional): Identifica a versão do *blob* que o veículo pretende. No caso do veículo não enviar este parâmetro, irá sempre entregar a versão mais recente do *blob*.
- *encoding* (opcional): Indica qual o *encoding* no qual a resposta deve ser entregue. No caso do veículo não indicar *encoding*, o *backend* não irá aplicar nenhum *encoding* no *blob*;

```
GET /ota/version
```

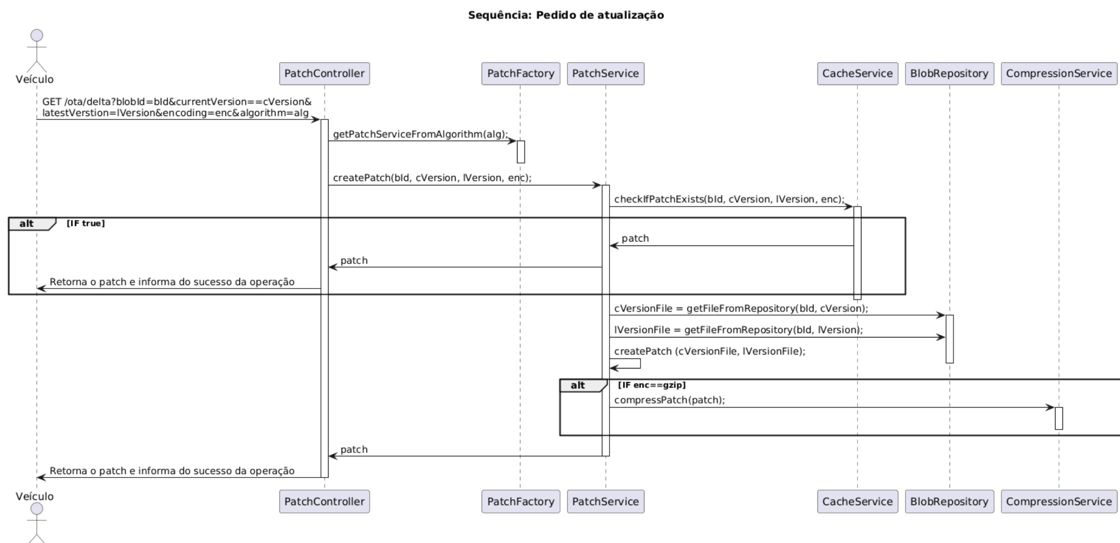
Este *endpoint* não recebe parâmetros e apenas indica a versão mais recente do mapa.

```
GET
/ota/delta?blobId=123&currentVersion=10&latestVersion=12&encoding=gzip&algo
rithm=alg
```

Este *endpoint* é responsável por entregar ao carro um *patch*, tendo em conta as duas versões entre as quais será criado o *patch*. Possui os seguintes parâmetros:

- *blobId* (obrigatório): Identifica o ID do *blob* que se está a pedir ao *backend*;
- *currentVersion* (obrigatório): Identifica a versão do *blob* que o veículo tem;
- *latestVersion* (obrigatório): Identifica a versão da qual o carro pretende que o *backend* crie um *patch*;
- *encoding* (opcional): Indica qual o *encoding* no qual a resposta deve ser entregue. No caso do veículo não indicar *encoding*, o *backend* não irá aplicar nenhum *encoding* no *blob*;
- *algorithm* (obrigatório): Indica qual o algoritmo de *patching* a utilizar (*bsdiff* ou *xdelta*).

O Raspberry Pi atua como consumidor direto destes *endpoints*, onde envia pedidos e mede a latência, tamanho do *payload* e tempo de processamento.



4.3 Algoritmos de compressão e diferenciais (gzip, bsdiff, xdelta3)

No âmbito da entrega de conteúdo *over-the-air*, a escolha dos algoritmos de compressão e diferenciação é determinante para reduzir a latência e a largura de banda consumida. Estes mecanismos permitem otimizar a transmissão de informação entre o *backend* e o veículo, a fim de evitar o envio repetido de ficheiros completos sempre que ocorre uma atualização.

A compressão tradicional, representada neste trabalho via *gzip*, atua sobre um único ficheiro, explorando redundâncias internas para reduzir o seu tamanho. O seu principal benefício é a facilidade de integração e a ampla utilização em sistemas distribuídos. Contudo, a compressão apresenta um custo de processamento acrescido e revela-se menos eficaz em ficheiros de pequena dimensão, como *patches* diferenciais. Nestes casos, o *overhead* associado pode mesmo aumentar o *payload* final, comprometendo a eficiência global da solução.

De forma complementar, os algoritmos diferenciais procuram apenas as diferenças entre duas versões de um mesmo ficheiro, construindo um *patch* que, quando aplicado ao ficheiro de base existente no veículo, resulta na versão atualizada. Este princípio reduz de forma significativa o volume de dados transmitido, sobretudo em cenários em que as versões diferem apenas em pequenas partes.

Entre os algoritmos diferenciais escolhidos para este estudo, destacam-se *bsdiff* e *xdelta3*. O algoritmo *bsdiff* é reconhecido por gerar *patches* particularmente compactos, sendo adequado quando o objetivo principal é reduzir ao máximo o tamanho da atualização. No entanto, o seu processo de geração é mais exigente em termos de recursos, o que pode impactar a escalabilidade do *backend* em cenários com elevada concorrência. Já o *xdelta3* oferece outro compromisso: a produção de *patches* ligeiramente maiores, mas beneficia de maior rapidez na sua criação e aplicação, apresentando-se como uma solução mais pragmática em ambientes online.

Como apresentado no capítulo 4.2 na apresentação da REST API, para que a compressão diferencial seja feita, é necessário que o carro tenha, em memória, uma versão diferente da mais recente disponibilizada pelo *backend*. Assim, se isto acontecer, irá requisitar um *patch*, onde deverá indicar o algoritmo de compressão diferencial a ser utilizado, bem como se pretende que o *payload* venha comprimido. Este tipo de estratégia que visa utilizar o maior benefício das estratégias de compressão diferencial trará grandes vantagens na implementação, uma vez que poderá criar cenários adaptáveis à realidade do momento da simulação.

4.4 Estratégia de *caching* e conceito de *hot zones*

O conceito de *hot zones* tem como principal objetivo poupar a capacidade de processamento e de armazenamento do *backend*. Assim, os *blobs* ou patches que são mais utilizados pelos clientes serão guardados em *cache*, e quando forem pedidos mais do que uma vez, o *backend* não terá de criar novamente o *patch* ou ir ao armazenamento buscar a informação, uma vez que já a terá carregada.

De forma a emular este conceito, a *hot zone* terá um tamanho variável, onde *k* representará o número de elementos presente na *hot zone*, com $k \epsilon \{20, 100, 500\}$. A escolha dos elementos será feita de forma aleatórios (de entre a lista de elementos existentes), de acordo com o tamanho de *k* previsto para o cenário em teste.

Uma vez que nos testes que estão a ser feitos, os *blobs* pedidos são aleatórios, e para conseguir simular o comportamento regular dos veículos a pedirem muitas vezes o mesmo *blob* (ex: os veículos pedem o *blob* que corresponde no mapa a uma área com bastante tráfego), acrescentou-se o conceito de probabilidade do *blob* pertencer à *hot zone* como sendo p, com $p \in \{0.2, \ 0.3, 0.4, 0.5\}$.

4.5 Métricas e critérios de avaliação

A avaliação da solução proposta exige a definição clara de métricas objetivas que permitam comparar diferentes combinações de protocolos, compressão e algoritmos diferenciais. O principal objetivo é quantificar de forma rigorosa a eficiência da entrega OTA, medindo não apenas o impacto na rede, mas também no processamento do veículo. Para tal, foram estabelecidas duas principais métricas:

- **Tempo total do pedido**

Esta métrica corresponde, essencialmente, ao intervalo de tempo decorrido desde o momento em que o Raspberry Pi inicia o pedido ao *backend* até ao instante em que o ficheiro final está em memória, incluindo, quando relevante, a restituição e validação do ficheiro. Inclui, portanto, o tempo de transmissão do *payload*, a latência da rede, o processamento no *backend* (geração de *patch*, compressão, etc) e o processamento no veículo, quando aplicável. Esta métrica reflete de forma integrada a experiência real do veículo durante uma atualização.

- **Tamanho do *payload* transmitido**

O tamanho total em *bytes* do conteúdo transmitido entre o *backend* e o veículo (*blobs* completos, comprimidos, *patches* diferenciais, etc). Esta métrica é fundamental para avaliar a eficiência da largura de banda, permitindo entender o impacto da compressão no processo de envio OTA.

A recolha destas métricas permitirá a construção de séries de dados que podem ser posteriormente analisadas em termos estatísticos, facilitando a comparação entre diferentes cenários de combinações de estratégias. Com base nestas medições, será possível identificar *trade-offs* entre tempo de atualização e volume de dados transmitidos, avaliando assim quais as estratégias mais adequadas em função das restrições de rede e requisitos do sistema OTA.

4.6 Alternativas de *design*

Durante o desenho da solução proposta, foram consideradas diversas alternativas que poderiam ter sido seguidas tanto a nível da arquitetura como das tecnologias adotadas. A escolha final recaiu sobre uma abordagem modular de Java 21 com Spring Boot no *backend* e um Raspberry Pi 5 como emulação do veículo, por permitirem equilíbrio entre simplicidade de desenvolvimento, suporte a bibliotecas necessárias e capacidade de medição das métricas relevantes. No entanto, outras opções poderiam ter sido exploradas.

4.6.1 Alternativas *backend*

No que toca ao *backend*, poderia ter sido utilizada uma *stack* em Node.js ou Go, frequentemente associadas a sistemas com requisitos de baixa latência e elevada concorrência. Estas opções poderiam simplificar a integração com ferramentas externas, como binários de *bsdiff* ou *xdelta*, ou reduzir o consumo de recursos, em comparação com a Java Virtual Machine (JVM).

Outra alternativa seria recorrer a soluções que não recorram a servidores constantemente ligados, que apenas são ligados quando existe carga, ou seja sistemas *serverless* (por exemplo, AWS Lambda ou Azure Functions), simulando um ambiente ainda mais próximo de produção em larga escala. Isso permitiria avaliar como é que a elasticidade da *cloud* poderia impactar a geração de *patches* sob alta carga, mas à custa de maior complexidade no *setup* experimental.

Para além destas alternativas, poderia ainda ser considerada a utilização de Rust como linguagem para o *backend*. Rust tem vindo a ganhar bastante relevância devido à sua elevada performance e ao foco na segurança de memória, permitindo maior previsibilidade no consumo de recursos. Esta abordagem poderia ser vantajosa em operações intensivas, tais como a geração e aplicação de *patches* diferencias, reduzindo a sobrecarga no servidor. Contudo, a curva de aprendizagem acentuada e a integração com *frameworks* empresariais dificultariam a sua adoção no contexto apresentado.

Adicionalmente, no caso de Node.js, embora a sua arquitetura *event-driven* simplifique a gestão de milhares de conexões simultâneas e seja amplamente suportada pela comunidade, apresenta limitações quando sujeita a cargas computacionais pesadas, como compressão ou algoritmia diferencial, onde a performance tende a degradar-se. Quanto a Go, as vantagens oferecidas prendem-se pela leveza do modelo de concorrência, o que poderia proporcionar ganhos na execução de múltiplos pedidos em paralelo, mantendo um consumo de memória controlado.

4.6.2 Alternativas simulação veículo

Embora o Raspberry Pi 5 ofereça maior capacidade de processamento, seria possível optar por microcontroladores mais limitados (como Arduinos). Esta escolha teria permitido estudar o impacto dos algoritmos diferenciais em dispositivos com fortes limitações de memória e CPU, aproximando-os ainda mais do cenário de veículos conectados.

Contudo, e apesar de terem sido feitas várias implementações e testes em Arduinos durante o desenvolvimento desde documento, as limitações do mesmo não permitiram a utilização dos algoritmos de descompressão, principalmente devido a restrições de memória e arquitetura.

Assim, a ideia inicial seria que o Raspberry Pi servisse de *backend*, onde iria escalar automaticamente, tivesse papel também de *load balancer* e que iria gerar uma rede, comportando-se como um *Access Point* (AP), onde os Arduinos se iriam conectar. Esta solução foi produzida inicialmente, no entanto apresentou bastantes conflitos com o *NetworkManager* do Raspberry Pi, pois ao estar conectado a uma rede via LAN e a gerar uma via Wi-Fi, o mesmo não era capaz de as gerir.

O código em C para o Arduino foi produzido, como é possível analisar através da Figura 11, no entanto aquando da implementação dos algoritmos diferenciais, surgiram demasiados erros, associados à memória do mesmo. Foi criada uma função em C *bspatch_embedded*, por falta de bibliotecas compatíveis, no entanto, apesar da implementação, não se mostrou útil, pois os erros de falta de memória para os *headers* eram constantes. Assim, e após não encontrar nenhuma solução para o problema, e ler *feedback* de outros internautas quanto ao assunto, decidiu-se desistir da utilização de Arduino para emular um veículo conectado.

```
void connectWiFi() {
  Serial.print("Connecting to Wi-Fi: ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println();
  Serial.print("Connected! IP address: ");
  Serial.println(WiFi.localIP());
}
void getBytes(const uint8_t *data, size_t length) {
  String url = String(serverPath) + "?" + param1Name + "=" + param1Value +
               "&" + param2Name + "=" + param2Value;

  Serial.print("GETing to: ");
  Serial.println(url);

  client.beginRequest();
  client.get(url);
  client.sendHeader("Content-Type", "application/octet-stream");
  client.sendHeader("Content-Length", length);
  client.beginBody();
  client.endRequest();

  int statusCode = client.responseStatusCode();
  String response = client.responseBody();

  Serial.print("Status code: ");
  Serial.println(statusCode);
  Serial.print("Response: ");
  Serial.println(response);
}
```

Figura 11 - Código em C para Arduino

4.6.3 Protocolos adicionais

Apesar da implementação ter sido focada em HTTP, o que se deveu principalmente a restrições empresariais, podia ter sido implementada em MQTT ou QUIC, explorando as suas capacidades de redução de latência em ambientes com elevada perda de pacotes. Também seria possível avaliar a utilização de CoAP, uma alternativa que também é bastante utilizada em cenários de IoT.

4.6.4 Comparação de arquiteturas

As alternativas analisadas ao longo desta secção evidenciam que não existe uma única arquitetura universalmente superior para suportar entregas OTA em veículos conectados. Cada escolha tecnológica apresenta um conjunto de compromissos entre desempenho, complexidade e escalabilidade que devem ser ponderados em função do contexto de utilização.

A opção por um *backend* em Java com Spring Boot destaca-se pela maturidade do ecossistema, facilidade de integração com bibliotecas empresariais e capacidade de expansão modular. Esta abordagem mostrou-se adequada para ambientes de teste controlados, garantindo fiabilidade e rapidez de desenvolvimento, ainda que com algum custo associado à sobrecarga da JVM.

Alternativas como Node.js oferecem simplicidade na integração de módulos externos e grande eficiência em cenários I/O bastante intensivos, mas podem revelar limitações quando sujeitas a cargas computacionais pesadas, como compressão e criação de *patches* binários. Já Go apresenta-se como uma solução mais equilibrada para operações concorrentes, proporcionando latência reduzida e baixo consumo de memória, embora careça de um ecossistema tão vasto quanto o de Java ou Node.js. Rust, por sua vez, acrescenta ganhos relevantes em performance e segurança de memória, sendo particularmente interessante em cenários de elevado rigor computacional, mas a sua adoção implica maior complexidade no desenvolvimento.

De forma análoga, também no que respeita à simulação do veículo, o Raspberry Pi 5 constituiu uma escolha válida e mais eficiente, pela capacidade de processamento e proximidade às condições de um sistema real, mas alternativas como Arduino poderiam permitir avaliar de forma mais rigorosa os impactos de algoritmos diferenciais em ambientes altamente limitados. Ainda assim, as restrições de memória e arquitetura destes últimos inviabilizaram a adoção prática no presente trabalho.

4.6.5 Conclusão

Estas alternativas demonstram que a arquitetura escolhida não é a única possível, mas sim uma decisão de compromisso entre facilidade de implementação, controlo experimental, restrições empresariais e de tecnologia. A escolha por uma solução modular e extensível garante, no entanto, que futuras iterações do trabalho possam incorporar algumas destas opções, enriquecendo substancialmente o estudo.

5 Implementação

A implementação da solução teve como principal objetivo validar, em ambiente controlado, a arquitetura definida no capítulo anterior. Para tal, foram concebidos cenários experimentais que permitiram avaliar diferentes combinações de algoritmos diferenciais, compressão, utilização de *cache* e políticas de *hot zones*, recorrendo às métricas definidas previamente. Esta fase foi essencial para aferir a robustez da solução, bem como para compreender os *trade-offs* inerentes à escolha de diferentes estratégias.

O *backend* foi desenvolvido em Java 21 com Spring Boot 3, utilizando Localstack para emulação de serviços de armazenamento em S3. O veículo foi simulado através de um Raspberry Pi 5, que executou *scripts* em Python. Estes *scripts* foram chamados de forma paramétrica, através de *shell scripting*, com parâmetros como o tipo de pedido efetuado (versão, *blob* ou *delta*), o algoritmo diferencial (*bsdiff* ou *xdelta3*), a aplicação (ou não) de compressão, a utilização de *cache* e os parâmetros associados ao conceito de *hot zone* (*k*, *p*, número de iterações).

A infraestrutura que serviu de suporte a esta arquitetura é apresentada na Figura 13.

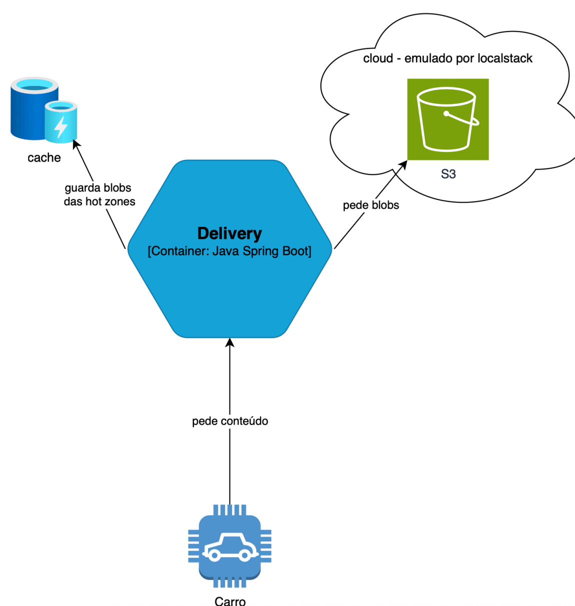


Figura 12 - Diagrama de interação carro-*backend*-*cloud*

5.1 Ambiente de testes

O ambiente de testes utilizado foi desenhado para refletir um cenário próximo de produção, mas em escala reduzida, de forma a garantir repetibilidade e controlo sobre os parâmetros avaliados.

5.1.1 Backend

O *backend* emulou a infraestrutura utilizada pelo veículo conectado, onde se utilizou *Spring Simple Cache*, de forma a simular o conceito de *hot zones*.

Os dados que simularam os *blobs* existentes foram criados através de um algoritmo para o qual era passado um *n* que era o número de *blobs* a ser criado, um *v* que era a versão em questão, e o algoritmo era responsável por pegar num *blob* em *JSON* inicial e modificá-lo ligeiramente, de forma que gerasse *blobs* novos, mas pouco diferentes, para validar os cenários de criação de *patches*, como apresentado na Figura 13.

```java
public void generateAndUpload(String bucket, String keyPrefix, int numberOfElements, String path) throws IOException {
    ObjectMapper mapper = new ObjectMapper();
    byte[] base = Files.readAllBytes(Path.of(path));
    JsonNode baseNode = mapper.readTree(base);

    IntStream.rangeClosed(1, numberOfElements).forEach( i -> {
        try {
            ObjectNode variant = baseNode.deepCopy();
            mutateSlightly(variant, i, ThreadLocalRandom.current());
            byte[] payload = toBytes(variant, mapper);
            String id = variant.path( "id").asText(String.valueOf( i * 1000));
            String key = keyPrefix.endsWith("/") ? keyPrefix + "blob-" + id + ".json" : keyPrefix + "/blob-" + id + ".json";
            System.out.println(key);
            s3.putObject(
                    PutObjectRequest.builder()
                            .bucket(bucket)
                            .key(key)
                            .contentType( "application/json")
                            .build(),
                    RequestBody.fromBytes(payload)
            );
        } catch (Exception e) {
            throw new RuntimeException("Failed at i=" + i, e);
        }
    });
}
```

Figura 13 - Código de geração de blobs aleatórios

A execução de algoritmos diferenciais foi feita através da invocação externa de binários (*bsdiff* e *xdelta3*), geridos por *ProcessBuilder*. A utilização deste tipo de estrutura é evidenciada através da Figura 14.

```java
@Override  1 usage  António Barbosa (CTW)
@Cacheable(value = "javaPatchCache")
public File encryptFile(final String oldVer, final String newVer) {
    try {
        File oldVerFile = File.createTempFile( prefix: "oldVer",  suffix: ".json");
        try (FileOutputStream fos = new FileOutputStream(oldVerFile)) {
            fos.write(storageRepository.getObjectFromS3( bucket: "tile-store", oldVer).get());
        }
        File newVerFile =  File.createTempFile( prefix: "newVer",  suffix: ".json");
        try (FileOutputStream fos = new FileOutputStream(newVerFile)) {
            fos.write(storageRepository.getObjectFromS3( bucket: "tile-store", newVer).get());
        }

        File deltaFile = File.createTempFile( prefix: "xdelta-delta-",  suffix: ".bin");
        deltaFile.deleteOnExit();
        ProcessBuilder pb = new ProcessBuilder(
                 ...command: "xdelta3", "-f", "-e", "-s",
                oldVerFile.getAbsolutePath(),
                newVerFile.getAbsolutePath(),
                deltaFile.getAbsolutePath()
        );
        pb.redirectErrorStream(true);
        Process p = pb.start();
```

Figura 14 - Execução do algoritmo diferencial através de ProcessBuilder

Para além disto, o *backend* teve de ser capaz de analisar os pedidos e escolher qual o serviço responsável a ser utilizado, tendo em conta o algoritmo diferencial a utilizar. Para isso, criaram-se dois principais serviços, *PatchXDelta3ServiceImpl* e *PatchBSDiffServiceImpl*, onde ambos implementam o *PatchService*, com um método, *createPatchFile*, como apresentado na Figura 15.

```java
public interface PatchService {  6 usages  2 implementations  António Barbosa (CTW) *
    File createPatchFile(String oldVersionPath, String newVersionPath);  1 usage  2 implementations
```

Figura 15 - Interface *PatchService*

Para que o controlador fosse capaz de decidir qual o serviço a utilizar, criou-se uma *Factory*, responsável por analisar o parâmetro que define o algoritmo a utilizar e, assim, selecionar o serviço correto, como apresentado na Figura 16.

```java
@AllArgsConstructor  2 usages  António Barbosa
@Component
public class PatchServiceFactory {

    private PatchBSDiffServiceImpl patchBSDiffService;
    private PatchXDeltaServiceImpl patchXDeltaService;

    public PatchService getPatchServiceFromAlgorithm(Algorithm algorithm) {  1 usage  António Barbosa
        if(algorithm.equals(Algorithm.BSDIFF)){
            return patchBSDiffService;
        }else if (algorithm.equals(Algorithm.XDELTA3)){
            return patchXDeltaService;
        }else {
            throw new RuntimeException("Unsupported algorithm: " + algorithm);
        }

    }
}
```

Figura 16 - Utilização de Factory para distinguir entre serviços por algoritmo diferencial

O *backend* disponibilizou uma REST API que foi consumida pelos simuladores de veículos, para que estes pudessem obter informação relativamente ao *blob* e à versão dos *blobs* disponibilizada naquele momento. Este versionamento de *blobs* foi o que permitiu que se construíssem *patches* diferenciais entre versões, apenas com a informação binária necessária. Assim:

- **GET */ota/version*** para obter a versão mais recente;
- **GET */ota/blob*** para descarregar *blobs* completos;
- **GET */ota/delta*** para solicitar *patches* diferenciais entre versões;

Um exemplo da forma como os *endpoints* foram disponibilizados está presente na Figura 17.

```java
@GetMapping(path = "/delta", produces = MediaType.APPLICATION_OCTET_STREAM_VALUE)
public ResponseEntity<ByteArrayResource>  createAndStreamEncryptedPatch(
        @RequestParam("blobId") String blobId,
        @RequestParam("currentVersion") String currentVersion,
        @RequestParam("latestVersion") String latestVersion,
        @RequestParam("encoding") Encoding encoding,
        @RequestParam("algorithm") Algorithm algorithm) throws Exception {

    PatchService patchService = this.patchServiceFactory.getPatchServiceFromAlgorithm(algorithm);
    File patchFile = patchService.createPatch(blobId, currentVersion, latestVersion, encoding);
    byte[] data = Files.readAllBytes(patchFile.toPath());

    //Checksum production - MD5
    MessageDigest digest = MessageDigest.getInstance( algorithm: "MD5");
    digest.update(data);
    final String checksum = DatatypeConverter.printHexBinary(digest.digest()).toUpperCase();

    ByteArrayResource resource = new ByteArrayResource(data);
      return ResponseEntity.ok()
            .contentLength(patchFile.length())
            .contentType(MediaType.APPLICATION_JSON)
            .header(CHECKSUM_HEADER,  ...headerValues: checksum)
            .header(HttpHeaders.CONTENT_DISPOSITION,  ...headerValues: "inline; filename=\"" + patchFile.getName() + "\"")
            .body(resource);
}
```

Figura 17 - Exemplo REST API de produção de *patches*

5.1.2 Simulação cliente OTA

O Raspberry Pi desempenhou o papel de cliente OTA, comunicando com o *backend* através de HTTP. A escolha do Raspberry Pi foi principalmente influenciada por limitações técnicas na utilização de Arduino, como insuficiência de memória para algoritmos diferenciais, custo-benefício e também a possibilidade de utilização de linguagens como Python, que facilitam o processo de produção, execução e *debugging* dos *scripts*.

O Raspberry Pi estava conectado a uma rede local através da utilização de Wi-Fi, com a qualidade máxima de rede (devido à proximidade ao distribuidor). O *backend* estava também conectado a esta rede, com a atribuição de um endereço estático, para que o cliente OTA pudesse conectar-se sempre ao mesmo endereço na rede.

A decisão de implementar apenas na rede local deveu-se a limitações na construção de uma rede customizada, na qual fosse possível emular *jitter* e variações na rede. Também devido a falta de recursos não foi possível emular uma rede celular, que mais se aproximava da realidade dos veículos conectados. Assim, os dados recolhidos não contam com qualquer tipo de interferência de rede, uma vez que os pedidos foram feitos, executados e recolhidos dentro da mesma rede.

Para testar os cenários relevantes, foram executados vários cenários que foram automatizados através de um *shell script*, que invocava um *script* em Python que executa pedidos de acordo com os parâmetros adequados. Para o caso dos pedidos *delta*, o *shell script* especificava qual o algoritmo diferencial a utilizar, se devia aplicar compressão *gzip*, bem como os valores de *k*, *p* e o número de iterações.

Foram gerados múltiplos cenários através da combinação dos seguintes parâmetros:

- $k \in \{20, 100, 500\}$
- $p \in \{0.2, 0.3, 0.4, 0.5\}$
- Iterações $\in \{1000, 2000, 5000, 8000, 10000\}$
- Algoritmos diferenciais $\in \{bsdiff, xdelta3\}$
- Compressão $\in \{gzip, nenhuma\}$

Estes cenários foram escolhidos por combinarem todas as variáveis em teste para a validação da qualidade de soluções de *hot zones* e de compressão diferencial. Os valores escolhidos para *k*, *p* e número de iterações tiveram como base alguns valores sugeridos pela empresa, baseados em análise de resultados anteriores, bem como valores que se assemelhassem a situações tanto quanto reais.

Nos cenários de *hot zones*, a multiplicação destes fatores resulta em 240 cenários distintos, cobrindo diferentes combinações de utilização de *cache*, algoritmos diferenciais e compressão. A forma como o Raspberry Pi, representativo do veículo conectado, foi utilizado para simular os pedidos feitos está presente na Figura 18.

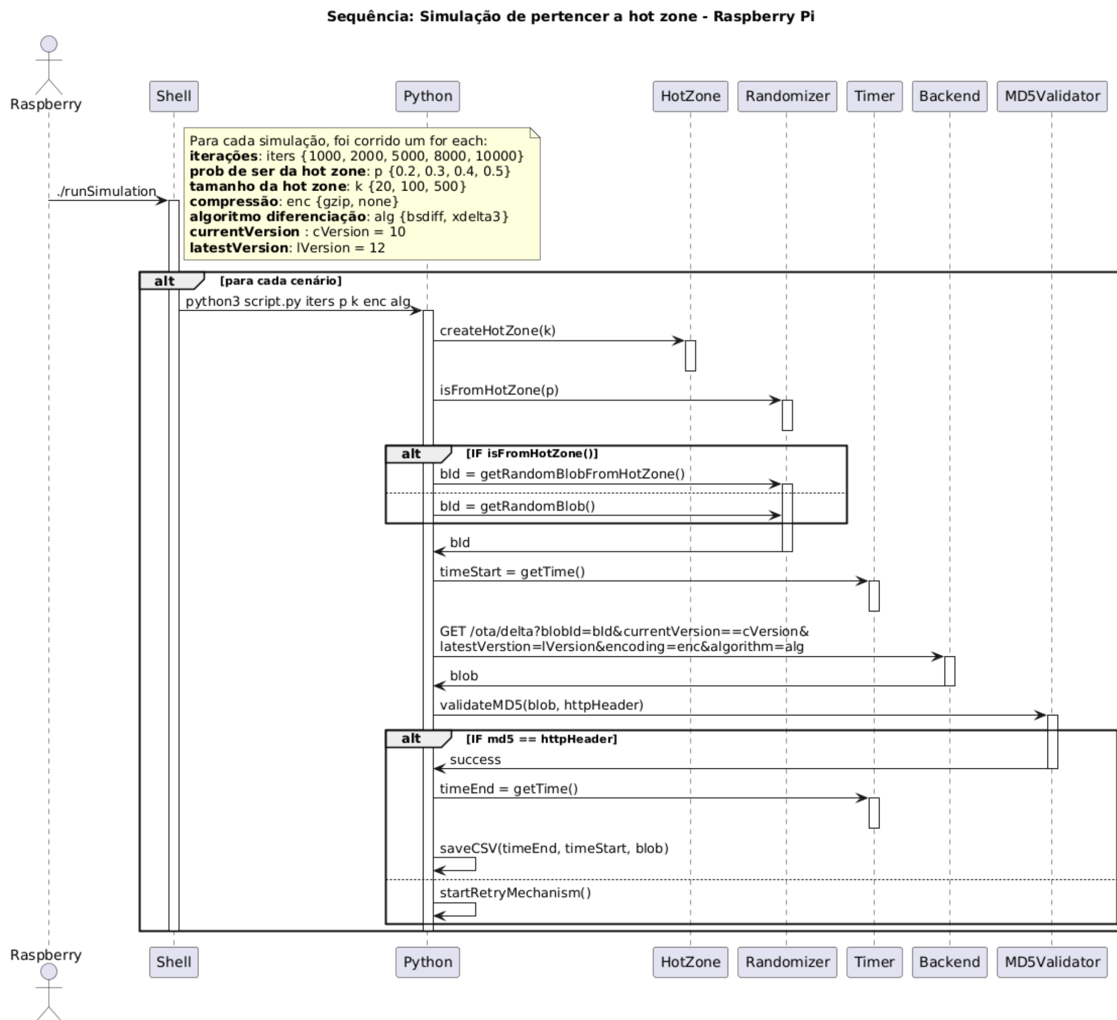


Figura 18 - Diagrama de sequência da interação do Raspberry Pi com o *backend* em cenários de *hot zones*

5.2 Ferramentas utilizadas

O desenvolvimento e experimentação desta solução recorreu a um conjunto diversificado de ferramentas de *software* e *hardware*, que permitiram implementar, executar e analisar os cenários definidos.

No *backend*, foi utilizado Java 21, em conjunto com Spring Boot 3, que garantiu a criação da REST API e integração com mecanismos de *cache*. O desenvolvimento foi realizado no IDE InteliJ IDEA, pelas capacidades avançadas de *debugging*, gestão de dependências e integração com sistemas de controlo de versão (git).

O armazenamento de *blobs* e versões foi emulado com recurso a Localstack, executado em Docker, permitindo simluar serviços de AWS sem custos associados. A gestão do código

produzido foi assegurada através de GitHub, que centralizou o versionamento e documentação técnica do produto.

Para a execução de algoritmos diferenciais, foram utilizados os binários de *xdelta3* e *bsdiff*, instalados através de gestores de pacotes (ex: *apt-get*) e invocados no backend através da classe de *ProcessBuilder*. A utilização de *ProcessBuilder* deve-se principalmente a restrições que surgiram no desenvolvimento do código, uma vez que com a utilização de bibliotecas nativas de *xdelta3* e *bsdiff* para Java, surgiram bastantes condicionantes que impediram o bom funcionamento da aplicação. Esta solução pareceu não aumentar o tempo de resposta, não trouxe contrapartidas de desenvolvimento, tornando-o até mais fácil. A compressão com *gzip* também foi realizada com recurso ao binário, dado que as bibliotecas nativas apresentaram inconsistências em determinados cenários, bem como dificultaram o processo de descompressão no cliente (Raspberry Pi).

Do lado do cliente, o veículo foi simulado através de Raspberry Pi, que executou scripts em *Python*, produzidos no PyCharm e que foram invocados de forma paramétrica a partir de *shell scripts*, permitindo a configuração dinâmica dos cenários.

Para a recolha das métricas, foram utilizados ficheiros .csv (*comma-separated values*), nos quais foram registados os resultados de cada execução. Não foi utilizada qualquer ferramenta externa de análise de rede ou benchmarking, garantindo simplicidade e controlo total sobre as variáveis de teste.

5.3 Configuração dos testes

5.3.1 Justificação na escolha dos parâmetros

Os testes foram desenhados para abranger múltiplas combinações de parâmetros, conforme descrito no capítulo 5.1. Para cada cenário experimental foram definidos:

- $k \in \{20, 100, 500\}$ (tamanho da *hot zone*)
- $p \in \{0.2, 0.3, 0.4, 0.5\}$ (probabilidade de pertencer à *hot zone*)
- $iterações \in \{1000, 2000, 5000, 8000, 10000\}$
- $algoritmo\ diferencial \in \{bsdiff, xdelta3\}$
- $compressão \in \{gzip, nenhuma\}$

O tamanho da *hot zone* selecionado, *k*, foi dimensionado com a utilização de base a 100 elementos, 20 para simular um cenário minimalista e 500 para simular um cenário de *stress*. Esta escolha permitiu uma obtenção e análise de resultados mais fidedigna. Também se escolheram valores com ordens de grandeza diferentes para simular cenários onde ter uma *hot zone* demasiado grande pudesse não ser benéfico.

O valor da probabilidade de pertencer à *hot zone*, *p*, visa simular o contraste entre o tráfego urbano e o tráfego disperso. Um cenário com um *p* baixo pode representar um meio rural, onde o *blob* não pertence à *hot zone* por ser poucas vezes descarregado, não se justificando ficar guardado em memória. Em contrapartida, um cenário com um *p* elevado representa um cenário urbano, onde a probabilidade do *blob* a ser descarregado pertencer à *hot zone* é elevada, pois aquela zona é representativa de uma zona do mapa com muitos acessos, como por exemplo, um centro de cidade em hora de ponta.

O número de iterações variou entre 1000 e 10000, de forma a assegurar validade e significância estatística, uma vez que repetir os cenários poucas vezes poderia levar a resultados inconsistentes ou levar à existência de valores que, pelos mais variados motivos, pudessem ter tido algum erro de medição.

Para além disto, foram corridos também cenários base de pedidos de *blobs* normais, sem qualquer tipo de algoritmo de diferenciação, para haver uma *baseline* de comparação. A importância da *baseline* prende-se, nomeadamente, pela necessidade de compreender um cenário inicial sem a implementação do conceito de *hot zones* e de *caching*, onde os veículos pedem única e exclusivamente a informação mais atualizada relativa àquele *blob* representativo da zona onde se encontram.

Os *blobs* utilizados tinham, em média, 2.5kB, em formato JSON, variando o conteúdo de cenário para cenário, mantendo uma estrutura semelhante. Esta variação foi introduzida de forma controlada, assegurando que os algoritmos diferenciais fossem testados com diferentes graus de similaridade entre versões.

As medições de tempo foram efetuadas no cliente (Raspberry Pi), registando o instante de início do pedido até ao momento em que o ficheiro final estava disponível em memória. Etsa abordagem integra o tempo de transmissão, latência de rede, tempo de compressão/*patching* no *backend* e tempo de descompressão/restituição no cliente, refletindo a experiência real de um veículo conectado.

5.3.2 Execução sequencial vs concorrente

A execução foi sequencial e não concorrente, decisão tomada para preservar a validade das medições. Caso os cenários tivessem sido executados em paralelo, a *cache* do *backend* seria influenciada por pedidos simultâneos, comprometendo a avaliação justa do impacto da política de *hot zones*.

A opção por executar os cenários de forma sequencial assegurou a validade dos resultados, ao evitar a contaminação da cache com pedidos concorrentes. No entanto, esta escolha limita a generalização para contextos reais, onde múltiplos veículos comunicam em simultâneo com o *backend*. Em cenários concorrentes, a pressão sobre a memória da cache e sobre os recursos de CPU seria substancialmente maior, podendo levar a degradação da latência ou até falhas de serviço. Assim, como trabalho futuro, será essencial a realização de *load tests* em ambientes

concorrentes controlados, simulando milhares de veículos a efetuar pedidos simultâneos. Este tipo de experimentação permitirá avaliar a robustez da solução, quantificar a escalabilidade da *cache* e identificar os limites de *throughput* da arquitetura, fornecendo dados mais próximos da realidade operacional de sistemas OTA em larga escala.

5.4 Testes unitários no *backend*

5.4.1 Estratégias de testes unitários

A validação de uma aplicação não deve ser feita apenas no final da mesma, mas sim incorporada no ciclo de desenvolvimento. No caso do *backend* implementado, a adoção de testes unitários surge como uma prática indispensável para garantir robustez e fiabilidade, reduzindo a probabilidade de falhas em sistemas produtivos e aumentando a confiança nos resultados obtidos. Apesar do foco principal deste trabalho não ter sido a criação de uma suíte de testes, é relevante explicitar a estratégia que poderia ser seguida e apresentar exemplos que ilustram a forma como os testes unitários e de integração poderiam ser estruturados.

A estratégia de testes visou privilegiar a separação entre lógica de negócio e dependências externas. Serviços responsáveis pela geração de *patches*, compressão e interação com sistemas de armazenamento devem ser testados de forma isolada, utilizando mecanismos de *mocking* para simular respostas de dependências externas. Assim, torna-se possível validar cenários de sucesso, erro ou exceções sem necessidade de aceder a sistemas externos como Localstack ou AWS reais. Adicionalmente, os testes devem incluir casos de integração com os *endpoints* REST expostos, de forma a verificar a coerência entre camadas e o correto funcionamento das diferentes dependências quando combinadas.

No caso concreto da obtenção de *patches*, a lógica de chamadas à AWS através das *frameworks* utilizadas pode ser *mockada* nos serviços para conseguir isolar a lógica de negócio, uma vez que não se mostra relevante testar o funcionamento da *framework* externa.

Um exemplo de teste unitário com recurso a Mockito pode ser ilustrado na Figura 19, onde a chamada ao *StorageRepository* é *mockada* e o que está a ser testado, o serviço, tem a anotação de *@InjectMocks*. Assim, todas as chamadas ao repositório são emuladas pelo método *when*, que informa o Mockito de como proceder quando recebe um pedido ao serviço que está *mockado*.

```java
@ExtendWith(MockitoExtension.class)
@AllArgsConstructor
public class PatchServiceTests {

    @Mock
    private StorageRepository storageRepository;
    @InjectMocks
    private PatchService patchService;
    @Test
    void testPatchService(){
        byte[] myFile = new byte[100];
        byte[] myOtherFile = new byte[100];
        when(storageRepository.getObjectFromS3("my-bucket", "my-prefix")).thenReturn(Optional.of(myFile));
        when(storageRepository.getObjectFromS3("my-bucket", "my-other-prefix")).thenReturn(Optional.of(myOtherFile));
        this.patchService.createPatchFile("my-prefix", "my-other-prefix");
    }
}
```

Figura 19 - Exemplo utilização *Mockito*

Para além de testes unitários, é igualmente importante validar o comportamento da API como um todo. Utilizando o módulo de testes do Spring Boot, é possível simular chamadas HTTP aos *endpoints* expostos, avaliando a resposta gerada. Este tipo de teste, embora mais pesado, assegura que a integração entre controladores, serviços e camadas de infraestrutura funciona corretamente. Um exemplo deste tipo de testes é apresentado na Figura 20.

```java
@SpringBootTest
@AutoConfigureMockMvc
class PatchControllerIntegrationTests {

    @Autowired
    private MockMvc mockMvc;

    @MockitoBean
    private PatchService patchService;

    @Test
    void shouldReturnPatchWhenValidDeltaRequest() throws Exception {
        File fakePatch = Files.createTempFile("test", "patch").toFile();
        Mockito.when(patchService.createPatchFile(Mockito.any(), Mockito.any()))
                .thenReturn((fakePatch));

        mockMvc.perform(MockMvcRequestBuilders.get("/ota/delta")
                        .param("blobId", "123")
                        .param("currentVersion", "1")
                        .param("latestVersion", "2")
                        .param("algorithm", "xdelta3"))
                .andExpect(status().isOk())
                .andExpect((ResultMatcher) content().bytes(Files.readAllBytes(fakePatch.toPath())));
    }
}
```

Figura 20 - Exemplo utilização MockMVC

Neste teste de integração, o serviço responsável por criar *patches* é substituído por um *mock*, garantindo que o *endpoint* responde de forma coerente com os parâmetros recebidos.

No caso com as interações com Localstack ou AWS S33, estas podem também ser *mockadas* em testes unitários, evitando a necessidade de levantar containers ou serviços externos. No entanto, para validação de integração mais próxima da realidade, podem ser utilizados perfis dedicados e dependências como *TestContainers*, que permitem levantar Localstack de forma isolada e descartável durante a execução da suíte de testes.

A execução dos testes é realizada através de Maven, integrando o ciclo de compilação habitual. A configuração típica inclui dependências para JUnit, Mockito e Spring Boot Test no ficheiro pom.xml, assegurando que a suíte de testes pode ser executada de forma automatizada tanto em ambiente local como em pipelines de integração contínua.

5.5 Estratégia de CI/CD

A adoção de práticas de integração e entrega contínua (CI/CD) é hoje considerada fundamental para garantir a fiabilidade, a replicabilidade e a escalabilidade do processo de desenvolvimento de software. No caso desta dissertação, a implementação de uma *pipeline* de CI/CD foi estruturada de forma a cobrir todo o ciclo de vida de desenvolvimento, desde a compilação inicial do código, passando pela execução de testes automáticos, pela análise de qualidade, até à disponibilização do artefacto em ambiente *containerizado* com recurso a Docker e Localstack.

A opção pela utilização de GitHub Actions teve em consideração a sua integração nativa com o sistema de controlo de versões GitHub, utilizado como repositório oficial do projeto. Este serviço permite a criação de *workflows* declarativos com os passos a executar no processo de CI/CD em ficheiros YAML, assegurando que cada alteração submetida ao repositório desencadeia um conjunto de verificações pré-definidas (por exemplo, a execução da *pipeline* de testes sempre que se adiciona código novo). Esta abordagem elimina a necessidade de *pipelines* externas ou ferramentas adicionais (como, por exemplo, Jenkins), reduzindo a complexidade de execução.

O primeiro passo desta pipeline consistiu em ir buscar as alterações mais recentes do código (*checkout*) e configuração da máquina onde os testes irão correr, com a instalação de Java, linguagem utilizada, *maven*, a *build tool* escolhida e outras relevantes. Assim, após configurar tudo, executou-se o comando *mvn clean compile*, garantindo que todas as dependências estão corretamente resolvidas e que o código compila sem erros. Para além disso, assegura a produção de um artefacto consistente (ficheiro JAR), que poderá ser posteriormente utilizado em fases de execução e *deploy*. Foram também corridas algumas verificações de código, como por exemplo a execução de *checkstyle*, que verifica que o código produzido segue algumas normas de formatação e de organização, bem como a utilização de *Javadoc*, artefacto essencial para a documentação.

Após isto, a *pipeline* executa os testes através do *lifecycle mvn verify*, responsável por executar os testes unitários e de integração desenvolvidos para validar a qualidade do *software* produzido. Esta etapa é crítica, uma vez que valida automaticamente a lógica de negócio

desenvolvida, incluindo o funcionamento dos serviços de geração de *patches* diferenciais, compressão e mecanismos de *cache*. O resultado é um conjunto de relatórios que asseguram que novas alterações não introduziram erros no sistema.

Após a execução dos testes, é efetuada a análise de qualidade de código através do SonarQube, que recolhe métricas relativas a duplicação de código, complexidade, cobertura de testes e potenciais vulnerabilidades de segurança. Para que o código seja considerado válido, deverá ter uma cobertura de testes no código novo de, pelo menos, 80%. A Figura 21 apresenta um exemplo da utilização da pipeline.

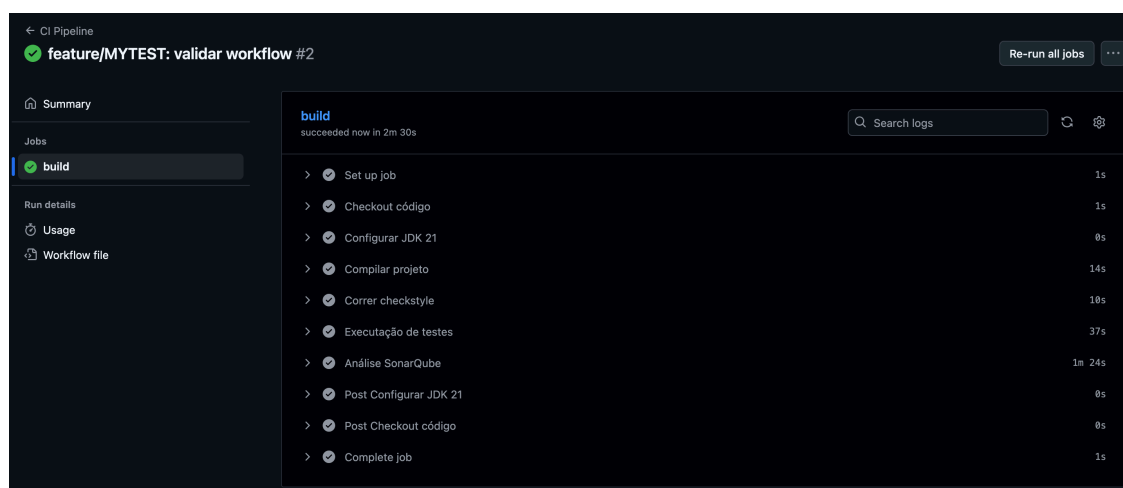


Figura 21 - Exemplo *pipeline* validação

A utilização de uma pipeline CI/CD estruturada desta forma traz benefícios claros para o projeto. Por um lado, garante que cada alteração é validada de forma consistente, aumentando a confiança no sistema. Por outro, promove que seja possível identificar possíveis erros e ter referência dos mesmos para futuro, uma vez que todos os resultados ficam disponíveis na interface do GitHub. Finalmente, a integração de Docker e Localstack permite validar cenários mais realistas, aproximando a experimentação académica de práticas utilizadas no ambiente produtivo real.

5.6 Automação da recolha de métricas

A recolha de métricas foi automatizada através dos *scripts* em Python, que registavam os resultados de cada execução em ficheiros CSV, como apresentado na Figura 22.

```
def save_avg_results(file_name, total_ms, total_patch_bytes, iters, scenario):  1 usage  António Barbosa
    avg_time = total_ms / iters
    avg_patch_size = total_patch_bytes / iters
    file_path = file_name + ".csv"

    file_exists = os.path.isfile(file_path)

    with open(file_path, "a", newline="") as f:
        writer = csv.writer(f, delimiter=';')
        if not file_exists:
            writer.writerow(["scenario", "avg_time", "avg_patch_size"])
        writer.writerow([f"{scenario}", f"{avg_time:.2f}", f"{avg_patch_size:.2f}"])
```

Figura 22 - Exemplo de código *Python* para guardar os dados em CSV

Cada linha do CSV correspondia a um pedido efetuado, com o seguinte cabeçalho:

```
scenario;avg_time;avg_patch_size
```

- **scenario**: identificador do cenário (ex: algoritmo, compressão, valores de k e p);
- **avg_time**: tempo total decorrido entre o início do pedido e a disponibilidade do ficheiro no Raspberry Pi (em milissegundos);
- **avg_patch_size**: tamanho do *payload* transferido entre o *backend* e o cliente (em *bytes*).

A Tabela 3 exemplifica uma parte de um ficheiro de métricas:

Tabela 3 – Exemplo de ficheiro de métricas

scenario	avg_time	avg_patch_size
HTTP_bsdiff_GZIP_k20_p0.2_1000	60.56	263.38
HTTP_xdelta3_NO_k500_p0.3_8000	22.48	255.48
HTTP_bsdiff_NO_k20_p0.3_2000	29.04	240.35
HTTP_xdelta3_GZIP_k500_p0.2_1000	57.67	278.27

A validação da integridade dos ficheiros reconstituídos foi realizada através do *hash* MD5, incluído no *header* da resposta do *backend*. O Raspberry Pi calculava o MD5 do ficheiro recebido e comparava-o com o valor recebido no *header*, garantindo a consistência e fiabilidade dos resultados.

A recolha de métricas foi mantida em formato bruto, sem pós processamento estatístico automático, permitindo flexibilidade futura para análises adicionais. Esta opção também garantiu que os dados registados correspondiam diretamente às medições experimentais, sem transformação ou filtragem intermédia.

5.7 Métodos de captura e análise

Na recolha de métricas, nomeadamente na questão da medição do tempo, foi medido o instante inicial da execução do pedido e o instante final, que posteriormente foram subtraídos. Para cada execução do cenário, o valor foi somado e foi feita a média de tempo do número de execuções.

Não foram utilizadas ferramentas externas de *benchmarking* devido à necessidade de simplificação do ambiente de teste, por limitações técnicas e temporais, mas seria bastante valioso que a informação recolhida via CSV pudesse popular uma base de dados, para melhor análise estatística futura, bem como integração direta e menos possibilidade de falha na recolha de informação.

Na captura e análise de dados, foram encontradas algumas limitações que poderão levar a resultados menos aproximados da realidade. Assim, foi levantado que a infraestrutura de teste não tem em consideração a captação de *jitter* associado à rede utilizada, característica bastante relevante em redes "reais", onde as interferências são bastante comuns e que acabam por levantar problemas de envio de informação.

Também não foi analisada até à exaustão a sobrecarga do sistema onde as simulações estão a correr, bem como não foram limitados os recursos a serem utilizados (tanto o *backend* como o veículo conectado utilizaram todos os recursos disponíveis nas máquinas). Este tipo de análise poderia ser relevante, uma vez que nos veículos conectados, a capacidade de processamento e de memória é bastante limitada.

Como integração futura, poderia ser útil a utilização de ferramentas de recolha e análise de métricas, como por exemplo Prometheus, ou de análise de rede, como por exemplo Wireshark.

5.8 Considerações finais sobre a implementação

Em suma, para esta implementação de solução, foram relevantes algumas decisões críticas que moldaram o desenvolvimento. A utilização do Raspberry Pi em vez de Arduino revelou-se determinante, dada a limitação de memória e processamento dos microcontroladores, permitindo maior flexibilidade na execução dos algoritmos diferenciais e na produção de código.

A execução sequencial dos cenários foi também uma escolha essencial para não comprometer os resultados relativos às *hot zones*, ainda que tenha limitado a aproximação a cenários reais onde múltiplos veículos comunicam em simultâneo com o *backend*. O armazenamento em CSV, embora não constitua a solução mais avançada, revelou-se vantajoso pela simplicidade, velocidade de integração e facilidade de análise. Para além destes pontos, a opção por excluir a variabilidade da rede assegurou o controlo experimental necessário para validar a interação entre *backend* e veículo conectado, mas afastou o estudo das condições adversas de conectividade, frequentes em ambientes reais. Esta decisão, juntamente com a execução

sequencial, evidencia limitações que deverão ser ultrapassadas futuramente através da realização de testes de carga e da simulação de redes com latência, *jitter* e perda de pacotes.

Também a utilização de ficheiros CSV poderá evoluir para ferramentas de monitorização mais avançadas, como Prometheus ou Grafana, permitindo recolha em tempo real e integração direta com mecanismos de orquestração dinâmica. Em síntese, a implementação privilegiou a simplicidade e a reprodutibilidade, garantindo resultados fiáveis para os objetivos definidos, mas reconhecendo compromissos que deverão ser endereçados em fases futuras para aproximar o estudo das condições reais de veículos conectados.

6 Análise de Resultados

6.1 *Baseline* de resultados

Os resultados da *baseline* constituem o ponto de partida fundamental para todas as comparações subsequentes. Como descrito no capítulo 5.3.1, este cenário representou o caso mais simples possível: a entrega de *blobs* completos sem qualquer tipo de compressão, sem aplicação de algoritmos diferenciais e sem utilização de cache no *backend*. Desta forma, foi possível estabelecer uma referência que traduz de forma aproximada o comportamento de um sistema tradicional, onde cada pedido resulta na transmissão integral do conteúdo, independentemente da sua dimensão ou de versões anteriores já armazenadas no veículo.

Os resultados obtidos para esta simulação apresentaram um tamanho médio de *payload* de 2487.88 *bytes* e um tempo médio de resposta de 88.32ms (em 63 mil execuções dos cenários). Estes resultados são expectáveis e permitem indicar o cenário onde o veículo não tem qualquer tipo de informação (impedindo-o de pedir *delta*). O valor médio do *payload* tende para o valor médio de um *blob* (2.5k), uma vez que apenas os *blobs* são enviados, sem qualquer tipo de informação a mais ou a menos.

Importa notar que esta *baseline*, apesar de simples, revela desde logo limitações significativas em termos de eficiência. Em cenários com múltiplos veículos, a transferência contínua de *blobs* completos, mesmo quando apenas pequenas partes do ficheiro sofrem alterações, resultaria num consumo de largura de banda desnecessário e num esforço adicional de processamento.

6.2 Resultados com cache (*hot zones*)

6.2.1 Visão global e comparação com a *baseline*

A Figura 23 e Figura 24 sintetizam o comportamento médio das quatro combinações avaliadas: *gzip_bsdiff*, *gzip_xdelta3*, *no_cache_bsdiff* e *no_cache_xdelta3*. Os resultados indicam, de forma consistente, que a utilização de *deltas* reduz substancialmente a latência e o volume transferido, face ao envio integral do *blob*.

Na Figura 23, são apresentados os tempos médios de resposta. Estes tempos situam-se, aproximadamente, entre 21 e 30 ms sem compressão e 36-45 ms com compressão, dependendo do algoritmo diferencial. Em comparação com os 88.32 ms da *baseline*, os ganhos de latência atingem os 50-75%. De forma mais concreta, *no_compression_xdelta3* evidenciou os menores tempos (entre 21 e 26 ms), seguido de *no_compression_bsdiff* (25-29 ms). A aplicação de *gzip* aumenta sistematicamente a latência, com *gzip_xdelta3* a rondar 36-39 ms e *gzip_bsdiff* 41-45ms.

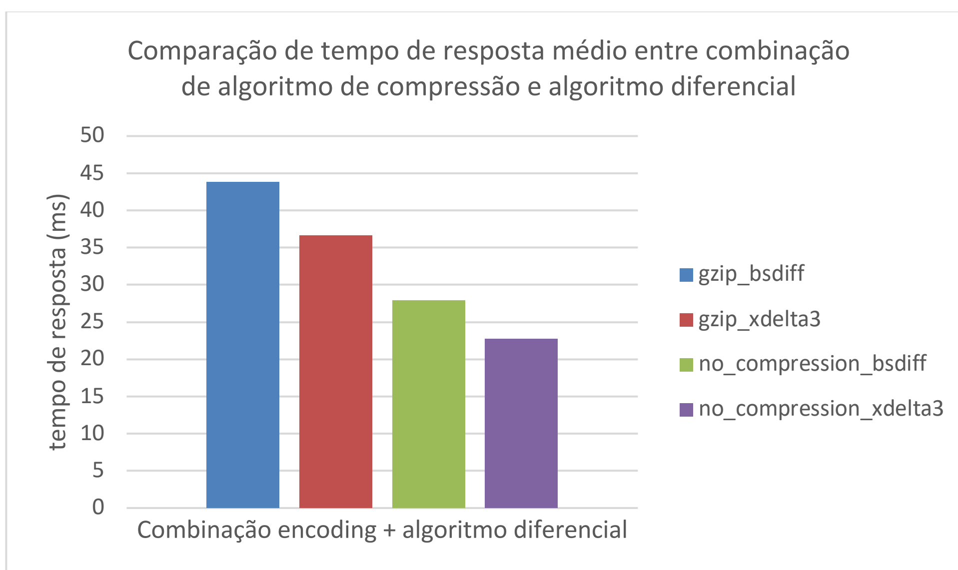


Figura 23 – Comparação de tempo de resposta médio entre combinação de algoritmo de compressão e algoritmo diferencial

Os *patches* diferenciais apresentam tamanhos entre os 240 e 280 bytes, como é apresentado pela Figura 24. Em termos relativos, vê-se representada uma poupança na ordem dos 90%, face ao envio de *blobs* completos (aprox. 2.5kB). Entre algoritmos, *no_compression_bsdiff* produz, em média, patches ligeiramente menores (241 bytes) do que *no_compression_xdelta3* (257 bytes). A compressão com *gzip* não reduz o tamanho médio neste regime, pelo contrário, observa-se um ligeiro aumento (entre 266 -280 bytes), consequência do *overhead* do formato e da baixa entropia residual de objetos muito pequenos.

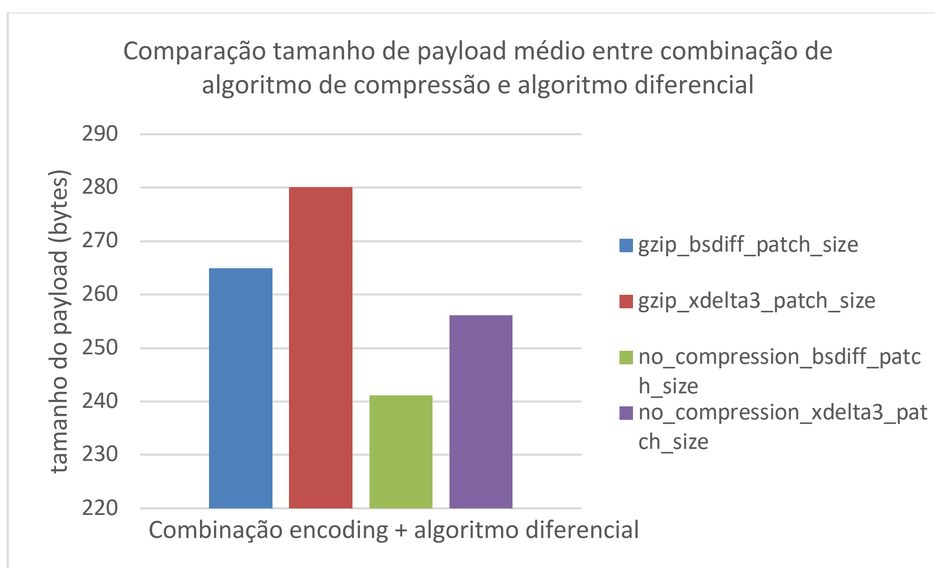


Figura 24 – Comparação de tamanho de *payload* médio entre combinação de algoritmo de compressão e algoritmo diferencial

Pode desde já concluir-se que, quando existem *hot zones*, o par *xdelta3* sem compressão tende a otimizar a latência, enquanto *bsdiff* sem compressão tende a minimizar o *payload*. Em ambos os casos, os ganhos absolutos face à *baseline* são expressivos.

6.2.2 Impacto da probabilidade de acerto na *cache* (p)

O critério de probabilidade de acerto na *cache* foi introduzido para simular acessos à cache de forma mais natural, uma vez que as simulações dependeram da aleatoriedade dos pedidos feitos. Assim, um valor elevado de p emula um cenário urbano densamente populoso, onde existem muitos pedidos iguais, pertencentes à *hot zone*.

A Figura 25 apresenta a variação do tempo médio de resposta quando se aumenta a probabilidade, p, de um pedido pertencer à *hot zone* (p). A probabilidade variou entre 20 e 50%. A tendência apresentada mostra-se constante: quanto maior é p, menor é a latência, traduzindo o benefício de encontrar deltas pré computados e de evitar trabalho de geração em tempo real.

- *no_compression_xdelta3*: redução aproximada de 24 ms para 21–22 ms (≈10–12%).
- *no_compression_bsdiff*: de 27–28 ms para 24–25 ms (9–12%).
- *gzip_xdelta3*: de 39 ms para 36–37 ms (6–8%).
- *gzip_bsdiff*: de 45 ms para 41–42 ms (7–9%).

A magnitude do ganho é maior nas variantes sem compressão, refletindo que, quando o custo de CPU está menos carregado (sem *gzip*), a melhoria introduzida pela cache destaca-se com mais clareza. Em termos práticos, elevar p (por exemplo, através de políticas de seleção de conteúdos com maior taxa de acesso) é uma alavanca eficaz para baixar a latência, sem impactar o consumo de rede.

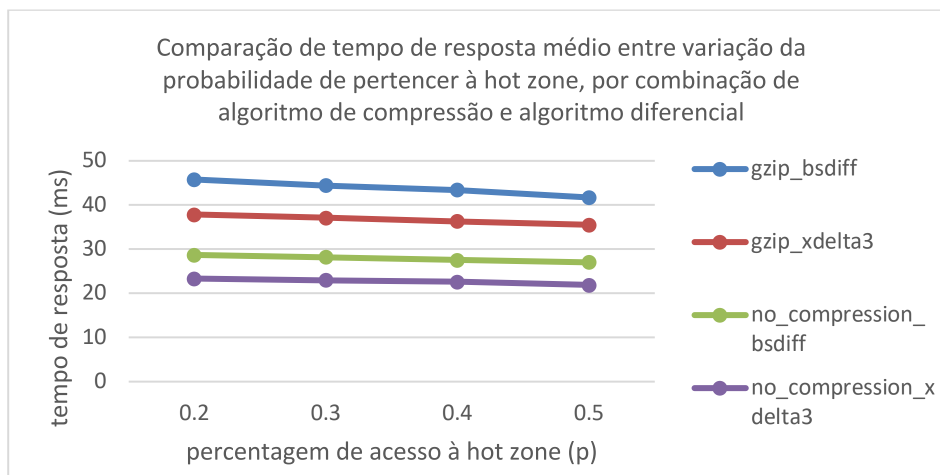


Figura 25 – Comparação de tempo de resposta médio entre variação da probabilidade de pertencer à hot zone, por combinação de algoritmo de compressão e algoritmo diferencial

A Figura 26 mostra que o tamanho médio do *patch* se mantém praticamente invariável com p. Isto decorre do facto de p governar o número de acertos em *cache* e não a magnitude das diferenças entre versões. Assim, os ganhos por aumento de p incidem primordialmente sobre tempo de processamento e não sobre o volume transferido.

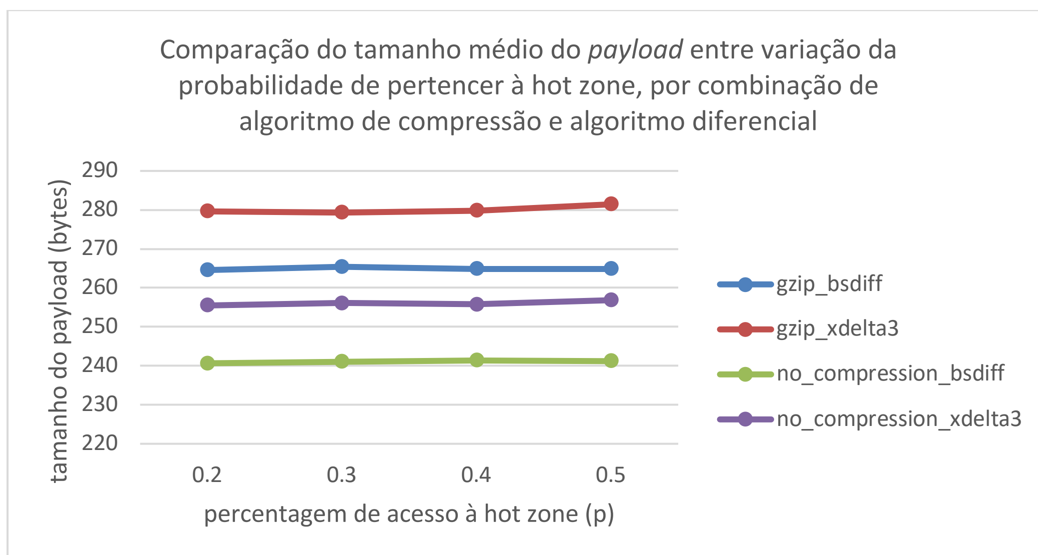


Figura 26 - Comparação do tamanho médio do *payload* entre variação da probabilidade de pertencer à hot zone, por combinação de algoritmo de compressão e algoritmo diferencial

6.2.3 Impacto da dimensão da *hot zone*

A Figura 27 analisa o tempo médio de resposta quando a dimensão da *hot zone* cresce de 20 para 100 e 500. Observa-se uma tendência de degradação com k, típica de *caches* com *footprint* alargado:

- *no_compression_xdelta3*: aumento de 22–23 ms (k=20) para 25–26 ms (k=500), +15–18%.
- *no_compression_bsdiff*: de 26–27 ms para 29–30 ms, ≈10–15%.
- *gzip_xdelta3*: de 33–35 ms para 39–40 ms, +15–20%.
- *gzip_bsdiff*: de 37–38 ms para 44–45 ms, +18–25%.

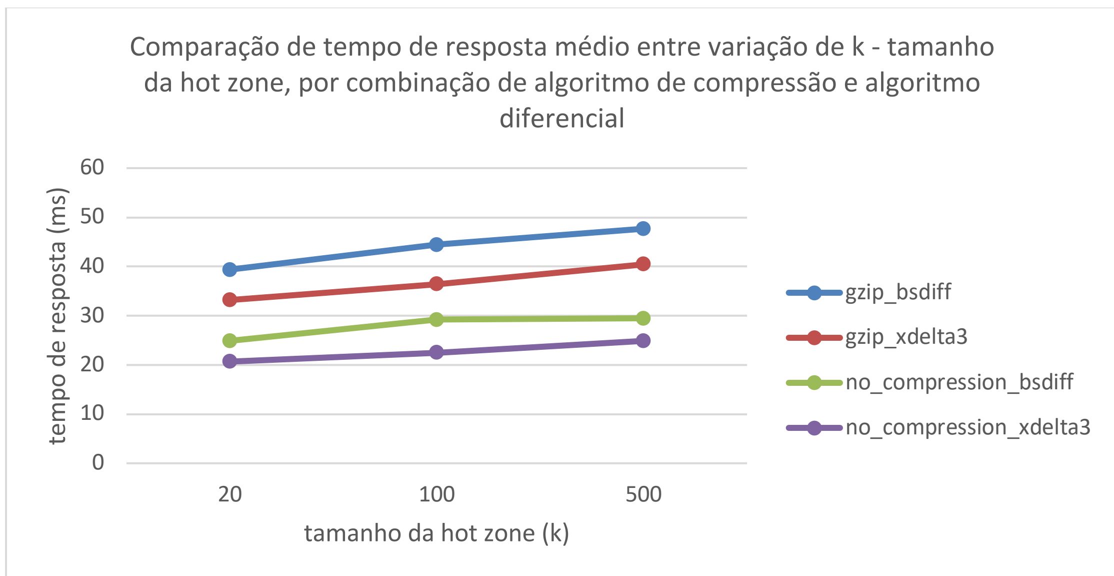


Figura 27 - Comparação de tempo de resposta médio entre variação de k - tamanho da hot zone, por combinação de algoritmo de compressão e algoritmo diferencial

A explicação provável é multifatorial: maior k implica mais entradas, maior pressão de memória, tempos de *lookup* superiores e maior custo de manutenção (carregamento/expiração). Como p e k operam em forças opostas (p acelera por aumentar hits, enquanto k pode atrasar por aumentar *overhead*), é necessário dimensionar k de forma conservadora.

A Figura 28 demonstra, novamente, que o *payload* médio mantém-se estável quando k varia: as curvas oscilam poucos bytes em torno de cada combinação. Ou seja, k impacta latência, não volume transferido. A decisão sobre k deve, portanto, otimizar tempo e capacidade do *backend* e não a largura de banda.

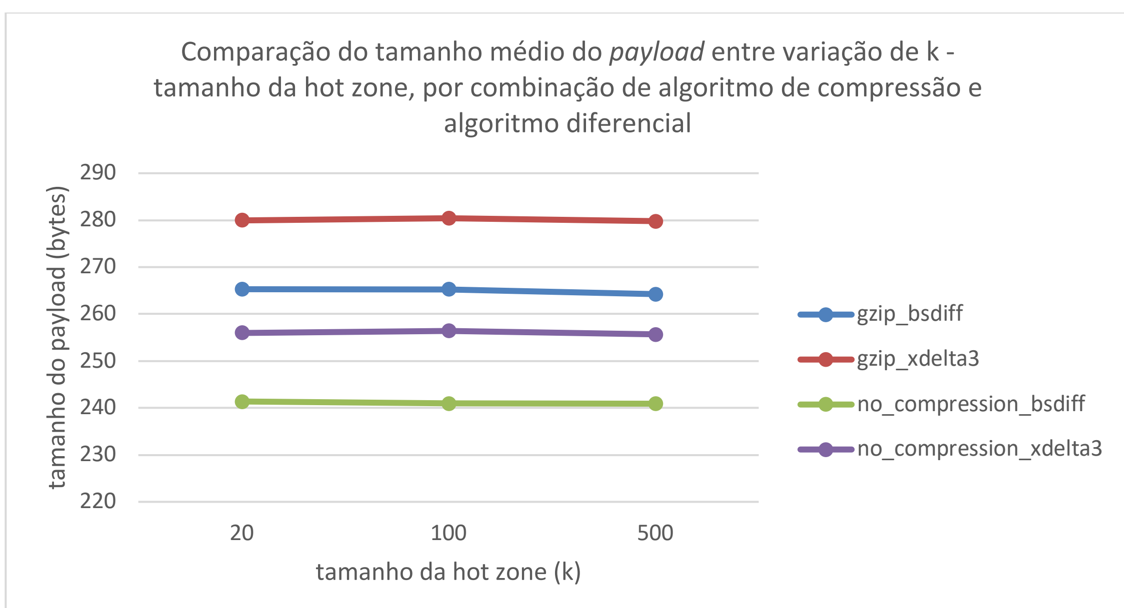


Figura 28 – Comparação do tamanho médio do *payload* entre variação de k - tamanho da hot zone, por combinação de algoritmo de compressão e algoritmo diferencial

6.3 Resultados sem *cache* no *backend*

A ausência de *cache* no *backend* tem impacto direto na eficácia do sistema, uma vez que cada pedido obriga à geração de um *patch* em tempo real. Este tipo de processo inclui a leitura do *blob* anterior e do *blob* atualizado no armazenamento, a execução do algoritmo diferencial (*bsdiff* ou *xdelta3*) e a eventual compressão do resultado. Ao contrário do cenário com *hot zones*, não existe reutilização de artefactos previamente calculados, o que aumenta de forma significativa o tempo médio de resposta.

Como apresentado pela Figura 29, o tempo médio de resposta varia entre 50-90ms. Estes valores estão única e exclusivamente relacionados com o facto do *backend* não ter *cache* e ter de fazer sempre a criação do *patch*. Este resultado serve de base para comparar a utilização de *cache* na solução proposta.

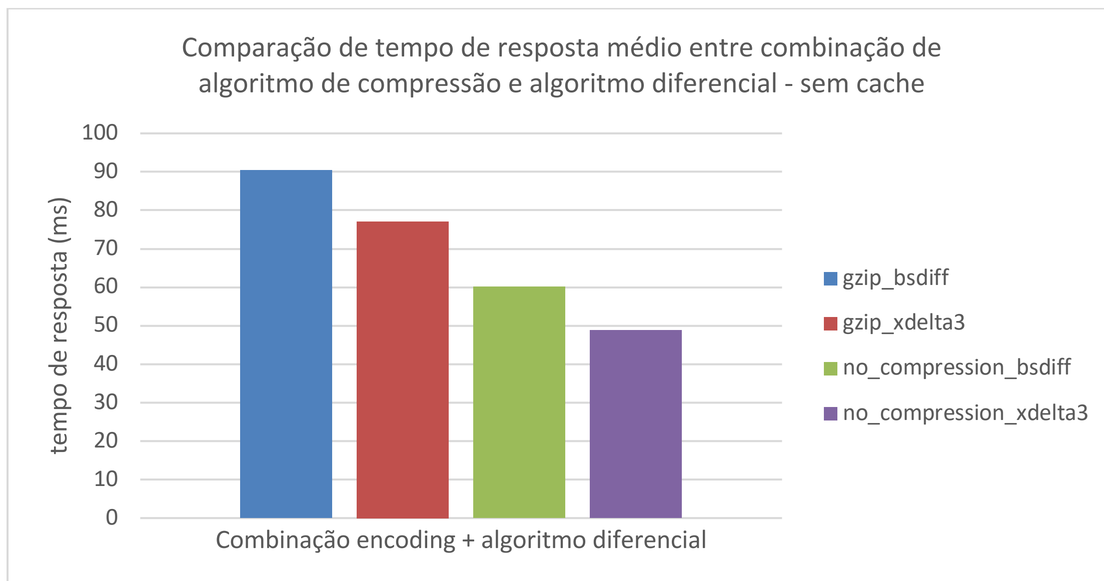


Figura 29 - Comparação de tempo de resposta médio entre combinação de algoritmo de compressão e algoritmo diferencial sem cache

Analisando o tamanho do *payload* e de acordo com a Figura 30, o tamanho médio do *payload* parece apenas variar de acordo com a combinação de algoritmo de compressão e algoritmo diferencial. Isto deve-se ao facto que a ausência de *cache* não alterar o artefacto final transmitido, alterando apenas o tempo necessário para o produzir. Assim, a largura de banda consumida não sofre alterações

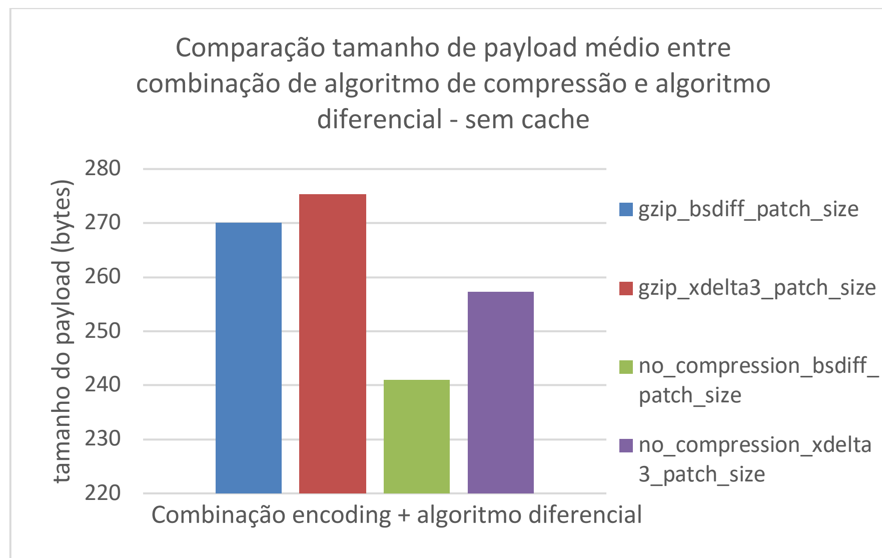


Figura 30 - Comparação tamanho de payload médio entre combinação de algoritmo de compressão e algoritmo diferencial sem cache

6.3.1 Comparação com a solução com *cache* no *backend*

Os resultados obtidos em 6.2 mostram que o tempo de resposta praticamente duplica quando o *backend* não dispõe de *cache*. Nos cenários apresentados, o tempo médio situa-se entre os 90ms para o par *gzip_bsdiff*, 77ms para o par *gzip_xdelta3*, 60ms para o par *no_compression_bsdiff* e 49ms para o par *no_compression_xdelta3*. Comparando com os valores com cache apresentados em 6.2, conclui-se que a não existência de cache aumenta o tempo de resposta em 100%, independentemente do algoritmo diferencial ou do número de iterações. Este resultado é consistente, e reforça a relevância da *cache* na mitigação do custo computacional associado à geração de *patches*.

Assim, a *cache* atua como um fator crítico de otimização, reduzindo o tempo médio de resposta em cerca de 50%. A ausência de *cache* não compromete o tamanho do *payload*, mas compromete a escalabilidade e a experiência do utilizador. Em cenários produtivos, com múltiplos veículos a solicitar *patches* em simultâneo, a inexistência de *cache* poderá constituir um *bottleneck* preocupante, esgotando rapidamente os recursos computacionais do *backend*. Estes resultados suportam a conclusão de que a utilização de *cache* é indispensável para garantir eficiência, principalmente em arquiteturas que visam suportar atualizações OTA em larga escala.

6.4 Comparação entre algoritmos

A avaliação comparativa entre *bsdiff* e *xdelta3* (Figura 23, Figura 25 e Figura 27) permite consolidar duas conclusões robustas:

1. Quanto à latência, *xdelta3* apresenta, de forma consistente, menor tempo médio do que *bsdiff*. Sem compressão, a diferença típica situa-se perto de 15-25% a favor de *xdelta3*. Com compressão (*gzip*), a vantagem mantém-se (10-20%), ainda que mais atenuada pelo custo comum de compressão;
2. Quanto ao *payload*, *bsdiff* gera *patches* ligeiramente menores do que *xdelta3* (6-7% de diferença média na Figura 24). Em valores absolutos, esta diferença traduz-se em 15-20 bytes por *patch*, o que é irrisório, quando comparado com a poupança total face ao *blob* integral.

Para o caso de estudo em uso, onde os JSONs médios rondam os 2.5kB, atualizações frequentes e necessidade de resposta rápida, a latência é um objetivo primário. Assim, *xdelta3* emerge como a escolha mais eficaz na maioria dos cenários. A preferência por *bsdiff* justifica-se quando cada *byte* é essencial e quando uma melhoria de 15 bytes compensa a penalização de tempo observada.

6.4.1 Estabilidade face a p e k

A diferença entre algoritmos mantém-se estável nas variações de p (Figura 25) e k (Figura 27): *xdelta3* mantém a liderança em tempo, *bsdiff* mantém a liderança marginal em *bytes*. Isto indica que a escolha do algoritmo é ortogonal ao dimensionamento da *cache*. Pode-se, portanto, ajustar p e k por critérios de operação, sem inverter a classificação entre algoritmos.

6.5 Impacto da compressão

A compressão desempenha um papel relevante na tentativa de reduzir a dimensão do *payload*, mas os resultados obtidos neste estudo permitem concluir que a sua eficácia está dependente do tamanho e da natureza do artefacto transmitido. No caso de *patches* diferenciais, que apresentam dimensões médias na ordem das centenas de bytes (240–280 bytes), a compressão revelou-se ineficiente.

O custo de compressão/descompressão acrescenta 10-20ms face às variantes sem compressão (Figura 23, Figura 25 e Figura 27), traduzindo-se em 50-70% de aumento do tempo médio. Também se constatou que não reduziu o volume de dados, pois foi observado um ligeiro aumento do *payload* (Figura 24), explicado pelo *overhead* do cabeçalho *gzip*, *checksum* e baixa relação de compressão em objetos muito pequenos com alta estrutura.

A análise evidencia, portanto, que a compressão deve ser aplicada de forma seletiva. Para conteúdos pequenos, como *patches* diferenciais, a sua utilização é contraproducente. Já para *blobs* completos ou artefactos com dimensões superiores a 1 kB, *gzip* pode justificar-se,

sobretudo em contextos de rede mais limitada, onde a largura de banda se apresenta como um *bottleneck*.

6.6 Discussão dos resultados

6.6.1 Principais constatações

Na análise dos resultados, constataram-se os seguintes factos:

- *Deltas* são estruturalmente superiores a *blobs* completos, devido à redução na ordem dos 90% no tamanho dos ficheiros e de 50–75% de redução de latência. A adoção de algoritmos diferenciais é determinante para a eficiência e escalabilidade do sistema.
- *xdelta3* otimiza tempo, enquanto *bsdiff* otimiza bytes. A diferença de tempo é substancial; a diferença de tamanho é pequena. No perfil alvo (*edge* com limitação energética moderada e necessidade de reatividade), *xdelta3* sem compressão é a opção por omissão.
- *Hot zones*: p alto e k moderado. Elevar o valor de p melhora latência sem custo em *bytes*; aumentar k tende a penalizar latência por *overhead* de gestão. Uma configuração k=100 surge como ponto de compromisso saudável.
- Compressão seletiva. Em *patches* com tamanho inferior a 1kb, *gzip* é contraproducente. Deve ser reservado a conteúdos maiores. A adoção de limiares e de compressão orientada ao transporte é preferível.

6.6.2 Implicações operacionais para a arquitetura

Quanto à política de envio, a preferência deve ser por *xdelta3* sem compressão para atualizações incrementais. Apenas se deve utilizar o *blob* completo quando o veículo assim necessite e apenas se deve comprimir o *payload* se o corpo do mesmo exceder um limiar definido, que pode rondar 1kB ou 2kB.

Quanto à gestão de *hot zones*, deve elevar-se o valor de p por priorização de conteúdos recentemente acedidos e com maior taxa de atualização, como por exemplo conteúdo representativo e um mapa de uma zona densamente populada. Já o valor de k deve ser fixado por serviço ou tipo de conteúdo, onde o impacto na latência deve ser monitorizado. De acordo com os dados recolhidos, o valor de k=100 poderá ser um bom início, que deverá ser aumentado com peso e medida.

Quanto à observabilidade, deve registar-se o tempo de geração do *delta*, tempo de *lookup* na *cache*, tempo de transferência, tempo de aplicação no veículo, tamanho do *patch* e taxa de acerto na *cache*.

6.6.3 Ameaças à validade e generalização

Algumas das ameaças à validade dos resultados obtidos foram:

- O ambiente controlado que foi utilizado, onde a ligação entre o Raspberry Pi e o *backend* era estável poderá não representar a realidade dos veículos conectados, que normalmente utilizam redes celulares, onde a perda de dados é constante.
- O tamanho dos *blobs* utilizado poderá induzir resultados injustos para o algoritmo de compressão *gzip*, uma vez que para objetos substancialmente maiores, é plausível que este volte a ser vantajoso.
- A execução sequencial evitou interferências de cache concorrente, mas não mediu efeitos de acesso concorrente a recursos em cenários com vários veículos ao mesmo tempo. Futuros testes deveriam incluir concorrência controlada.

6.6.4 Perspetivas de evolução

Como o processo científico pressupõe a evolução da solução produzida, é necessário apresentar perspetivas que visem a melhoria do processo futuro, para melhor otimizar a solução dita como final.

Assim, alguns pontos de evolução seriam:

- Simular a degradação da rede (latência, *perda*, *throttling*), de forma a quantificar ganhos em ambientes adversos.
- Pré calcular, de forma assíncrona, os deltas e estudar o orçamento de CPU/memória no *backend*.
- Política adaptativa de compressão e *fallback* (*delta* vs *blob*), baseada em telemetria em tempo real, ou telemetria passada (por exemplo, comportamento habitual às segundas-feiras ou em hora de ponta).

6.7 Resultados sob perspetiva de escalabilidade

Embora os cenários avaliados tenham sido executados com um único cliente (Raspberry Pi) de forma sequencial, é relevante discutir os resultados sob a perspetiva de escalabilidade, considerando uma frota de, pelo menos, 10 000 veículos conectados em simultâneo, podendo este valor escalar consideravelmente. Este exercício, ainda que hipotético, permite projetar a viabilidade da solução em contextos mais próximos da realidade de sistemas automotivos modernos.

Se cada veículo realizasse pedidos ao *backend* sem qualquer mecanismo de cache, a infraestrutura teria de gerar, em tempo real, 10 000 *patches* diferenciais para *blobs* distintos ou coincidentes. Como demonstrado nos resultados da secção 6.3, a ausência de cache praticamente duplica a latência média, o que implicaria tempos de resposta superiores a 90 ms

por pedido. Num cenário concorrente, esse valor tenderia a escalar de forma não linear, esgotando rapidamente os recursos de CPU e memória do *backend*. A Figura 31 apresenta que é possível escalar os *pods* através de métricas de CPU, tanto para mais instâncias ou para menos instâncias.

```
metrics:
  {{- if 80 }}
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
  {{- end }}
behavior:
  scaleUp:
    policies:
      - type: Percent
        value: 100
        periodSeconds: 15
      - type: Pods
        value: 4
        periodSeconds: 15
    selectPolicy: Max
    stabilizationWindowSeconds: 0
  scaleDown:
    policies:
      - type: Percent
        value: 100
        periodSeconds: 15
    selectPolicy: Max
    stabilizationWindowSeconds: 300
```

Figura 31 - Escalabilidade em Kubernetes

A introdução de *cache* baseada em *hot zones* reduz significativamente esta pressão. Assumindo um valor elevado de probabilidade de acerto (p), grande parte dos pedidos concorrentes poderia utilizar os artefactos previamente calculados, diminuindo substancialmente o esforço computacional. No entanto, mesmo com *cache*, a escalabilidade teria de ser assegurada através de mecanismos de orquestração e distribuição de carga.

A utilização de soluções de orquestração, como por exemplo Kubernetes constitui uma abordagem eficiente para este desafio. Através da definição de métricas de *scaling*, seria possível aumentar dinamicamente o número de instâncias do *backend* em resposta ao crescimento da carga. Por exemplo, métricas como a utilização de CPU, memória ou a própria latência de resposta poderiam servir de *trigger* para escalar automaticamente o número de *pods* (instâncias) de um para dezenas ou centenas, de acordo com a necessidade. Em paralelo, o uso de *load balancers* asseguraria a distribuição uniforme dos pedidos entre instâncias, mitigando pontos únicos de falha, como apresentado na Figura 32.

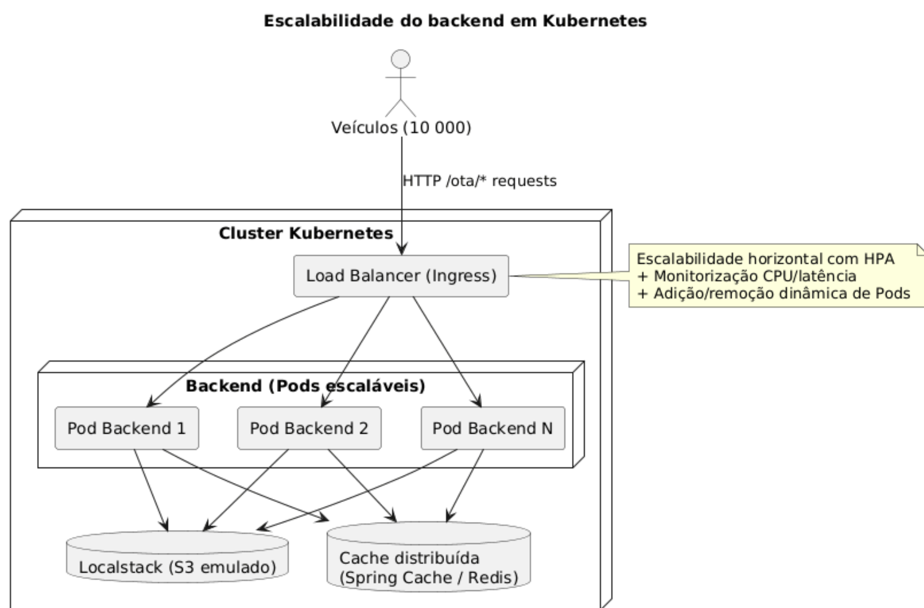


Figura 32 - Arquitetura Kubernetes

Do ponto de vista experimental, a extrapolação dos resultados sugere que, em cenários com pelo menos 10 000 veículos em simultâneo, a utilização de algoritmos diferenciais continua a ser um fator crítico de eficiência, reduzindo drasticamente o volume de dados transferidos. Contudo, sem mecanismos de orquestração, a geração de *patches* em tempo real tornar-se-ia insustentável.

Assim, a combinação de *caching*, compressão seletiva e escalabilidade horizontal via Kubernetes representa o caminho mais promissor para assegurar alta disponibilidade e baixa latência em contextos de larga escala, garantindo alta disponibilidade do serviço, com *downtime* praticamente nulo.

Finalmente, as soluções de *rollback* e de estratégias de *deploy* que Kubernetes fornece podem ser bastante produtivas para o cenário produtivo a ser analisado, uma vez que a visão 0 *downtime* da plataforma visa disponibilidade e qualidade de serviço.

Em síntese, a adoção de Kubernetes como mecanismo de orquestração demonstra-se essencial para garantir a escalabilidade e a resiliência de uma solução OTA em larga escala. Deste modo, Kubernetes não só assegura alta disponibilidade e tolerância a falhas, como também constitui um pilar estratégico para a implementação de arquiteturas OTA fiáveis e sustentáveis em cenários reais de mobilidade conectada.

7 Conclusões

Este trabalho teve como principal objetivo estudar e otimizar a comunicação entre veículos conectados e o *backend*, combinando protocolos de transporte, algoritmos diferenciais (*delta patches*), compressão e estratégias de *cache* baseadas no conceito de *hot zones*. Foi desenvolvido um protótipo *end-to-end*: *backend* em Java 21 com Spring Boot 3 e Localstack (emulação de S3) e cliente num raspberry Pi 5 que, por parâmetro, solicita versão, *blob* completo ou *delta*, com ou sem compressão. AS métricas observadas foram o tempo *end-to-end* do pedido (do envio até à reconstituição do ficheiro no veículo) e o tamanho do *payload*, registadas em CSV para análise posterior.

7.1 Conclusões

7.1.1 Eficácia das *hot zones*

A estratégia de *cache* por *hot zones* demonstrou impacto significativo na redução do tempo de resposta e do volume de dados transferidos, quando comparada com cenários sem pré-carregamento de conteúdos pertinentes.

A parametrização por k (dimensão da zona) e p (probabilidade de acesso dentro da zona) revelou a esperada relação entre localidade e benefício: quanto maior a localidade efetiva (p elevado) e quanto mais bem dimensionada a janela (k suficiente para cobrir os itens com maior frequência), maior a taxa de acertos e menor a necessidade de transferências integrais.

Um k excessivo dilui o foco da *hot zone* e pode degradar a eficácia. No cenário de um k demasiado pequeno existe a perda de oportunidades de acerto. Em síntese, o ganho é maximizado quando k e p refletem o padrão real de acessos.

7.1.2 Algoritmos diferenciais (*bsdiff* vs *xdelta3*)

Verificou-se o *trade-off* clássico entre qualidade do *patch* e custo computacional. O *bsdiff* tende a produzir *patches* ligeiramente mais compactos, à custa de maior tempo e consumo de memória, tanto na geração como na aplicação. O *xdelta3* apresentou um compromisso mais favorável em latência *end-to-end*, sobretudo para *blobs* de pequeno a médio porte, sendo por isso uma opção pragmática para cenários online.

Em conteúdos de reduzida dimensão e baixa similaridade, o benefício de enviar *patch* face a conteúdo integral diminui. É desejável que exista uma política de decisão dinâmica (*patch* vs *full*), suportada por limiares.

7.1.3 Compressão (*gzip*) em *blobs* e *patches*

A compressão reduziu sistematicamente o *payload*, mas o benefício em latência foi dependente do tamanho do artefacto e da similaridade. Em *patches* mais pequenos, o ganho com *gzip* pode não compensar o custo de compressão/descompressão no caminho críitico. Já em conteúdos maiores, ou quando a ligação for mais limitada, a compressão torna-se vantajosa. Pode ser útil ativar a compressão apenas acima de um limiar de tamanho e preferencialmente após a geração do *patch*.

7.1.4 Estratégia de execução e validade dos resultados

A execução sequencial dos ensaios foi adequada para evitar interferências na *cache* (efeitos de poluição entre cenários) e para assegurar reprodutibilidade. As medições *end-to-end*, realizadas no cliente, refletem a perceção que interessa ao veículo: tempo até à disponibilidade do ficheiro reconstruído

7.1.5 Implicações práticas

Da combinação de *cache* por *hot zones* com *diffs* e compressão seletiva, resulta uma abordagem de engenharia com boa relação de esforço e benefício:

- Guardar em *cache*, de forma proativa, conteúdos com maior probabilidade de acesso, utilizando para isso métricas recolhidas quanto à quantidade de acessos àquele conteúdo;
- Preferir *diffs* quando a similaridade esperada é elevada e o *payload* comprido permanece inferior a um limiar face ao *blob* completo;
- Aplicar compressão condicional com base no tamanho final do artefacto.

7.1.6 Estado dos protocolos

Os ensaios reportados focaram a utilização de HTTP. A integração e avaliação sistemática de MQTT e QUIC, previstos no desenho, permanecem como etapa posterior para completar a comparação de protocolos na mesma *pipeline* de processamento.

7.1.7 Resumo

Os resultados obtidos permitem afirmar que os objetivos definidos no capítulo 1.3 e 3.4.1, onde o problema foi apresentado e algumas perguntas de pesquisa foram levantadas. Foi realizada a comparação de combinações de protocolos e algoritmia, confirmando as vantagens e limitações em cenários OTA. Os algoritmos diferenciais *bsdiff* e *xdelta3* foram avaliados em detalhe, verificando-se a redução significativa do *payload* e latência, alinhados com o objetivo principal de otimização. A compressão seletiva e a estratégia de *hot zones* demonstraram ganhos claros

de eficiência e escalabilidade, respondendo ao objetivo de desenhar uma solução robusta o suficiente para suprir as necessidades de um sistema produtivo.

Finalmente, as métricas recolhidas validam a eficácia da abordagem proposta, reforçando a contribuição desta dissertação para uma solução otimizada de entrega OTA em veículos conectados.

7.2 Limitações técnicas

Durante o desenvolvimento desde documento, foram encontradas algumas limitações técnicas, cuja exposição é relevante. Assim:

1. Os testes decorreram numa rede LAN estável, sem injeção de latência, *jitter* ou perdas. Assim, os benefícios relativos de compressão e diferencias podem estar subestimados face a redes reais (rede celular, Wi-Fi congestionado).
2. Os *blobs* têm dimensão média reduzida (na ordem dos kilobytes) e natureza textual (JSON). Envios OTA reais podem envolver binários de dezenas a centenas de megabytes. A generalização para artefactos maiores requer análise e confirmação com análise de dados.
3. Mediu-se tempo *end-to-end* e *payload*. Não foram medidos percentis (p95/p99), consumo energético no dispositivo, utilização de CPU/RAM, nem tempos desagregados (geração de *patch* no servidor vs aplicação no cliente). Este tipo de métricas permitiria validar estes cenários em *head units* mas restritivas.
4. A comparação direta entre HTTP, MQTT e QUIC não foi completada, por limitações temporais e decisões técnicas da empresa onde este documento foi produzido. QUIC/HTTP-3, em particular, poderá alterar o balanço latênca vs fiabilidade em cenários com perdas de informação.

7.3 Sugestões de trabalhos futuros

Para trabalhos futuros relativos às informações recolhidas no desenvolvimento desta tese, sugere-se:

1. Conclusão e expansão do plano experimental:
    a. Introduzir emulação de rede (com recurso a *frameworks* como tc/netem) com perfis de latência, *jitter* e perda realísticas;
    b. Medir percentis (p50/p95/p99), CPU, memória e energia no dispositivo
2. Protocolos de comunicação:
    a. Implementar e avaliar MQTT (QoS 0/1/2, retenção, tópicos por *head units*) na mesma *pipeline*;
    b. Avaliar a utilização de QUIC/TTP-3, em comparação com HTTP/2.
3. Políticas adaptativas de envio

a. Desenhar um processo de decisão que, por pedido, selecione a utilização de *full blobs* vs *delta blobs* e compressão com base em previsões de similaridade, tamanho esperado e condições de rede;
b. Pré-computar e manter em *cache* pares de *diffs* mais prováveis nas *hot-zones*.

4. Algoritmos de *diffs* e compressão
    a. Avaliar *rdiff* e *zstd*, dependendo do conteúdo (tamanho e formato);
    b. Medir impacto de paralelização na geração e aplicação de *patches*.
5. Robustez e segurança
    a. Assinatura e verificação de *patches*/*blobs*, versionamento com ETags, *rollbacks* seguros e tolerância à falha;
    b. *Load tests* na arquitetura, de forma a validar casos de falha de rede a meio de transferências.

Com a utilização do conceito de *hot zones*, é possível que este trabalho seja expandido, para ser representativo dos comportamentos habituais dos utilizadores dos veículos conectados. Assim, e numa primeira instância, através dos dados de *download* dos veículos, pode analisar-se quais os *blobs* com mais acessos para que estes possam ser pré-carregados para a *cache* quando uma nova versão do mapa é disponibilizada. Assim, as zonas que tipicamente apresentam maior carga, reduzirão o impacto no *backend*, consequentemente reduzindo a latência.

Numa segunda fase dessa implementação, poderia integrar-se um modelo de Inteligência Artificial, com mecanismos de *machine learning*, que analisa os comportamentos dos utilizadores e, de forma dinâmica, aumenta ou diminui os recursos do *backend*, bem como prepara a *cache* para possíveis comportamentos (eg: após analisar que todas as sextas-feiras há um grande fluxo de veículos entre Porto e Lisboa, prepara essa zona do mapa já em mapa com possíveis *patches* diferenciais).

Em síntese, os resultados obtidos validam a hipótese de que a combinação de *cache*, orientada por *hot zones*, algoritmos diferenciais e compressão seletiva constitui uma solução eficaz para reduzir latência e tráfego na entrega de conteúdos a veículos conectados. A consolidação desta abordagem requer, contudo, a extensão dos ensaios a redes adversas, cargas concorrentes, conteúdos de maior dimensão e a incorporação sistemática de outros protocolos de comunicação, culminando numa política adaptativa de envio e numa operação robusta em escala.

8 Referências

[1] E. Uhlemann, “Introducing Connected Vehicles,” *IEEE Vehicular Technology Magazine,* vol. 10, no. 1, pp. 23-31, 2015.

[2] S. Halder, A. Ghosal and M. Conti, “Secure OTA Software Updates in Connected Vehicles: A Survey,” *Computer Networks,* 2020.

[3] N. Yakusheva, “State of the art and trends of Vehicle Communication: Overview,” in *27th Telecommunications forum TELFOR 2019*, Belgrado, 2019.

[4] E. Ndego, “Secure Over-the-Air Software Updates for Autonomous Vehicle Operating Systems,” *Distributed Learning and Broad Applications in Scientific Research ,* vol. 10, pp. 206-230, 2024.

[5] F. Yang, “Revisiting WiFi offloading in the wild for V2I applications,” vol. 202, 2022.

[6] M. J. Page, “PRISMA 2020 explanation and elaboration: updated guidance and exemplars for reporting systematic reviews,” 2021.

[7] IEEE, “IEEE Xplore,” IEEE, [Online]. Available: https://ieeexplore.ieee.org.

[8] Google, “Google Scholar,” Google, [Online]. Available: https://scholar.google.com/.

[9] ACM, “ACM Digital Library,” Association for Computing Machinery, [Online]. Available: https://dl.acm.org/.

[10] C. C. Silva, “Agile Methods Adoption on Software Development: A Pilot Review,” in *2014 Agile Conference* , 2014, Orlando.

[11] I. G. A. Premananda, “Design Science Research Methodology and Its Application to Developing a New Timetabling Algorithm,” in *IEEE International Conference on Computational Intelligence and Cybernetics (CyberneticsCom)* , Malang, 2022.

[12] G. Motors, “OnStar & Connected Services,” [Online]. Available: https://www.gm.com/onstar. [Accessed 9 12 2024].

[13] AutoConnectedCar, “Definition of Connected Car – What is the connected car?,” 2014. [Online]. Available: https://www.autoconnectedcar.com/definition-of-connected-car-what-is-the-connected-car-defined/. [Accessed 9 12 2024].

[14] M. Elhoseny and A. E. Hassanien, Emerging Technologies for Connected Internet of Vehicles and Intelligent Transportation System Networks: Emerging Technologies for Connected and Smart Vehicles, Springer International Publishing, 2020.

[15] S. Husain, "An Overview of Standardization efforts for enabling Vehicular-to-Everything Services," *IEEE Journal,* 2017.

[16] B. Kim and S. Park, "ECU Software Updating Scenario Using OTA Technology through Mobile Communication Network," in *2018 IEEE 3rd International Conference on Communication and Information Systems (ICCIS)*, Singapore, 2018.

[17] A. Keskar, "Enhancing Software-Defined Vehicles with Service- Oriented Architecture: A Framework for Scalable and Modular Mobility Solutions," *IRE Journals,* vol. 4, no. 5, pp. 90-102, 2020.

[18] A. Mostafa, "QoSHVCP: Hybrid Vehicular Communications Protocol with QoS Prioritization for Safety Applications," ISRN Communications and Networking, 2012.

[19] Imperva, "OSI Model," [Online]. Available: https://www.imperva.com/learn/application-security/osi-model/. [Accessed 17 12 2024].

[20] F. B. Abdesslem, "Measuring Mobile Network Multi-Access for Time-Critical C-ITS Applications," in *2018 Network Traffic Measurement and Analysis Conference (TMA)*, Viena, 2018.

[21] S. R. Pokhrel, "Improving TCP Performance Over WiFi for Internet of Vehicles: A Federated Learning Approach," *IEEE Transactions on Vehicular Technology,* vol. 69, no. 6, pp. 6798-6802, 2020.

[22] G. A. Thomaz, "UBOTA Protocol - UDP Bursts for Over-the-Air Secure Vehicular Software Updates," in *2024 IEEE 13th International Conference on Cloud Networking (CloudNet)*, Rio de Janeiro, 2024.

[23] D. Gourley, B. Totty, M. Sayer, S. Reddy and A. Aggarwal, HTTP: The Definitive Guide, O'Reilly Media, 2002.

[24] D. Candal-Ventureira, "5G Network Slicing as a Service Enabler for the Automotive Sector," *Engineering Reports,* 2024.

[25] Y. Shin and S. Jeon, "MQTree: Secure OTA Protocol Using MQTT and MerkleTree," *MDPI - Sensors,* 2024.

[26] M. Szántó, "Trajectory Planning of Automated Vehicles Using Real-Time Map Updates".

[27] A. Langley, “The QUIC Transport Protocol: Design and Internet-Scale Deployment,” in *SIGCOMM ’17*, Los Angeles, 2017.

[28] A. Matthieu, T. Ludovic and S. Ye-Qiong, “A Performance Evaluation of QUIC in Real-Time Networks,” in *RTNS'24 A*, Porto, 2024.

[29] Medium, “Edgecast - Medium,” Medium, 29 5 2018. [Online]. Available: https://edgecast.medium.com/how-quic-speeds-up-all-web-applications-62964aadb3d1. [Accessed 30 12 2024].

[30] D. Piatkowski, T. Puslecki and K. Walkowiak, “Study of the Impact of Data Compression on the Energy Consumption Required for Data Transmission in a Microcontroller-Based System,” *Sensors,* 2024.

[31] K. Sayood, Lossless Compression Handbook, 2002.

[32] Y. Onuma, Y. Terashima and S. Nakamura, “Compression Method for ECU Software Updates,” in *2017 Tenth International Conference on Mobile Computing and Ubiquitous Network (ICMU)*, Toyama, 2017.

[33] W. Qiao, “An FPGA-Based BWT Accelerator for Bzip2 Data Compression,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, San Diego, 2019.

[34] Z. Li, “BSDIFF Difference Algorithm Based on LZMA2 for In-Vehicle ECUs,” in *7th International Conference on Computing, Control and Industrial Engineering (CCIE 2023)* , Hangzhou, 2023.

[35] E. Westerberg, “Efficient delta based updates for read-only filesystem images,” KTH Stockholm, 2021.

[36] D. Bogdan, R. Bogdan and M. Popa, “Delta ashing of an ECU in the automotive industry,” in *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2016, Timisoara.

[37] G. Ni, “Research on Incremental Updating,” in *International Conference on Communications, Information Management and Network Security (CIMNS 2016)*, Shanghai, 2016.

[38] H. Tan, “Exploring the Potential of Fast Delta Encoding: Marching to a Higher Compression Ratio,” in *IEEE International Conference on Cluster Computing* , Kobe, 2020.

[39] A. Haroun, A. Mostefaoui and F. Dessables, “A Big Data Architecture for Automotive Applications: PSA Group Deployment Experience,” in *International Workshop on Distributed Big Data Management*, Madrid, 2017.

[40] S. Hashima, “Multi-armed Bandit-Aided Near-Optimal Over-The-Air Updates in Multi-Band V2X Systems,” in *ICCCI 2023 5th International Conference on Computer Communication and the Internet (ICCCI)*, Fujisawa, 2023.

[41] Amazon, “S3 Homepage,” AWS Amazon, 2025. [Online]. Available: https://aws.amazon.com/s3/. [Accessed 13 9 2025].

[42] Localstack, “Localstack Homepage,” 2025. [Online]. Available: https://www.localstack.cloud/. [Accessed 13 9 2025].

[43] D. Inc., “Docker Homepage,” Docker Inc, 2025. [Online]. Available: https://www.docker.com/. [Accessed 13 9 2025].

[44] Raspberry Pi Foundation, “Raspberrry Pi,” 2025. [Online]. Available: https://www.raspberrypi.com/. [Accessed 13 9 2025].

[45] A. Ioana and A. Korodi, “OPC UA Publish-Subscribe and VSOME/IP Notify-Subscribe Based Gateway Application in the Context of Car to Infrastructure Communication,” *MDPI - Sensorss,* 2020.

[46] “A Hybrid Vehicle Hardware-in-the-Loop System with Integrated Connectivity for eHorizon Functions Validation,” IEEE Transactions on Vehicular Technology, 2021.

[47] M. Forster, R. Frank and T. Engel, “An Event-Driven Inter-Vehicle Communication Protocol to Attenuate Vehicular Shock Waves,” in *2014 International Conference on Connected Vehicles and Expo (ICCVE)*, Viena, 2014.

[48] A. Rahmatulloh, “Event-Driven Architecture to Improve Performance and Scalability in Microservices-Based Systems,” in *2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS)* , Bandung, 2022.

[49] www.onlinegantt.com, “Online Gantt,” [Online]. Available: https://www.onlinegantt.com/#/gantt. [Accessed 2 1 2025].