FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Future Cities Data and Services API

**João Cardoso**


Mestrado Integrado em Engenharia Informática e Computação

Orientador: Ana Aguiar

Co-orientador: Ricardo Morla

26 de Julho de 2016

# Future Cities Data and Services API

**João Cardoso**

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Luís Teixeira
Arguente: Pedro Brandão
Vogal: Ana Aguiar
26 de Julho de 2016

# Resumo

Atualmente, o número de pessoas que vivem em zonas urbanas tem vindo a crescer, sendo que no ano de 2010, 75% das pessoas na União Europeia viviam em cidades, e é esperado que este número cresça para 85% até 2050. Acrescentado ao desenvolvimento das mesmas, o termo *Smart Cities* — aplicado às cidades que implementam soluções que lhes permitem melhorias qualitativas e quantitativas na produtividade bem como na qualidade de vida das pessoas — e *Internet of Things* — que se refere à conexão de "coisas" à Internet, como por exemplo sensores — têm vindo a ganhar destaque.

Neste sentido, o *Future Cities Project* surgiu com o objetivo de tornar a cidade do Porto numa *smart city* e num laboratório vivo à escala urbana, através da instalação de duas grandes plataformas de sensores, o *SenseMyCity* e o *UrbanSense*. Estas plataformas recolhem milhares de dados geo-indexados através dos vários sensores presentes num *smartphone* e dados atmosféricos e de ruído através de sensores fixos.

Com o surgimento das *Smart Cities*, começaram a aparecer *middlewares* para o desenvolvimento de aplicações e serviços para estas. Com o aumento do número destes, há uma crescente necessidade de comparações entre eles, tanto na forma quantitativa como qualitativa, descrevendo qual o melhor em cada situação e o porquê de escolher um ou outro.

Esta análise quantitativa consiste num *benchmark* de performance e incide sobre quatro cenários distintos: publicação de dados de forma sequencial com o modelo *publish/subscribe*, publicação de vários dados em simultâneo com o mesmo modelo referido anteriormente, *request/response* e disponibilização de um *dataset*. Cada cenário tem métricas de *performance* ajustadas ao mesmo. Nos dois primeiros, são avaliadas métricas como o tempo de publicação, tempo de subscrição, total de *bytes* utilizados na publicação, *goodput*, entre outros. No terceiro, são avaliadas métricas como o tempo que demora a obter a resposta (essencial em cenários como este, onde o tempo de resposta é crucial), total de *bytes* utilizados, entre outros. Por fim, no quarto e último, não existe muito a avaliar em termos quantitativos, está mais relacionado com aspetos qualitativos como por exemplo as limitações em termos da engenharia de *software* que cada tecnologia tem para albergar milhares de dados.

A análise qualitativa é focada numa análise de requisitos para *middlewares* neste contexto e numa avaliação das várias soluções em termos de métricas não quantificáveis como a usabilidade, suporte para a resolução de *bugs*, documentação das suas *APIs*, viabilidade em termos da engenharia de *software* nos vários cenários, entre outros.

Os *middlewares* visados por estas análises são o *REST* convencional, *ETSI M2M*, *FIWARE* e o por fim o *CitySDK*. É determinada a aplicabilidade de cada nos vários cenários, para posteriormente ser possível avaliá-los em cada um destes.

Em relação às experiências efetuadas para a avaliação quantitativa, foi verificado que o *FIWARE* tem um desempenho melhor nos dois primeiros cenários, tanto em termos da eficiência dos recursos da rede como em termos da velocidade. No terceiro, foi verificado que o *REST* é o mais indicado, visto ser mais eficiente e rápido que os demais.

# Abstract

The number of people living in urban areas has been growing over the years, for example, in 2010, 75% of the European Union population lived in cities and this number is expected to increase to 85% by 2050. Allied to the development of these urban areas, the terms Smart Cities — that refers to cities that implement solutions that allow them significant qualitative and quantitative improvements in the productivity and quality of people's life — and Internet of Things, which is the connection of "things", such as sensors to the Internet, are becoming more important by the minute.

Therefore, the Future Cities Project appeared with the objective of turning the city of Porto into a smart city and a living lab at urban scale. Thus, two sensor platforms, SenseMyCity and UrbanSense, were deployed all over the city. These platforms collect geo-indexed data through the several sensors available in a smart phone and atmospheric data and noise through stationary sensors.

Due to the appearance of these smart cities, more and more middlewares to develop services and applications for these started to appear. Therefore, there is the need to have quantitative and qualitative comparisons between them, describing which one is the best in each situation and why should we choose one or another.

This quantitative analysis is aimed at generating a performance benchmark and focuses on four different scenarios: publishing data in a sequential way using the publish/subscribe model, publishing data at the same time using the same model as before, request/response and the provision of a data set. Each scenario has different performance metrics, tailored for each one. In the first two, the performance metrics observed are the time to publish the data, the subscription time, the amount of bytes used to publish the data, the goodput, among others. In the third one, the response time (essential in scenarios like this one), amount of bytes used, among others are evaluated. At last, in the fourth one there is not much to evaluate in quantitative terms, it is more focused on software engineering limitations of each technology to hold all the data needed.

The qualitative analysis is focused on a requirements analysis for middlewares in this context and on the evaluation of all the middlewares in terms of non quantifiable metrics such as usability, support to fix bugs, API documentation, viability of each technology in each scenario according to the software engineering limitations of each one, among others.

The middlewares used in this analysis are plain REST services, ETSI M2M, FIWARE and CitySDK, and each one will be evaluated in all scenarios if it is applicable.

Regarding the experiences conducted in order to evaluate these middlewares in quantitative terms, it was observed that FIWARE had a better performance in the first two scenarios, as it made more efficient use of network resources and it was quicker. In the third scenario, it was concluded that REST was more efficient and quicker than the other middlewares.

# Agradecimentos

Gostaria de agradecer em especial à Professora Ana Aguiar por todo o excelente acompanhamento ao longo da minha dissertação, tanto durante o primeiro como no segundo semestre. Gostaria também de agradecer ao Professor Ricardo Morla por toda a ajuda e conselhos dados ao longo da realização desta dissertação. Por fim, gostaria de agradecer ao João Rodrigues por me ajudar com algumas questões mais específicas e ao Carlos Pereira pela ajuda a resolver algumas questões pontuais.

João Cardoso

*"Why do we fall? So we can learn to pick ourselves up"*

Alfred in Batman Begins

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# Abreviaturas e Símbolos

| | |
|---|---|
| API | *Application Programming Interface* |
| CPU | *Central Processing Unit* |
| CRUD | *Create, Read, Update and Delete* |
| CSS | *Cascading Style Sheets* |
| CSV | *Comma Separated Values* |
| DSCL | *Device Service Capability Layer* |
| ETSI | *European Telecommunications Standards Institute* |
| FAQ | *Frequently Asked Questions* |
| GPL | *General Public License* |
| GPS | *Global Positioning System* |
| GSCL | *Gateway Service Capability Layer* |
| HTML | *HyperText Markup Language* |
| HTTP | *Hypertext Transfer Protocol* |
| HTTPS | *Hypertext Transfer Protocol Secure* |
| JSON | *JavaScript Object Notation* |
| M2M | *Machine to Machine* |
| NFC | *Near Field Communication* |
| NGSI | *Next Generation Services Interface* |
| NSCL | *Network Service Capability Layer* |
| OWL | *Web Ontology Language* |
| PCAP | *Packet Capture* |
| PEP | *Policy Enforcement Point* |
| POI | *Point of Interest* |
| QR code | *Quick Response code* |
| REST | *Representational State Transfer* |
| RFID | *Radio-Frequency Identification* |
| SDK | *Software Development Kit* |
| SNMP | *Simple Network Management Protocol* |
| SOAP | *Simple Object Access Protocol* |
| SSL | *Secure Sockets Layer* |
| STCP | Sociedade de Transportes Colectivos do Porto |
| TCP | *Transmission Control Protocol* |
| TLS | *Transport Layer Security* |
| UML | *Unified Modeling Language* |
| URI | *Unique Resource Identifier* |
| URL | *Uniform Resource Locator* |
| UTC | *Coordinated Universal Time* |
| WGS | *World Geodetic System* |
| XML | *eXtensible Markup Language* |

# Capítulo 1

# Introdução

*Smart Cities* é um conceito cada vez mais presente devido ao desenvolvimento da tecnologia e das cidades [CDN11], como é possível verificar na figura 1.1. Este termo aplica-se a cidades que implementam soluções inteligentes, melhorando nomeadamente os seus serviços através das tecnologias de informação, permitindo melhorarias qualitativa e quantitativamente na sua produtividade, fazendo-as prosperar. O projeto *Future Cities* vem neste sentido, combinando *Internet of Things* — a conexão de aparelhos/dispositivos ("coisas") à *Internet* — com *Smart Cities*.

Em 2014, o setor da *Internet of Things* estava avaliado em cerca de 665 mil milhões de dólares, e é esperado que cresça para 1.7 biliões de dólares até 2020. [Rag16].

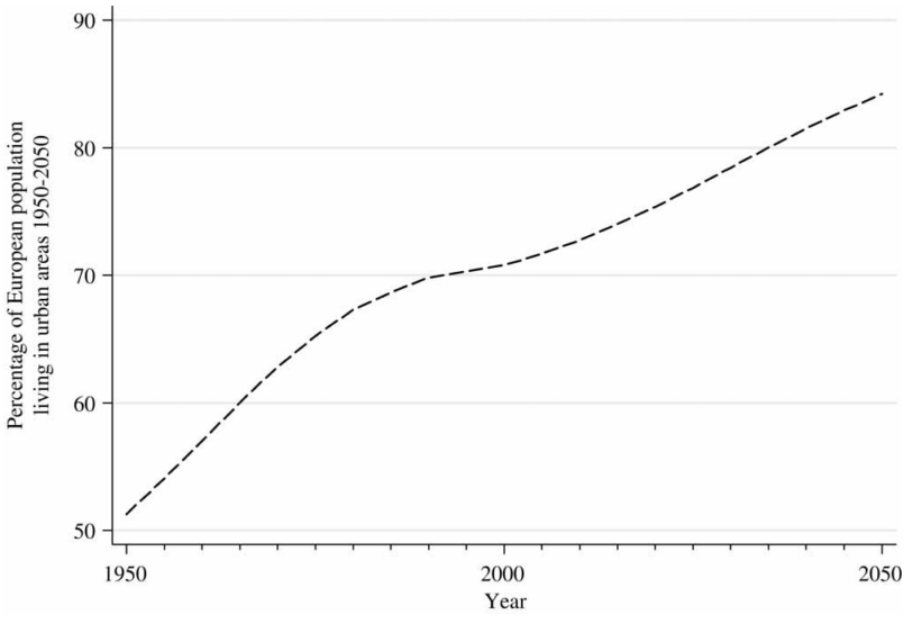



Figura 1.1: Percentagem da população europeia a viver em cidades, de 1950 a 2050[CDN11]

## 1.1 Contexto

Esta dissertação surge após o projecto *Future Cities*[1], que teve como objetivo tornar a cidade do Porto numa *Smart City*. Como tal, foi equipada com duas plataformas de sensores, móveis (*SenseMyCity*) e fixos (*UrbanSense*). A primeira coleciona dados com uma referência geográfica (usando o *GPS* do telemóvel) através dos vários sensores presentes num *smartphone* sobre a área urbana da cidade do Porto. A segunda recolhe dados atmosféricos (qualidade do ar, níveis de poluição), ruído, radiação, entre outros, através de 19 sensores espalhados por toda a cidade do Porto.

Já existem fluxos de dados gerados pelos sensores, utilizados por aplicações de terceiros. De modo a isto ser possível, é necessário uma tecnologia intermediária — um *middleware* [Ber96a] — para que os sensores e serviços sejam interoperáveis, atuando como uma espécie de "cola"entre estes dois. Desta forma, é construído um sistema distribuído de grande escala.

Estes *middlewares* tipicamente comunicam através de *frameworks* de comunicação como o *SOAP*[2] ou o *REST* [Fie00].

## 1.2 Motivação

Poder contribuir com uma análise quantitativa inédita sobre vários *middlewares* que permitem desenvolver aplicações ou serviços neste contexto, como por exemplo disponibilizar os milhares de dados recolhidos pelos sensores publicamente para serem utilizados por terceiros para desenvolver aplicações/serviços com estes. Existem vários cenários num contexto de *smart cities* e *internet of things*, que vão desde uma simples obtenção de informações sobre um dado tópico até fluxos autónomos de disponibilização de dados, ajustados às necessidades de cada caso. Esta análise quantitativa permite que no futuro, quando for necessário decidir sobre um ou outro *middleware*, existam dados concretos sobre qual é o mais adequado em cada circunstância, ao invés de se decidir recorrendo a preferências.

## 1.3 Problema

Visto que os vários sensores recolhem milhares de dados numa grande área, é necessário escolher a forma mais indicada de os disponibilizar. Para os disponibilizar existem vários cenários. Existem também vários *middlewares* para disponibilizar os dados, que são genéricos, isto é, funcionam de várias formas diferentes, não tendo sido desenhados para uma situação em específico. Como tal, é necessário verificar qual é o melhor para cada caso. Têm limitações, sendo necessário descobrir quais, e por fim é também necessário verificar como é o seu desenvolvimento, se existem comunidades de desenvolvimento para estes e como é o seu suporte. Deste modo, esta dissertação procura esclarecer estes pontos.

[1] http://futurecities.up.pt/site/
[2] https://www.w3.org/TR/soap/

Não existem ainda comparações quantitativas, isto é, um *benchmark* de desempenho em termos de aspetos ao nível da aplicação e da rede nos vários cenários entre os vários *middlewares*. Também não existem comparações qualitativas, nomeadamente limitações em termos da engenharia de *software*, se são aplicáveis e viáveis nos vários cenários, como são as suas ferramentas, a sua documentação, se existe suporte, entre outros.

Torna-se então necessário definir quais os cenários mais apropriados neste contexto, estudar a aplicabilidade de cada tecnologia em cada um destes e definir quais as medições a efetuar para a implementação do *benchmark* e como e a que nível as obter.

## 1.4 Objetivo

O objetivo consiste em realizar uma análise quantitativa e qualitativa de vários *middlewares* para disponibilizar um grande volume de dados recolhidos pelos sensores espalhados pela cidade. Em termos da análise quantitativa, é proposta uma metodologia para a avaliação de *middlewares* neste contexto. Adicionalmente, esta tem como objetivo implementar uma *tool box* capaz de capturar métricas relevantes para poder obter conclusões sobre o desempenho destes *middlewares* nos cenários propostos. Para tal, estes *middlewares* foram implementados nos cenários em que são aplicáveis. De referir ainda que estas tecnologias compõem sistemas *black-box*, isto é, não existe nenhuma informação como estes sistemas funcionam internamente, e deste modo estas análises procuram também definir o seu comportamento interno.

Esta análise incide em quatro *middlewares* diferentes: serviços *REST* convencionais, *ETSI M2M*, *FIWARE* e *CitySDK* e em quatro cenários distintos: publicação de dados de forma sequencial com o modelo *publish/subscribe*, publicação de vários dados em simultâneo com o mesmo modelo referido anteriormente, *request/response* e disponibilização de um *dataset*. Estes *middlewares* são *RESTful* e funcionam com base em pedidos *HTTP*, isto é, para realizar operações com estes são efetuados pedidos *HTTP*. O *REST* e o *CitySDK* funcionam num modelo de cliente-servidor, enquanto que os dois restantes funcionam através de *brokers*. Foram escolhidos estes quatro visto serem os que se tinha conhecimento que seriam aplicáveis neste contexto.

## 1.5 Estrutura da Dissertação

Para além da introdução, esta dissertação contém mais 6 capítulos.

No capítulo 2, é feita uma revisão bibliográfica onde são apresentadas em detalhe várias tecnologias que podem ser utilizadas para desenvolver serviços para *smart cities*, bem como trabalhos relacionados, incidindo sobre sistemas que façam *benchmarks* no contexto de *smart cities* e *internet of things*.

No capítulo 3 é descrito o problema em detalhe, a metodologia proposta para o resolver, os vários cenários propostos e por fim o caso de uso utilizado.

No capítulo 4 é apresentada uma avaliação qualitativa entre os vários *middlewares*, no qual é feita uma análise de requisitos, uma análise da aplicabilidade de cada *middleware* em cada

cenário, uma descrição da documentação e suporte disponível, e para terminar é feita uma análise da viabilidade de cada *middleware* em cada cenário.

No capítulo 5 é descrito o processo de desenvolvimento da *tool box* para a avaliação quantitativa dos vários *middlewares*, a sua arquitetura, quais são os seus componentes, como são constituídos e como interagem entre si. Por fim, é descrito o *workflow* da *tool box* desenvolvida.

No capítulo 6 são apresentados resultados relativos à análise quantitativa nos cenários propostos e são tiradas conclusões sobre estes, compreendendo os resultados obtidos e colocando hipóteses para estes.

Por fim, no capítulo 7 é feita uma conclusão sobre o trabalho desenvolvido, bem como possíveis melhorias futuras da *tool box* desenvolvida.

# Capítulo 2

# Revisão Bibliográfica

## 2.1 Modelos de Comunicação

### 2.1.1 Cliente-Servidor

Este modelo [Ber96b] separa as funções entre a entidade que faz os pedidos de serviços — o cliente — e a que lhe fornece esses serviços, tendo na sua posse todos os dados necessários para esse efeito: o servidor.

O cliente não partilha nenhum dos seus recursos, recorrendo ao servidor para obter informações sobre um dado conteúdo ou função. Este também não precisa de saber como o servidor processa o pedido e entrega a resposta. Apenas necessita de conseguir descodificar a resposta, isto é, o seu conteúdo e a sua formatação.

### 2.1.2 Publish/Subscribe

O modelo de *publish-subscribe* [EFGK03] especifica duas operações essenciais:

- *publish*, que permite publicar dados num *software bus* ou *event manager*;
- *subscribe*, que permite a uma entidade subscrever um tipo de evento, e ser notificada de qualquer evento gerado por um *publisher*.

O evento despoletado aquando de uma publicação é propagado de forma assíncrona para todas as entidades que o tenham subscrito. Existem 3 tipos de modelos:

- baseado em tópicos, no qual os *subscribers* subscrevem dados com base em tópicos, ou seja, todas as informações publicadas para esse tópico;
- baseado em conteúdo, onde os *subscribers* expressam o seu interesse em determinado conteúdo, ao invés de subscrever um tópico bem definido, de acordo com uma série de filtros ou restrições, que vão de encontro aos seus interesses;

- baseado em tipos, em que os *subscribers* subscrevem o tipo de evento que desejam, por exemplo os eventos de ações podem ser divididos em ver a sua cotação e em comprar ações, e os *subscribers* subscrevem apenas o evento da cotação das ações.

A grande vantagem deste modelo é a dissociação total entre *publishers* e *subscribers*, como é possível verificar na figura 2.1, em termos de:

- tempo, os intervenientes não necessitam de participar ativamente ao mesmo espaço de tempo, por exemplo, o *publisher* pode publicar enquanto o *subscriber* está *offline*, e quando estiver *online*, irá receber as notificações, não sendo necessário o *publisher* estar *online* nesse momento;

- espaço, os intervenientes não necessitam de se conhecer, os dados são publicados para um servidor ou um *broker* e como tal não têm conhecimento de quem subscreve e vice versa, nem necessitam, visto que os *subscribers* obtêm as atualizações diretamente deste servidor/*broker*;

- sincronização, os *publishers* não ficam bloqueados enquanto publicam dados, e os *subscribers* recebem as notificações de forma assíncrona, através de um *callback*, enquanto realizam qualquer outra operação.

Em muitos sistemas de *publish/subscribe* as publicaçãoes são enviadas para um intermediário entre o *publisher* e o *subscriber*: um *message/event broker*. Este recebe as publicações dos *publishers* e permite que os *subscribers* subscrevam tópicos/conteúdos nesse mesmo. Aquando de um evento gerado por uma publicação, o *broker* encarrega-se de encaminhar as notificações para quem subscreveu.

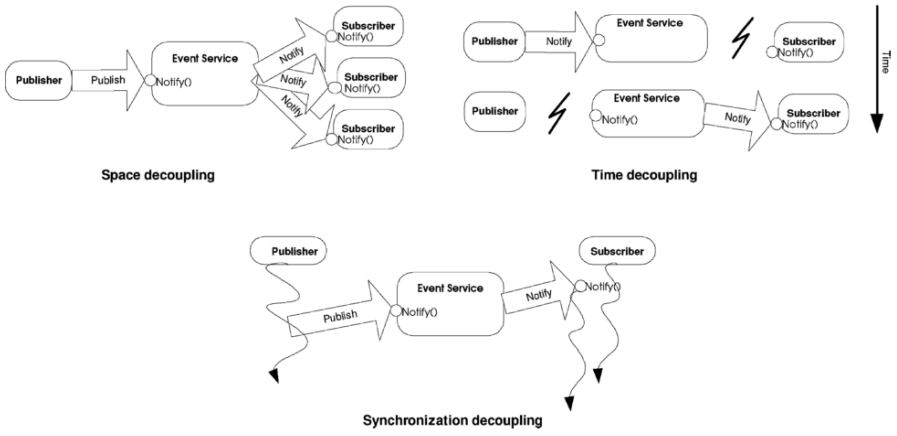



Figura 2.1: Dissociação em terms de tempo, espaço e sincronização [EFGK03]

## 2.2 Trabalhos Relacionados

Com o crescer da importância da *Internet of Things* [Rag16], é necessário que standards para a comparação de tanto *hardware* e *software* direcionados a esta área surjam. Em Agosto de 2015 o *Transaction Processing Performance Council*[1] anunciou a criação de um comité para a criação de um standard para *benchmarks* nesta área. Até à data desta dissertação, este standard para *benchmarks* ainda não se encontrava no sítio *web* do *Transaction Processing Performance Council*.

Neste sentido, foi feita uma revisão bibliográfica de sistemas que façam *benchmarks* na área de *smart cities*, *internet of things* e *middlewares*. No entanto, não foi encontrado nada especificamente direcionado a *middlewares* para este contexto, apenas foram encontrados alguns sistemas remotamente parecidos.

Um dos sistemas de *benchmark* encontrados foi o *ApacheBench* [The16], que permite fazer testes de carga a servidores *HTTP*. É possível selecionar qual o *endpoint* deste a ser testado e enviar um dado número de pedidos concorrentes, sendo que é possível definir qual o número máximo de pedidos a efetuar e qual o número de pedidos concorrentes. Permite também uma série de parâmetros adicionais, como adicionar credenciais de autenticação aos pedidos efetuados, escrever resultados para um ficheiro *CSV* contendo os vários percentils, desde 1% a 100%, entre outros.

Existem mais ferramentas de *benchmark* parecidas com este, com o mesmo objetivo, de testar como um servidor *HTTP* se comporta sob muita carga. Um destes é o *Siege* [Jef16], um sistema que funciona através da linha de comandos em sistemas *Unix* com licença *GNU GLP open-source*. Permite fazer testes de carga durante um tempo predefinido, por exemplo um minuto, com um número de pedidos concorrentes predefinidos também.

Em termos do modelo *publish/subscribe*, foi encontrado um sistema de *benchmark* de sistemas de mensagens com este modelo [SAKB], o *jms2009-PS*. Este *benchmark* é baseado no standard de *benchmark* para servidores *MOM (Message Oriented Middlewares) SPECjms2007* [SKB+09], que utiliza o *JMS (Java Message Service)*. No entanto, o *jms2009-PS* é direcionado exclusivamente a sistemas que implementem o modelo de *publish/subscribe*. Este *benchmark* foi testado em três cenários diferentes, diferindo no número de destinos diferentes das mensagens e dos dos tipos de destinos usados nas comunicações.

O escopo deste *benchmark* é diferente do objetivo desta dissertação, no sentido em que são avaliadas métricas como a utilização do *CPU*, a carga máxima em cada cenário, a latência média para o recebimento das várias mensagens trocadas e o tempo de *CPU* por cada parâmetro BASE, um parâmetro que determina a taxa nas quais as interações são executadas. O objetivo desta em termos da avaliação quantitativa é avaliar o desempenho de *middlewares* que funcionam através de pedidos *web* (sendo desta forma importante métricas ao nível da rede). Adicionalmente, o facto de que estas métricas são mais indicadas ao funcionamento interno do *broker* em si, como a utilização do *CPU*, e visto que não existe acesso interior ao *broker*, a única métrica relevante é a latência média para o recebimento das várias mensagens enviadas.

[1] http://www.tpc.org/

## 2.3 Middlewares

### 2.3.1 Introdução à Service Oriented Architecture

As tecnologias descritas nas próximas subsecções são orientadas a serviços (*SoA (Service Oriented Architecture*) [PL03]. Esta arquitetura define uma forma de construir aplicações de modo a que estas sejam compostas por serviços que têm interfaces simples e bem definidas, não dependendo do contexto ou estado de outros serviços (*loosely coupled*). Estes comunicam entre si através de protocolos de comunicação, tipicamente através da rede. Existem três princípios fundamentais:

- *Service Loose Coupling*, os serviços não devem depender de detalhes de representação ou de implementação específicos;
- *Service Cohesion*, o grau de coesão funcional (baixo grau de acoplamento e altamente reutilizável) das operações num serviço;
- *Service Granularity*, escopo da funcionalidade dos serviços, isto é, serviços mais básicos (*fine-grained*) versus serviços mais complexos (*coarse-grained*) compostos por serviços mais básicos.

### 2.3.2 Introdução à Arquitetura REST

A arquitetura *REST* é uma arquitetura distribuída bastante usada no desenvolvimento de *Web Services* baseada em recursos. De acordo com o sítio *web Programmable Web* [Web16] — o mais popular em termos de notícias e informações sobre *APIs* — 69% das *APIs* utilizadas são *REST*. Este é um valor bastante alto quando comparado com outra arquitetura popular — o *SOAP* — como está ilustrado na figura 2.2, ou na figura 2.3 que mostra o interesse entre *REST* e *SOAP*, usando o *Google Trends* [Goo16] (uma métrica da popularidade de pesquisas no motor de pesquisa *Google*).


Figura 2.2: Popularidade das *APIs* [Web16]

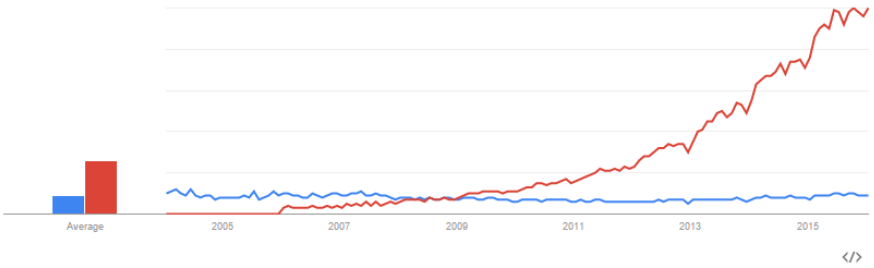



Figura 2.3: Gráfico do *Google Trends* sobre a popularidade de *REST* — a vermelho — e *SOAP* — a azul [Goo16]

Esta permite operações *CRUD (Create, Read, Update, Delete)* e utiliza o protocolo *HTTP* para comunicar, estando os recursos disponíveis num *URI* específico para cada um deles, por exemplo *https://testapp/login*, usando os mesmos códigos de retorno, como está descrito na tabela 2.1, e métodos do protocolo *HTTP* para realizar as várias operações desejadas:

- *GET*, obter um recurso ;
- *POST*, para submeter por exemplo um bloco de dados de um formulário, ou seja, dados para serem processados pelo recurso identificado;
- *PUT*, adicionar informação a uma entidade já existente, se não, esta é criada;
- *DELETE*, apagar o recurso especificado;
- *HEAD*, semelhante ao método *GET*, mas apenas retorna os *headers* e não o *body*, sendo útil para obter meta-informação sobre o recurso especificado, sem obter o corpo (dados) do mesmo;

- *OPTIONS*, que retorna os métodos que o recurso especificado suporta.

Tabela 2.1: Códigos de retorno do protocolo *HTTP* [W3C16a]

| Código | Descrição | Exemplo |
|---|---|---|
| 100 | Informativo | 100 Continue |
| 200 | Sucesso | 200 OK |
| 201 | Recurso Criado | 201 Created |
| 202 | Pedido aceite mas ainda não processado | 202 Accepted |
| 301 | Recurso redirecionado | 301 Moved Permanently |
| 304 | Recurso não modificado | 304 Not Modified |
| 400 | Erro do cliente | 400 Bad Request |
| 401 | Não autorizado | 401 Unauthorized |
| 402 | Pagamento necessário | 402 Payment Required |
| 403 | Proibido | 403 Forbidden |
| 404 | Recurso não encontrado | 404 Not found |
| 500 | Erro do servidor | 500 Internal Error |
| 501 | Recurso não implementado | 501 Not Implemented |
| 503 | Serviço Indisponível | 503 Service Unavailable |

A arquitura *REST* impõe uma série de restrições [Fie00]:

- *Null Style*, significa que não existem limites distintos entre componentes;
- Cliente-Servidor, que separa as funções entre a entidade que faz pedidos de serviços - o cliente - e a que lhe fornece esses serviços e que contém todos os dados necessários - o servidor, permitindo que os componentes evoluam separada e independentemente;
- *Stateless*, significa que não é guardado nenhum contexto do cliente no servidor entre pedidos, ou seja, é sempre enviada toda a informação necessária para o servidor servir o pedido, e como tal toda a informação do estado da sessão é guardado do lado do cliente;
- *Cacheable*, isto é, os clientes ou intermediários podem fazer *cache* das respostas, como tal, é necessário definir se deve ser feito *cache* às respostas ou não. Se for possível fazer *cache* das respostas, então em pedidos equivalentes pode-se usar essa mesma resposta, não sendo necessário comunicar com o servidor novamente, melhorando a eficiência, escalabilidade e latência da interação com o utilizador;
- *Layered System*, ou seja esta arquitetura está organizada por camadas, cada uma apenas se preocupando com a imediatamente adjacente, promovendo desta forma a independência de componentes, sendo mais fácil integrar componentes bem como reutilizar componentes antigos (*legacy*) e reduzindo a complexidade. Adicionalmente um cliente não sabe se está a comunicar com um servidor intermediário ou com o final, pois podem existir intermediários que melhorem a escalabilidade e *load balancing*;

- Interface Uniforme, os recursos são acedidos através de um conjunto fixo de métodos (*GET*, *PUT*, *DELETE* e *POST*), simplificando a arquitetura.

- *Code-On-Demand* (opcional), consiste na possibilidade de atualizar o código do lado do cliente sem ser necessário atualizar também o código do lado do servidor, por exemplo na forma de *applets* ou *scripts*.

Se estas restrições forem cumpridas, os sistemas podem ser considerados como *RESTful*. Como é possível verificar, esta tecnologia é uma base para desenvolver serviços e arquiteturas mais complexas, nomeadamente os *middlewares* descritos a seguir utilizam esta tecnologia para implementar os seus serviços e abstrações. Estes utilizam esta tecnologia visto que é eficiente em termos do uso da largura de banda e é menos verbosa que o *SOAP*. O cliente apenas necessita de saber qual o *URI* do recurso e qual o seu tipo de dados. Como é desenhada para ser *stateless*, é possível fazer *cache* dos resultados, oferecendo um melhor desempenho e escabilidade. Adicionalmente, como o cliente e o servidor estão pouco ligados (*loosely coupled*), o servidor pode alterar os recursos expostos sem qualquer problema. Por fim, o *REST* suporta vários tipos de dados, como o *JSON*, não suportando apenas *XML* como no caso do *SOAP*.

### 2.3.3 FIWARE

O *FIWARE* é um *middleware* - impulsionado pela União Europeia[2] - que consiste em várias *APIs* que facilitam o desenvolvimento de aplicações para *Smart Cities*. É de utilização gratuita e a sua especificação é pública, pretendendo desta forma facilitar a sua adopção e atenuar a curva de entrada. Está organizado em *Generic Enablers*[3]: componentes de software que têm como objetivo facilitar o desenvolvimento de aplicações complexas em diferentes áreas, por exemplo, componentes de segurança, gestão de dados, interface *web* para utilizadores, entre outros.

Um destes componentes, orientado à gestão de dados, é o *Publish/Subscribe Context Broker - Orion Context Broker*[4], que se enquadra neste contexto. Este implementa o modelo de *publish/-subscribe* — já descrito acima na secção 2.1.2 — utilizando as interfaces *NGSI9* e *NGSI10*. Este componente usa estruturas de dados genéricas denominadas de *Context Elements* para representar a informação. Referem-se a informações produzidas, colecionadas ou observadas para serem processadas, analisadas e extraída informação destas, como por exemplo um sensor de temperatura num sala. São representadas utilizando *JSON* [ECM16], compostas por uma série de campos (normalmente uma sequência de triplos <*name,type,value*>) denominados de *Context Element Attributes* [FIW16]:

- *id* da entidade;

- *type*, tipo da entidade;

[2] https://www.fi-ppp.eu/
[3] http://catalogue.fiware.org/enablers
[4] http://catalogue.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker

- *attributes*, um *array JSON*, onde são definidos os vários atributos com os seguintes parâmetros:
  - *name*, nome do atributo;
  - *type*, tipo do valor do atributo: inteiro, *float*, *string*, entre outros;
  - *value*, valor do atributo;
  - *metadatas*, um *array* opcional onde é possível definir meta-dados afetos a um atributo, como a precisão de um atributo do tipo *float*, média de um valor (por exemplo a média de todos os valores deste atributo que tenham sido publicados), ou, no caso do atributo ser do tipo *position* (localização geográfica, latitude e longitude), este campo é obrigatório. Este *array* tem os seguintes valores:
    * *name*, nome do meta-dado, "location"no caso de o atributo ser do tipo *position*;
    * *type*, tipo do meta-dado, "string"no caso de o atributo ser do tipo *position*;
    * *value*, valor do meta-dado, "WGS84"(sistema standard de coordenadas) no caso de o atributo ser do tipo *position*.

Uma representação visual do diagrama *UML* da relação entre os *Context Elements* e os seus atributos está disponível na figura 2.4.

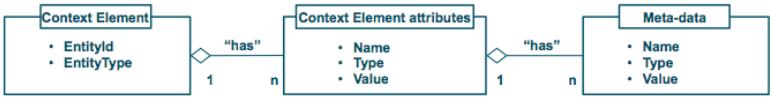



Figura 2.4: Diagrama *UML* da entidade e os seus atributos como definido pela especificação *OMA NGSI* [FIW16]

O *FIWARE* é *RESTful*, como tal todas as operações são realizadas utilizando os métodos *HTTP* já referidos na secção 2.3.2. Existem 2 versões da *API* utilizada para comunicar com este *broker*, v1 e v2. Na primeira, as interfaces *NGSI9* e *NGSI10* têm 2 tipos de operações, conveniência — operações que facilitam o uso de implementações de *NGSI*, definidas pelo próprio projeto do *FIWARE* — que complementam as operações standard — definidas diretamente pela especificação *OMA NGSI* [All12]. Em termos de operações standard [Tel16] existem 5, todas elas utilizando o método *HTTP POST*:

- *updateContext*, que permite criar entidades, com o parâmetro *updateAction* definido como *APPEND*, atualizar uma entidade, com o parâmetro *updateAction* definido como *UPDATE* e por fim apagá-la, com o parâmetro *updateAction* definido como *DELETE*;
- *queryContext*, permitindo obter toda informação sobre uma entidade, apenas o valor de um ou mais atributos da mesma, o valor de um ou mais atributos de várias entidades do mesmo

tipo. Para tal, é utilizada uma expressão regular, por exemplo, todas as entidades cujo atributo *ID* começa com *Room*, sendo neste caso usada a expressão regular *Room.**, definindo o atributo *isPattern* como verdadeiro. Embora seja similar à operação conveniente equivalente, permite fazer *queries* tendo por base a localização geográfica da entidade (latitude e longitude). É possível inclusive definir círculos, indicando a latitude e longitude do centro da circunferência e o seu raio. Também é possível definir polígonos, definindo todos os seus vértices, para determinar a área sobre a qual obter informação sobre as entidades. Por fim, é possível indicar se se pretende usar a área interna ou externa do círculo/polígono;

- *subscribeContext*, que permite subscrever uma entidade, nos mesmos moldes descritos acima;
- *updateContextSubscription*, permitindo atualizar a subscrição como definido anteriormente;
- *unsubscribeContext*, que permite cancelar uma subscrição.

Em termos de operações de conveniência, são definidos 3 tipos:

- *contextEntities*, onde é possível criar entidades através do método *HTTP POST*. Uando o método *HTTP GET* é possível obter todas as entidades existentes, informação sobre uma entidade usando o mesmo método. Com o método *HTTP PUT* é possível alterar valores de um ou mais atributos de uma entidade usando o método. Por fim, é possível apagar a entidade, com recurso ao método *HTTP DELETE*;
- *contextTypes*, que possibilita obter todos os tipos de entidades, detalhes (de momento apenas os seus atributos) de um tipo de entidade, obter toda a informação e os valores dos atributos usando o método *HTTP GET*;
- *ContextSubscription*, sendo possível subscrever entidades, escolhendo também quais os atributos dessa entidade sobre os quais receber notificações, o *URL* no qual receber as notificações, a duração da subscrição, o tipo de método de notificação (receber apenas notificações quando houver uma publicação de dados ou fazer *polling*) usando o método *HTTP POST*. Através do método *HTTP Put* pode-se atualizar a subscrição e mudar o método de notificação. Finalmente, é possível cancelar a subscrição através do método *HTTP DELETE*.

A segunda versão[5] realiza as mesmas operações que a primeira. No entanto há algumas diferenças: a estrutura da *ContextEntity* é diferente dado que já não existe o *array attributes*, os atributos são declarados diretamente sob a forma de um objeto *JSON* e os metadados da mesma forma, um objeto dentro do objeto do atributo. É possível filtrar resultados/fazer *queries* às entidades através de uma string (que não é possível com entidades criadas com a *API* da versão v1). Por fim, nas *queries* geográficas não é possível definir circunferências, bem como algumas diferenças pontuais no corpo de algumas operações.

Em termos de limitações deste *middleware*, existem algumas[6]:

[5] http://telefonicaid.github.io/fiware-orion/api/v2/stable/
[6] https://fiware-orion.readthedocs.org/en/develop/index.html

- o tamanho máximo dos pedidos de publicação de dados não pode exceder 1 *MB*;
- uma notificação não pode ser maior que 8 *MB*;
- a soma do tamanho do *id* do *Context Element* com o tipo e o caminho do serviço + 10 não pode ser exceder 1024 aquando da criação de um *Context Element*, por motivos relacionados com a camada de *MongoDB*[7].

Para efetuar o controlo de acessos, este componente pode ser combinando com o *Steelskin PEP*[8], uma camada de autenticação independente do *broker*, que verifica se existem permissões para aceder a um dado recurso, intercetando o pedido antes de este chegar ao *broker*. As informações que são verificadas são para atribuir permissão são as seguintes:

- um *token* gerado pelo servidor de autenticação que usa *OAuth*[9], sendo que este *token* é necessário ser gerado apenas uma vez (não tem data de expiração) e vai nos pedidos no *header x-auth-token*;
- *ServiceId*, que é obtido através do *header fiware-service* e identifica o componente protegido por este mecanismo;
- *SubServiceId*, obtido através do *header fiware-servicepath* e identifica futuras sub-divisões do serviço;
- por fim, a ação desejada.

O *Publish/Subscribe Context Broker - Orion Context Broker* pode deste modo ser visto como um intermediário entre os dispositivos de *Internet of Things* - como os sensores - e as aplicações finais direcionadas aos utilizadores.

### 2.3.4 M2M

As comunicações *M2M (Machine to Machine* permite que dispositivos sem-fios e com fios e serviços troquem informação ou a controlem sem necessidade de intervenção humana. [PA14]

É bastante útil nomeadamente como um *enabler* de *Internet of Things*. Permite por exemplo [DSs12]:

- disponibilizar dados sobre vários sensores publicamente (podendo usar o modelo *publish/-subscribe*);
- atualizar *placards* eletrónicos de publicidade remotamente;
- conectar os vários sensores/dispositivos à *Internet*, a denominada *Internet of Things*, de modo a processar os dados recolhidos por estes;

[7] https://www.mongodb.com/
[8] https://github.com/telefonicaid/fiware-pep-steelskin
[9] http://oauth.net/

- automatizar operações numa casa.

O standard *ETSI M2M* é atualmente a referência para comunicações *M2M* globais e *end-to-end* ao nível de serviços, facilita a integração de dispositivos e a interoperabilidade dos serviços *M2M* na medida em que:

- oferece uma arquitetura *M2M* com uma série de capacidades genéricas para serviços *M2M*;
- reduz a complexidade;
- providencia um modelo standardizado de recursos, facilitando a interoperabilidade;
- facilita o desenvolvimento de aplicações verticais, pois com este encapsulamento providenciado por este standard, são "escondidos"os detalhes de mais baixo nível e como tal o esforço necessário para desenvolver aplicações deste género é mais reduzido.

Por outro lado facilita o desenvolvimento de aplicações *M2M*. Sem standards isto não seria possível, devido à heterogeneidade das aplicações de *M2M*. Desta forma as *APIs* ou protocolos seriam diferentes, tornando a interação entre objetos usando aplicações de *M2M* não standardizadas difícil.

Em termos da sua arquitetura, tem como base os standards atuais de redes e aplicações, em conjunto com *Service Capability Layers*, que são *Service Capabilities* num dispositivo (DSCL), *gateway M2M* (GSCL) e na rede (NSCL), que possibilitam por exemplo suporte para vários protocolos de gestão, e aplicações *M2M*. As entidades mais importantes são:

- dispositivo *M2M*, que executa aplicações usando capacidades *M2M* e funções de rede;
- *gateway M2M*, que garante aos dispositivos interligação à rede (por exemplo através de uma rede sem-fios ou móvel) e interoperabilidade entre eles;
- aplicações *M2M*, que executam lógica de serviço e utilizam *Service Capabilities*, acessíveis via interfaces abertas, que se encontram na *GSCL* (*Gateway Service Capability Layer*) ou na *DSCL* (*Device Service Capability Layer*);
- *M2M Area Network*, liga os dispositivos, quer sejam compatíveis ou não com o standard da *ETSI*, via rede, por exemplo *Bluetooth* ou *Wi-Fi*, ao *gateway M2M*;
- domínio da rede e aplicações *M2M*, oferece conetividade entre *gateways M2M* e aplicações *M2M*;
- aplicações de rede *M2M*, aplicações *M2M* definidas acima mas que se encontram no domínio da rede e aplicações, na *NSCL* (*Network Service Capability Layer*).

Na figura 2.5 está ilustrada a arquitetura deste standard.

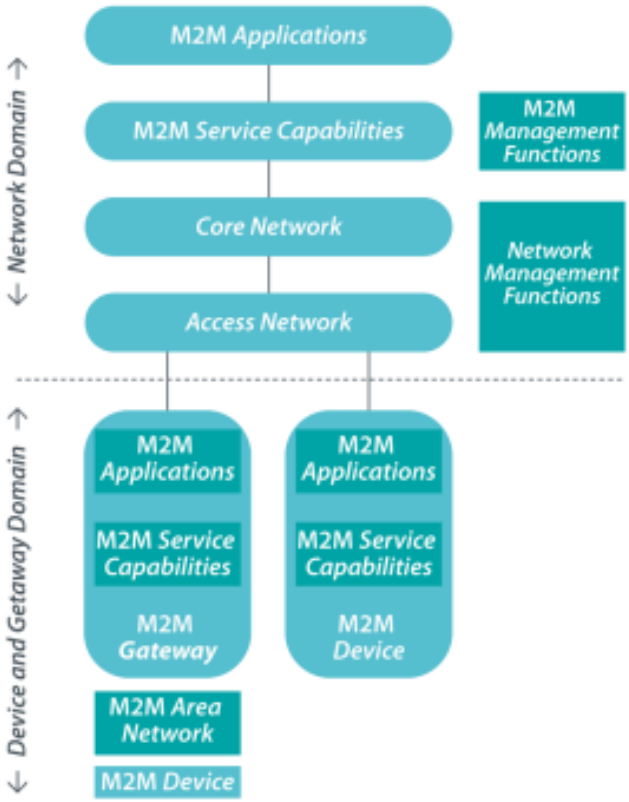



Figura 2.5: Arquiterura do standard de *M2M* da *ETSI* [Lab16]

Em termos de permissões de acesso às operações possíveis presentes numa *Service Capability Layer* existem 5: poder ler, escrever, apagar, criar filhos e descobrir aplicações.[Ets13]

Este standard utiliza *REST* e é *RESTful*, utilizando esta arquitetura para definir como os vários intervenientes comunicam entre si, bem como definindo a forma como a informação é representada. A informação é representada na forma de recursos que são estruturados como uma árvore. [ETS16b]

Os dados estão organizados em diferentes tipos de recursos que são guardados na *Service Capability Layer* respetiva, por exemplo, se for uma aplicação no domínio da rede então é guardada na *NSCL* [ETS16a]. Os vários recursos estão descritos na figura 2.6.

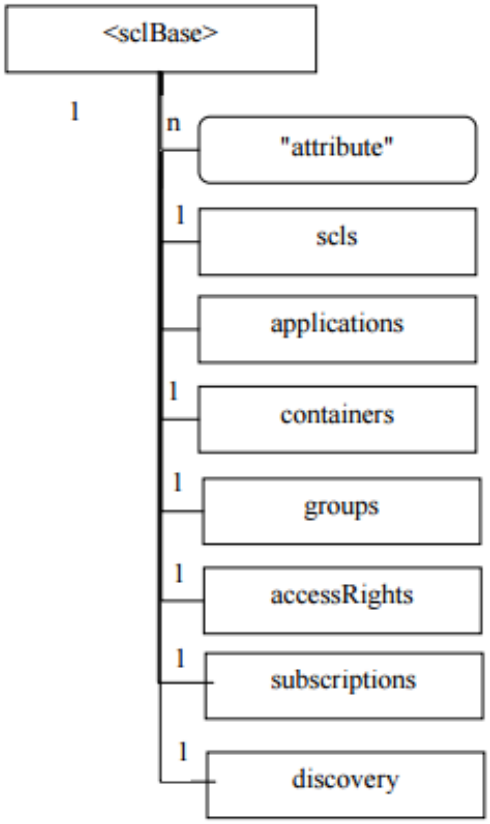



Figura 2.6: Vários tipos de recursos especificados pelo standard de *M2M* da *ETSI* [ETS16a]

Com esta tecnologia são possíveis comunicações síncronas, utilizando o modelo de comunicação *request/response*, e comunicações assíncronas, utilizando o modelo de comunicação *publish/-subscribe*.

Em termos práticos, é possível por exemplo criar aplicações no domínio da rede, definindo o seu *id*, ponto de contacto e a sua data de expiração, além do *URL* e a porta onde o *NSCL* está disponível. A aplicação *ETSI M2M* regista-se neste *NSCL*, autenticando-se através do protocolo *TLS* [The08]. A partir deste momento é possível disponibilizar os dados tanto na forma de *request/response* ou *publish/subscribe*. Estes podem estar organizados em *containers*,que têm um data de expiração, número máximo de instâncias de dados, tamanho máximo do *container* em *bytes* e são associados a uma aplicação *ETSI M2M*. Dentro de estes *containers* são colocadas *content instances* que podem ser de vários tipos de dados, por exemplo *JSON*, e o seu conteúdo varia de acordo com a sua finalidade, não existindo um modelo de dados definido.

Por fim, dado que esta tecnologia não disponibiliza nenhuma biblioteca ou *API* que implemente os standards definidos, apenas define o standard - modelo de dados, arquitetura funcional, *binding* dos protocolos, entre outros - é necessário desenvolver uma biblioteca/*API* para trabalhar com ela.

### 2.3.5 CitySDK

*CitySDK* é um *SDK* para desenvolver serviços e aplicações para *Smart Cities* no domínio da mobilidade, turismo e participação, que tem como objetivo normalizar e "abrir"as *APIs* entre cidades europeias (uma aplicação criada numa cidade europeia pode ser reutilizada noutra), permitindo desenvolver rapidamente aplicações bem como reutiliza-las. Foi implementado em 8 cidades europeias — Amesterdão, Barcelona, Helsínquia, Istambul, Lâmia, Lisboa, Manchester e Roma — com base numa arquitetura orientada a serviços (*SoA*), já definida na secção 2.3.1, em que uma aplicação é composta por um conjunto destes.

Cada um dos domínios referenciados (mobilidade, turismo e participação) tem uma *API* específica. Em termos do turismo, a *Tourism API* [Cit16e] é *RESTful* e utiliza *JSON* para definir o corpo dos pedidos. De modo a uniformizar a informação disponível sobre monumentos, eventos, edifícios (bibliotecas, museus), entre outros, foram definidos 3 conceitos essenciais para representar todos estes dados:

- *Point of Interest (POI)*, que oferece informações básicas sobre cada ponto de interesse da cidade, nomeadamente o nome, descrição, localização em coordenadas, endereço, telefone ou outros contactos (sítio *web*, *email*, fax, *skype*), horas nas quais está disponível e por fim o preço. Adicionalmente, um *POI* pode conter fotos/vídeos/modelos 3D, *links* para aplicações sobre estes, *tags* (por exemplo um hotel e um restaurante direcionados ao público jovem disporem de uma *tag* que indique isso), a ocupação atual ou o tempo médio para poder visitar no caso de se aplicarem. Finalizando, podem existir relações entre *POIs*, nomeadamente uma loja que está dentro de um centro comercial, que está também dentro de uma cidade, facilitando a pesquisa de pontos de interesse;

- *Route*, uma sequência de pontos de interesse, permitindo obter circuitos de pontos de interesse para uma visita à cidade pelos turistas. Em termos de campos, tem o nome, descrição, conteúdo multimédia, métodos de identificação, forma, categorias, *tags* e *links* para aplicações;

- *Event*, acontecimentos que têm um dada hora e lugar (por exemplo uma exibição temporária). Em termos dos atributos, utilizam os mesmos que os pontos de interesse, no entanto, em vez da localização em coordenadas, a sua localização é um *POI*.

Em termos de serviços, é possível pesquisar pontos de interesse, rotas e eventos por nome, descrição, localização, categoria e *tags* (como tal, também se pode obter todas as que existem, para poder pesquisar por elas), relação com outras entidades e a data da ocorrência (aplicado apenas a eventos). Em relação à arquitetura, é composta por 3 componentes essenciais:

- *Data Adapter*, que extrai os *datasets* sobre os assuntos já referidos acima - com o seu acesso específico (*web service*, ficheiro, etc) e com formato *XML*, *JSON*, *CSV*, entre outros - que a própria cidade providencia;

- *CitySDK Tourism Engine* que processa os dados extraídos pelos *Data Adapters* e os guarda na sua base de dados num formato apropriado e numa estrutura de modo a facilitar a leitura e *throughput* dos pedidos das aplicações que vêm através do componente *CitySDK Tourism Service Provider* e é responsável por fazer *polling* periodicamente às fontes de dados disponíveis da cidade e atualizar a sua base de dados com os novos valores;

- *CitySDK Tourism Service Provider*, que representa o *frontend* da aplicação. É responsável por servir os vários pedidos de dados, e para cada pedido utiliza o *CitySDK Tourism Engine* que obtém a informação necessária presente na sua base de dados.

A *Tourism API* obedece à recomendação *W3C POI* [W3C16c] para representar o formato das mensagens dos pontos de interesse com os seguintes campos.

Os eventos usam apenas alguns campos usados no ponto de interesse, nomeadamente o *ID*, nome, descrição, categoria, *tags*, data, localização, imagens e preço. As rotas são representadas por um grupo de pontos de interesse, ordenados de forma ao utilizador os visitar de uma dada forma (por exemplo cronológica) com uma descrição mínima de cada um.

Em termos da mobilidade [Cit16d], a *Mobility API* é *RESTful* e coleciona *open data*, descreve-a, obtém *links* dos dados para ficheiros de referência (por exemplo *OpenStreetMap*[10]), distribui estes dados via um *web service* unificado (*API*) e permite aos utilizadores anotar dados e/ou objetos. É classificada como *Linked Data* de 5 estrelas[11], que significa que:

- Está disponível na web (o formato não interessa) com uma *open license*, de modo a ser considerada *open data*.

- Disponível de forma estruturada, de modo a ser possível ser lida por uma máquina, por exemplo, no formato *Excel* em vez de ser apenas uma imagem com os dados (por exemplo uma captura de ecrã de uma tabela com dados).

- Está disponível num formato não-proprietário, como o *CSV*.

- Utiliza standards abertos do *W3C* como o *RDF*[12] ou o *SPARQL*[13] para identificar os recursos, de modo a estes poderem ser referenciados por outras entidades.

- É feito um *link* dos dados desta a dados de outras pessoas, de modo a estas obterem contexto sobre estes.

São agrupados todos os *datasets*, *data catalogs* e *API* abertos existentes, de modo a poderem ser acedidos num sítio único e unificado [Cit16d]. Estes dados são nomeadamente horários dos transportes públicos, informação meteorológica, dados do *OpenStreetMap* entre outros.

[10] https://www.openstreetmap.org/
[11] https://www.w3.org/DesignIssues/LinkedData.html
[12] https://www.w3.org/RDF/
[13] https://www.w3.org/TR/rdf-sparql-query/

Estes dados são organizados em nós (cada um com um *ID* único) - que podem ser uma paragem de autocarro, cidade, linha de transporte público, entre outros - no entanto há 4 nós especiais:

- *route*, conjunto de nós numa dada ordem, que podem ter um ou mais meios de transporte dos seguintes: comboio, elétrico, metro, autocarro, *ferry*, teleférico, gôndola, funicular, avião, pé, bicicleta, mota, carro, camião, cavalo;
- *region*, um nó que representa uma região administrativa formal, com 5 níveis de 0 a 5, sendo que conforme o número aumenta, este corresponde uma área mais pequena, por exemplo nível 0 é o país em si, e 5 um bairro de uma dada cidade;
- *ptstop*, uma paragem de um transporte público;
- *ptline*, uma linha de um transporte público.

Os dados poderão ainda estar organizados em camadas, que estruturam os dados e através das quais é possível aceder-lhes e atualiza-los. Cada camada tem um dono que é responsável pelos dados existentes na mesma. Existem 3 camadas: *osm* que contém os dados geográficos do *Open Street Maps*, *admr* que contém as regiões e por fim *gtfs* que contém os dados dos transportes públicos.

Em relação às operações, é possível obter todos os nós, rotas, regiões, paragens e linhas de transportes públicos bem como limitá-los a uma região específica ou especificar pares <camada, valor>. Com estes é possível, nomeadamente, retornar nós que contenham museus na camada *osm*, obter todas as camadas, adicionar, atualizar e apagar dados de uma camada, e por fim fazer *match/link* de *datasets* com os nós já existentes do *CitySDK Mobility*, que permite verificar se objetos (como estações de comboio ou horas de abertura de museus) existentes num dado *dataset* também se encontram no *CitySDK*.

Por último, o domínio da participação tem a *Smart Participation API*, consistindo num canal (interface aberta) para os cidadãos reportarem problemas existentes na sua cidade, baseado na tecnologia *Open311*[14], um protocolo standardizado para acompanhamento de problemas colaborativos baseados numa localização, ajudando também os programadores a criar aplicações que facilitem o *feedback* dos cidadãos e permitindo que estes dêm o *feedback* via nomeadamete aplicações virtuais, possibilitando que estes possam contribuir diretamente para melhorar a sua cidade [Cit16c]. Em termos de operações, é possível [Cit16b]:

- obter os tipos de *service requests* disponíveis (por exemplo reportar "buracos na estrada") e os seus códigos, que dependem da cidade/jurisdição;
- obter a definição de um *service request*, isto é, os seus atributos (nomeadamente o tipo de *feedback*), que podem ser únicos a cada cidade;
- criar *service requests*, indicando entre outros o tipo, localização em coordenadas, identificação da pessoa (nome, *email*, etc) endereço ou descrição do local, dados de multimédia (por exemplo imagens);

[14] http://www.open311.org/

- obter o estado atual de um ou vários pedidos, sabendo se ainda está aberto ou fechado, detalhes sobre o estado corrente, a data expectável do pedido estar resolvido, entre outros.

## 2.4 Conclusões

De acordo com a revisão bibliográfica efetuada, especificamente em termos dos trabalhos relacionados, não foi encontrado nenhum standard para o *benchmark* de *middlewares* neste contexto (*internet of things* em *smart cities*), apenas a intenção de criar um standard para estes *benchmarks*, e alguns *benchmarks* remotamente parecidos, como já referido anteriormente.

Foram revistos estes quatro *middlewares* visto serem os que se tinha conhecimento que seriam aplicáveis neste contexto.

Em termos das tecnologias aqui apresentados, é possível verificar que todas têm diferenças, sendo que algumas tecnologias são menos genéricas que outras. Neste sentido, o *REST* aparece como a mais genérica, visto apenas definir os métodos pelos quais os intervenientes podem comunicar, bem como os seus códigos de retorno. A seguir segue-se o *ETSI M2M* que já define onde colocar os dados e oferece controlo de acessos. Um pouco menos genérico é o *Orion Context Broker* do *FIWARE*, que já define a estrutura dos recursos e permite *geo queries* e *filter queries*. Por fim aparece o *CitySDK* que define exatamente quais os cenários de *Internet of Things* em que atua, toda a estrutura e todos os campos para cada objeto que possui, que nas tecnologias anteriores não acontece, limitando-o em termos da aplicabilidade a vários cenários de *Internet of Things*, sendo desta forma o *middleware* menos genérico e abstrato.

*REST* e o *CitySDK* funcionam num sistema de cliente-servidor, enquanto que o *FIWARE* e o *ETSI M2M* funcionam através de *brokers*, que tratam de toda a comunicação entre *publishers* e *subscribers*, bem como as comunicações com a base de dados. No entanto, todos os *middlewares* são *RESTful*.

Para finalizar, de modo a poder utilizar o *CitySDK*, este necessitaria de ser estensível, isto é, poder adicionar dados à sua plataforma, o que implicaria que esta tivesse uma *API* para esse efeito, que não existe. Também foi apenas adotada e implementada pela cidade de Lisboa em Portugal.

Destes quatro *middlewares* aqui apresentados, apenas três foram escolhidos para as análises qualitativas e quantitativas visto serem os únicos aos quais se tem acesso (no caso do *FIWARE* e *M2M* acesso a um *broker*) para desenvolver serviços e aplicações.

# Capítulo 3

# Problema e Metodologia

## 3.1 Problema

Como é possível verificar, existem vários *middlewares* para o desenvolvimento de aplicações e serviços para *Smart Cities*, no entanto, não existe ainda uma comparação qualitativa e quantitativa entre estes.

Visto que estes são construídos pensando em cenários genéricos, esta comparação é necessária para ser possível perceber como os *middlewares* se comportam em várias circunstâncias diferentes. É procurado esclarecer todos estes pontos acerca destes:

- o seu desempenho;
- as suas limitações;
- o seu funcionamento, como por exemplo a gestão de vários pedidos em simultâneo, dado que todos eles são *RESTful* e para efetuar operações, como uma publicação, é necessário enviar um pedido *web*.
- como estão organizadas e são compostas as mensagens trocadas ao nível da rede e a quantidade de *overhead* imposto pelo *middleware*;
- por fim, como é o desenvolvimento de aplicações com estes, usando os dados e serviços disponíveis. Isto é, quais as suas limitações, a sua viabilidade, que ferramentas estão disponíveis, se existe uma comunidade na *Internet* de *developers* destes *middlewares*, entre outros.

Para realizar a análise quantitativa, um *benchmark* dos vários *middlewares* referidos no capítulo anterior (excepto o *CitySDK*, pelas razões já referidas no capítulo anterior na secção 2.4), é necessário definir os cenários nos quais o *benchmark* será efetuado. Adicionalmente, é preciso estudar a aplicabilidade de todos os *middlewares*, já descritos anteriomente no capítulo 2, a cada um dos cenários e definir quais as métricas que são relevantes em cada um, em que nível e como as efetuar.

Dado que neste contexto existem vários cenários diferentes com requisitos díspares entre si e estes *middlewares* compõem sistemas *black-box*, ou seja, não existe nenhuma informação sobre como é o seu funcionamento interno, esta análise procura compreender o seu funcionamento, definindo o comportamento interno dos mesmos.

Por fim, é necessário definir o caso de uso sobre o qual os cenários irão assentar.

## 3.2 Metodologia

### 3.2.1 Introdução

Como os todos os *middlewares* apresentados são *RESTful* e funcionam à base de pedidos *web*, é necessário observar o tráfego da rede. Existem três formas distintas de obter medições de desempenho na rede [Rou05]:

- medições diretas, nomeadamente dados da rede colecionados pelo protocolo *SNMP* [IET16] — que coleciona dados de dispositivos como o *router* ou *switch* — normalmente de forma intermitente, devido a limitações de memória e à ineficiência deste protocolo;
- medições passivas, ou seja, monitorizar um ou mais caminhos (por exemplo de A a B) através de um *packet monitor/sniffer* como por exemplo o *Wireshark*[1], de modo a conseguir medições sobre o tráfego existente na rede, podendo nomeadamente obter o *delay* de pacotes entre 2 pontos (medindo o tempo de chegado num e no outro). Devido ao grande volume de tráfego na rede, é usada uma técnica chamada *sampling* na qual o *router* mantém estatísticas sobre o tráfego apenas numa parte (*sample*) do tráfego total da rede. No entanto, é necessária a sincronização de relógios para este tipo de medições;
- medições ativas (*active probing*), no qual é "injetado"tráfego na rede e é medido o tempo que demora a chegar a um destino final ou intermédio. Este método é particularmente útil quando se necessita de um aspeto específico de desempenho na rede. No entanto, tem os mesmos problemas de sincronização que a técnica anterior.

### 3.2.2 Cenários de Comunicação

Para efetuar a análise quantitativa e qualitativa foram utilizados 4 cenários diferentes. Estes estão diretamente relacionados com a aplicação exemplo referida na secção 1.2 do capítulo 1. Esta refere-se à disponibilização do grande volume de dados recolhido pelos sensores publicamente, permitindo que aplicações de terceiros utilizem estes para criar aplicações inovadoras com eles.

Esta disponibilização tem por base o modelo de comunicação *publish/subscribe* e *request/response*. Os cenários visados pela análise quantitativa e qualitativa são os seguintes:

1. publicação de dados, utilizando o modelo *publish/subscribe*, em que um utilizador publica um grande volume de dados continuamente de modo sequencial, sendo estes recebidos por

[1] https://www.wireshark.org/

quem os subscreveu. Na figura 3.1 encontra-se uma representação gráfica na forma de um diagrama *UML* de sequência sobre as interações entre os vários intervenientes neste cenário;

2. publicação de dados, utilizando o modelo *publish/subscribe*, em que vários utilizadores publicam apenas um recurso (visto que são pedidos *HTTP*) sendo este recebido por quem o subscreveu. Desta forma é possível verificar como o *broker* se comporta sobre uma carga diferente — neste caso maior — quais os seus limites e se suporta muitas publicações em simultâneo, para depois comparar com o cenário anterior. Neste caso, este cenário está na forma em que um único computador (um único endereço *ip*) publica vários recursos em paralelo, de modo a simular vários utilizadores distintos a publicarem uma entidade. Na figura 3.2 está uma representação gráfica na forma de um diagrama *UML* de sequência sobre as interações entre os vários intervenientes neste cenário;

3. *request/response*, onde uma entidade requisita um dado serviço através de um pedido *web*, por exemplo obter uma informação sobre um dado recurso com base em parâmetros previamente definidos. Na figura 3.3 é ilustrado um diagrama *UML* de sequência descrevendo as interações entre os vários intervenientes neste cenário;

4. disponibilização de um *dataset*. Na figura 3.3 encontra-se disponível um diagrama *UML* de sequência descrevendo as interações entre os vários intervenientes neste cenário.

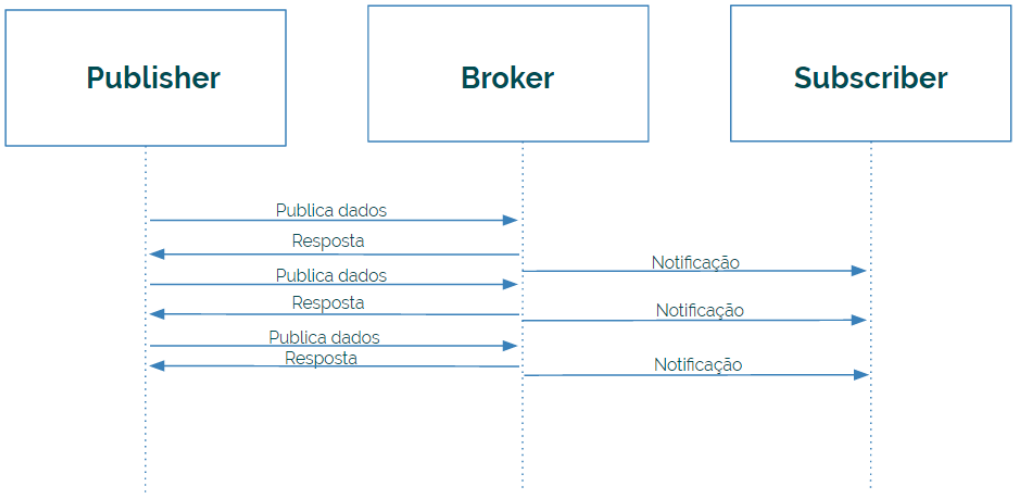



Figura 3.1: Diagrama *UML* de sequência sobre as interações entre os vários intervenientes no cenário *publish/subscribe* sequencial

.

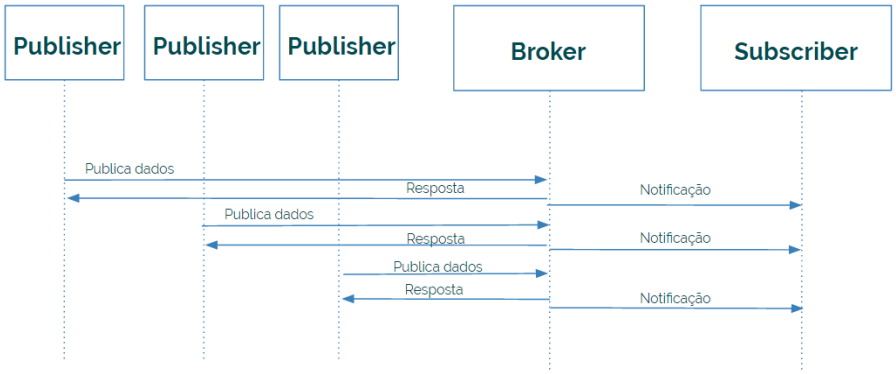



Figura 3.2: Diagrama *UML* de sequência sobre as interações entre os vários intervenientes no cenário *publish/subscribe* paralelo

.

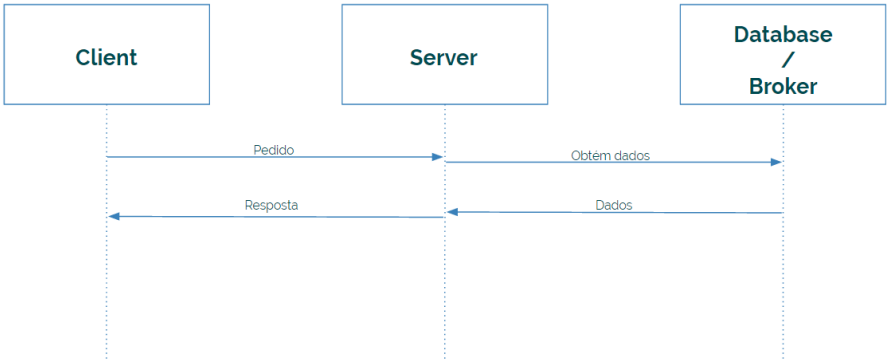



Figura 3.3: Diagrama *UML* de sequência sobre as interações entre os vários intervenientes no cenário *request/response* e de disponibilização de um *dataset*

.

### 3.2.3 Caso de Uso

O caso de uso utilizado consiste na disponibilização da velocidade média dada rua da cidade do Porto em quatro *timeslots*: das 8 as 10 da manhã, das 10 da manhã às 17 da tarde, das 17 da tarde às 19 da tarde, e das 19 da tarde às 8 da manhã. Engloba milhares de dados recolhidos pelos vários sensores móveis e fixos espalhados pela área urbana do Porto, que são agrupados por cada *edge* do *OpenStreetMap*, correspondente a uma rua da área urbana do Porto. Os dados a serem disponibilizados são exatamente 19884 e contêm os seguintes atributos:

- *id* do *edge* do *OpenStreetMap*;
- velocidade média na última hora;
- velocidade média nas últimas três semanas;

- combinação das duas velocidades anteriores, com um peso arbitrário para cada uma;
- latitude;
- longitude.

Em termos dos cenários de comunicação, os dois primeiros (*publish/subscribe*) consistem na publicação da velocidade média num intervalo fixo de uma hora, tendo por base a velocidade média na hora em questão e nas semanas anteriores no *timeslot* correspondente a essa hora. O terceiro permite obter os atributos acima definidos em cada *edge* do *OpenStreetMap*, além de uma série de filtros como por exemplo apenas obter *edges* em que a velocidade média é maior que um dado valor ou obter todos os *edges* contidos numa área geográfica. Por fim, no quarto é possível obter todos os *edges* de um dado *timeslot*.

Visto que são recolhidos milhares de dados, e necessitam de ser publicados (são 19884 dados para publicar), o cenário de *publish/subscribe* com vários pedidos em simultâneo ganha um relevo maior. Isto deve-se ao facto de ser necessário publicar todos estes dados em intervalos fixos de uma hora, sendo que o ideal é que estes sejam publicados rapidamente, de modo a que ser possível utiliza-los para outros efeitos, nomeadamente em aplicações de terceiros.

### 3.2.4 Métricas a Analisar na Avaliação Quantitativa

De modo a poder avaliar os *middlewares* de forma quantitativa e produzir um *benchmark*, é necessário avaliar o seu desempenho, tanto em termos da sua velocidade como em termos da sua eficiência em termos das comunicações ao nível da rede. A velocidade refere-se a tempo para publicar os dados nos dois primeiros cenários, ou para obter os dados de acordo com o pedido efetuado nos restantes. A eficiência em termos das comunicações ao nível da rede é referente ao *overhead* imposto pelos *middlewares* e à evolução em termos do tamanho que os dados tomam, desde que estes são variáveis no código, passando pela transformação num objeto *JSON* e por fim quando estes são incorporados no *payload* (a parte que contém os dados em si, não incluindo informação com os *headers* ou meta-dados) do protocolo *TCP*. Esta evolução do tamanho que os dados vão tomando pode ser verificada na figura 3.4.

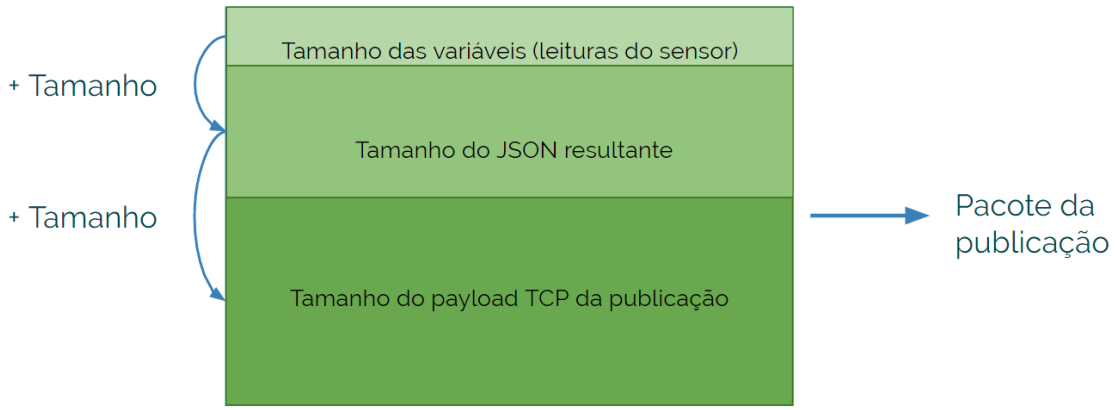



Figura 3.4: Transformação em termos de tamanho dos dados a publicar

.

Em termos das medições a efetuar, foi usado o método de medições ativas — já referido anteriormente na secção 3.2.1 — visto que em todos os cenários são enviados dados pela rede em direcção a um destino predefinido, isto é:

- nos cenários de *publish/subscribe* estes dados correspondem à publicação dos dados (pedidos *web* com os dados);
- no de *request/response* é enviado um pedido *web* para obter informações sobre um dado recurso;
- no de disponibilização de um *dataset* é também enviado um pedido *web* para o mesmo efeito que no cenário anterior.

Adicionalmente, o método das medições ativas foi escolhido dado que um dos objetivos é medir o tempo de chegada dos pacotes enviados.

No primeiro e segundo cenário de comunicação — nos quais o modelo *publish/subscribe* surgiu um pouco como alternativa ao facto de ser necessário fazer *polling* a páginas/recursos *web* para obter atualizações, pois permite obter as atualizações sem a necessidade de o fazer [WQA+02], recebendo-as de forma assíncrona — foram consideradas várias métricas, que podem ser categorizadas em principais ou auxiliares. As principais são as seguintes:

- tempo de publicação (*tPub*), isto é, o tempo decorrido desde que é efetuado o pedido até receber a resposta ao mesmo, exemplificado na figura 3.5;
- tempo de subscrição (*tSub*), isto é, o tempo que passou desde que é efetuado o pedido de publicação até o *subscriber* receber a notificação desse mesmo, como está ilustrado na figura 3.5;
- tempo total para publicar todos os dados;

- tamanho em *bytes* do *header Content-Length* [W3C16b] do protocolo *HTTP* referente ao *JSON* enviado no pedido contendo os dados a publicar;
- tamanho em *bytes* do *payload TCP* do pacote da publicação;
- total de *bytes* da comunicação em cada ligação em ambos os sentidos (cliente->servidor e servidor->cliente). Está englobada aqui toda a comunicação ao nível da rede para publicar um recurso, desde o estabelecimento da comunicação (pacotes do protocolo *TCP* com as *flags SYN* e/ou *ACK*), o pacote da publicação, e por fim o término desta comunicação, referente aos pacotes do protocolo *TCP* com as *flags FIN* e/ou *ACK*.
- *goodput* (fSize/$\Delta$t) da ligação, ou seja, o número de *bytes* úteis enviados num dado período de tempo. Neste caso, estes *bytes* referem-se ao tamanho do *frame* da publicação (*fSize*), tendo em conta o total de tempo da mesma. Este período de tempo refere-se à diferença de tempo entre o pacote *TCP SYN* que sinaliza o início da comunicação e o pacote *TCP ACK* referente ao pacote *TCP FINACK*, que dita o fim da comunicação ($\Delta$t). Um *frame* é a unidade de transmissão num protocolo da *link layer*, que consiste num *header* da *link layer* seguido de um pacote [For16]. Uma representação gráfica está disponível na figura 3.6.

As auxiliares são as seguintes:

- número de *retries*, em casos em que o pedido *web* da publicação não é bem sucedido devido a problemas do *broker*. Isto pode dever-se ao facto de serem recebidas respostas com *status* do protocolo *HTTP* [W3C16a] dentro da gama de 400 ou 500, referentes a erros do cliente e do servidor respetivamente, ou ao facto de ser enviada a *flag RST* do protocolo *TCP*;
- tamanho em *bytes* ocupado pelas variáveis relativas aos dados a publicar, como a velocidade média, a latitude, longitude, entre outros;
- *Round Trip Time* [Com00], o tempo que demora a obter um *acknowledgment* de um dado pacote. Este é obtido através da diferença de *timestamps* entre a resposta ao pacote *TCP SYN* e o mesmo, os dois primeiros pacotes trocados aquando do estabelecimento de uma comunicação entre dois pontos.
- o número de vezes que é necessária a retransmissões de pacotes [Com00] do protocolo *TCP* e *HTTP*. Isto é, ser necessário retransmitir um pacote devido a não se receber um *acknowledgment* dentro de um determinado período de tempo, que pode ser causado nomeadamente pelo excesso de carga no servidor, e também o *delay* registado para a retransmissão em causa;

Combinando estas as métricas principais e auxiliares, é possível observar o seguinte:

- verificar se existe algum aumento espontâneo do tempo de publicação/subscrição se deveu ao aumento do *round trip time* entre os dois pontos;

- verificar se algum aumento do tempo total de publicação em relação a outros se deveu a um aumento dos *retries* ou das retransmissões de pacotes (ou ambos).
- comparando o tempo de publicação, subscrição e *round trip time* (para despistar casos em que este possa aumentar apenas no tempo de publicação/subscrição e deste modo influenciar os resultados) é possível perceber se a notificação relativa à subscrição é enviada antes ou depois — ou ambos — de ser recebido o OK referente ao pedido de publicação (pedido *web* visto que tanto o *FIWARE* e o *ETSI M2M* são *RESTful*) do recurso em questão.
- observar o aumento de tamanho que os dados sofrem aquando da transformação das variáveis no código com o dados a publicar num *JSON*, que já está pronto para ser incluído no pedido de publicação.

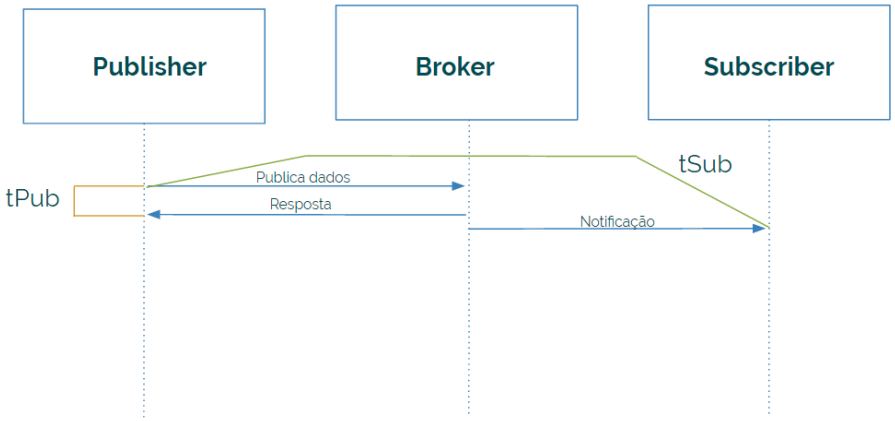



Figura 3.5: Diagrama *UML* de sequência contendo as medições do tempo de publicação e subscrição para os dois primeiros cenários

.

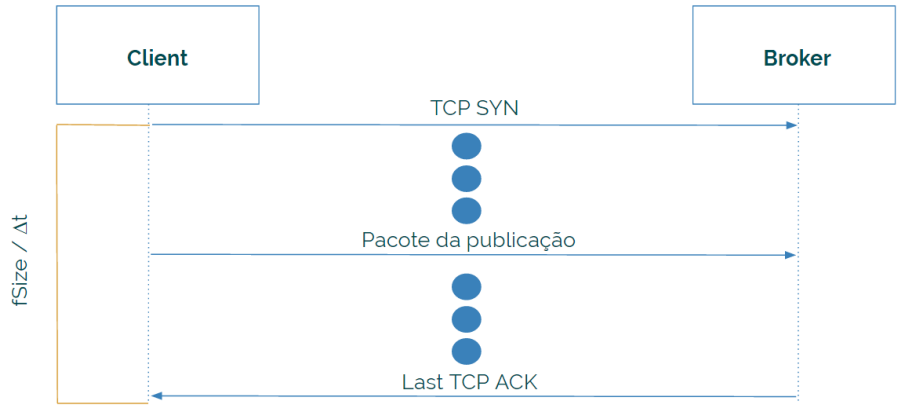



Figura 3.6: Diagrama *UML* de sequência exemplificando a medição do *goodput*

.

No terceiro cenário de comunicação (*request/response*), o objetivo consiste em receber a resposta ao pedido o mais rápido possível, dado que numa aplicação para *smartphones* que necessite de informações para apresentar uma dada vista, nomeadmente um pedido *web* para fazer o *render* desta, o ideal é que a resposta a esse pedido seja rápida de forma a que o utilizador não esteja muito tempo à espera. Como tal, o tempo de resposta é essencial. Adicionalmente, é também interessante registar métricas ao nível da rede, como o *Content-Length* do *JSON* com os resultados enviados referente a um dado pedido *web* e o total de *bytes* utilizados (apenas no sentido servidor->cliente, dado que no sentido cliente->servidor apenas são enviados pacotes de *acknowledgment* aos pacotes recebidos no outro sentido). Estas métricas tornam-se especialmente importantes em redes móveis, onde a largura de banda disponível é significativamente menor e onde a eficiência em termos da comunicação assume um relevo superior[2].

No quarto e último cenário de comunicação, não há muito a comparar em termos quantitativos, dado que o objetivo não é obter o *dataset* o mais rápido ou eficientemente possível, mas sim observar se este pode ser albergado em cada *middleware*. Logo está mais relacionado com aspetos qualitativos como limitações em termos da engenharia de software de cada *middleware* poder albergar milhares de dados, pois os dados estão organizados de formas distintas em cada *middleware*.

### 3.2.5 Tópicos a Analisar na Avaliação Qualitativa

Na avaliação qualitativa são analisados os seguintes pontos:

- comparação das tecnologias com base nos requisitos para aplicações de *Internet of Things* da *Internet of Things Architecture*[3];
- a aplicabilidade de cada tecnologia a cada um dos cenários;
- é feita uma exploração da documentação da *API* de cada tecnologia e os tutoriais disponíveis;
- suporte para resolver *bugs*, nomeadamente como são as comunidades em sítios *web* como o *Stack Overflow*;
- viabilidade de cada tecnologia em cada cenário em termos funcionais, como por exemplo limitações em termos da engenharia de software.

[2] http://www.makeuseof.com/tag/wi-fi-vs-ethernet-which-should-you-use-and-why/
[3] http://www.iot-a.eu/public

# Capítulo 4

# Análise Qualitativa

## 4.1 Comparação das Várias Tecnologias

De modo a obter uma base de comparação sólida entre os vários *middlewares* apresentados no capítulo 2 (com a excepção do *CitySDK*, visto não ser possível implementar este pelas razões já referidas no mesmo capítulo), foi utilizada a definição de requisitos para aplicações de *Internet of Things* da *Internet of Things Architecture*[1]. A *Internet of Things Architecture* é um projeto europeu que tem como objetivo criar uma arquitetura de referência para aplicações de *Internet of Things*. São definidos requisitos tanto de baixo nível como de alto nível e visto que as tecnologias aqui descritas têm como objetivo fazer a ponte entre a informação recolhida pelos sensores e o mundo exterior — empresas/entidades que queiram utilizar esta informação para criar aplicações neste domínio (não sendo necessários requisitos de baixo nível, ao nível dos dispositivos) — foram escolhidos os de mais alto nível. Estes requisitos estão descritos no anexo A.

[1] http://www.iot-a.eu/public

Na tabela 4.1 está descrito se as tecnologias cumprem ou não estes requisitos.

Tabela 4.1: Cumprimento dos vários requisitos por parte de cada tecnologia

| | REST | FIWARE | ETSI M2M |
|---|---|---|---|
| UNI.001 | Sim | Sim | Sim |
| UNI.002 | Sim | Sim | Sim |
| UNI.005 | Não | Sim | Sim |
| UNI.008 | Sim | Sim | Sim |
| UNI.016 | Não | Sim | Não |
| UNI.022 | Sim | Sim | Sim |
| UNI.023 | Sim | Sim | Sim |
| UNI.030 | Sim | Sim | Sim |
| UNI.036 | Sim | Sim | Sim |
| UNI.047 | Sim | Sim | Sim |
| UNI.048 | Sim | Sim | Sim |
| UNI.067 | Sim | Sim | Sim |
| UNI.071 | Sim | Sim | Sim |
| UNI.073 | Sim | Sim | Sim |
| UNI.092 | Sim | Sim | Sim |
| UNI.094 | Não | Sim | Sim |
| UNI.240 | Não | Sim | Não |
| UNI.245 | Sim | Sim | Sim |
| UNI.405 | Não | Não | Não |
| UNI.406 | Não | Sim | Não |
| UNI.426 | Sim | Sim | Sim |
| UNI.607 | Sim | Sim | Sim |
| UNI.608 | Sim | Sim | Sim |

Como é possível observar através desta tabela, os serviços *REST* convencionais são mais *barebones* em relação aos demais aqui apresentados, visto que cumprem menos requisitos. Por consequência, implementam menos funcionalidades, dado que apenas providenciam o essencial para a comunicação entre intervenientes, sendo que funcionalidades como o modelo *publish/subscribe* não são implementadas de base por esta tecnologia. Como tal, o *REST* é o mais genérico, seguindo-se o *ETSI M2M* que já permite comunicações com base no modelo *publish/subscribe* além de estabelecer standards para a comunicação entre várias aplicações e serviços *M2M*, como por exemplo na organização dos dados. Por fim, o *FIWARE* foi o que cumpriu mais requisitos destes três, visto oferecer por exemplo variáveis e *queries* geográficas, além de todas as funcionalidades oferecidas pelos anteriores.

## 4.2 Aplicabilidade das Tecnologias em Cada Cenário

Em termos da aplicabilidade de cada um dos cenários a cada *middleware*, de acordo com a revisão bibliográfica efetuada no capítulo 2, é possível verificar o seguinte:

- o *REST* é naturalmente indicada para o cenário *request/response*, visto ser o funcionamento inerente desta. É igualmente indicada para o cenário da disponibilização de um *dataset*, estando o recurso acessível através do seu *URL* correspondente e do método *HTTP GET*. No entanto, não se adequa ao modelo de *publish/subscribe*, pois este não é naturalmente suportado;

- o *FIWARE*, em conjunto com o *Generic Enabler Orion Context Broker*, que implementa o modelo *publish/subscribe*, possibilita inerentemente a implementação dos cenário deste. Nos dois cenários restantes, a informação está acessível através da operação conveniente *contextEntities* em conjunto com o método *HTTP GET* ou através da operação standard *queryContext*. No cenário *request/response* estas operações são combinadas com *filter queries*, tanto por *String* (*string query*) ou por localização geográfica (*geo query*). No entanto, é necessário criar os recursos (*context elements*) usando métodos da versão v2 da *API* para poder aplicar estes tipos de filtros;

- no *M2M*, é utilizada uma biblioteca para a linguagem de programação *Java*[2], desenvolvida aquando da dissertação do aluno António Pinto, no âmbito do Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos [Pin15]. Esta implementa o standard da *ETSI*, encapsulando toda a sua complexidade através do modelo *publish/subscribe* e é *RESTful*. Como tal, é possível implementar o cenário *publish/subscribe* porque suporta inerentemente este modelo. O cenário de disponibilização de um *dataset* também é suportado, criando um recurso(*container*) para o albergar, estando depois disponível através da operação que obtém as *content instances* do mesmo. Por fim o de *request/response*, utiliza a mesma fórmula do cenário anterior e os filtros são realizados nos mesmos moldes do *REST*, onde são passados parâmetros no *URL* do pedido *web* para serem depois processados.

Em resumo:

Tabela 4.2: Aplicabilidade de cada tecnologia a cada cenário

| | Dataset | Publish/Subscribe | Request/Response |
|---|---|---|---|
| REST | Sim | Não | Sim |
| FIWARE | Sim | Sim | Sim |
| ETSI M2M | Sim | Sim | Sim |

[2] https://www.java.com/en/about/

## 4.3 Documentação da API e Tutoriais Disponíveis

O *FIWARE* tem um manual (*API walkthrough*) disponível *online* para cada uma das versões da *API*, onde detalha todas as operações que esta disponibiliza bem como configurar e criar um *broker*. Existe também uma página *web* com uma especificação mais curta e direcionada apenas às operações da *API*, contendo exemplos de como efetuar os vários pedidos em várias linguagens de programação, como *Java*, *Javascript*, *Node.js*, *Perl*, *Python*, entre outros. Como a segunda versão da *API* ainda está em beta, esta página muda muito regularmente, bem como a própria especificação da *API*, adicionando novas operações ou modificando a sintaxe de operações já existentes. No entanto, existem duas páginas *web* diferentes para esta, uma com uma versão estável, e outra com a versão mais atualizada (mas ainda um pouco instável).

No caso do *ETSI M2M*, existem vários documentos especificando os vários aspetos do standard da *ETSI*, desde o *binding* de protocolos como o *HTTP* até à sua arquitetura funcional, disponíveis no url http://www.etsi.org/technologies-clusters/technologies/internet-of-things. Ao contrário do *FIWARE*, o standard *M2M* da *ETSI* apenas define o standard em si, não providencia nenhuma biblioteca ou implementação de referência, sendo necessário implementar o standard para poder utilizar esta tecnologia. Visto que foi fornecida uma biblioteca para trabalhar com este *middleware*, e dado que esta é uma biblioteca fechada que pertence à *Altice Labs*, foi disponibilizado um guia para a utilização da biblioteca, detalhando a sua estrutura e quais as operações que esta permite bem como *snippets* de código das operações.

## 4.4 Suporte

Existe uma *tag* no sítio web *Stack Overflow* com questões sobre o *broker* utilizado com o *FIWARE* (o *Orion Context Broker*), com 358 questões no momento da escrita (01/06/2016), disponível no seguinte endereço: http://stackoverflow.com/questions/tagged/fiware-orion. Está disponível também um *FAQ* sobre a plataforma *FIWARE* em geral (não específico ao *broker* utilizado), disponível no endereço web http://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE_Frequently_Asked_Questions_(FAQ). Adiconalmente, existem *mailing lists* específicas para vários problemas, como por exemplo um de ajuda a problemas técnicos com algum dos *Generic Enablers*, disponível no seguinte *email*: *fiware-tech-help@lists.fiware.org*, que foi inclusive utilizado para esclarecer algumas dúvidas nos resultados obtidos.

No *M2M*, não existe nenhuma comunidade no sítio *web Stack Overflow*. Foi utilizada uma biblioteca que implementa este standard, que é fechada e que pertence à *Altice Labs*, não estando disponível para o público em geral na *web*.

## 4.5 Viabilidade de Cada Tecnologia em Cada Cenário

O *FIWARE* impõe limites no tamanho máximo dos pedidos de publicação, 1 *MB*, e no tamanho da notificação, 8 *MB*. Como são recolhidos milhares de dados através dos vários sensores espalhados pela área urbana da cidade do Porto — sendo que no caso de uso abordado são cerca de 19884 *edges* do *OpenStreetMap*, cada um destes com seis atributos – estes acarretaram problemas devido às limitações já referidas. Estes problemas obrigaram a um *workaround* para os cenários de *publish/subscribe*. Como tal, foi feito o mapeamento de um recurso para cada *edge*, ao invés de estar a informação de todos os *edges* num único recurso, visto que se estivessem todos os *edges* apenas num recurso seria excedido tanto o tamanho máximo dos pedidos (visto que são 19884 *edges* a publicar) como o tamanho máximo de uma notificação. Podiam ser feitas várias publicações para o mesmo recurso, todas estas inferiores a 1 *MB*, para contornar este limite. No entanto, se toda a informação estivesse num único recurso, não seria possível efetuar *filter queries* (*queries* a recursos, nomeadamente pesquisar em quais é que a velocidade é menor que um dado valor) e também *geo queries*, visto que cada *edge*, e os seus atributos, teriam que ser um atributo do recurso, nomeadamente nesta forma:

```
{
  "vel_edge1": {
      "value": x
    },
    "vel_edge2": {
      "value": x
    }
}
```

Nesta forma não é possível pesquisar por exemplo quais os *edges* que têm uma velocidade média superior a x, visto que não é possível aplicar expressões regulares para escolher quais os atributos do recurso pelos quais pesquisar nas *queries*. Para poder ser pesquisável, o atributo tem que estar na forma de:

```
"vel": {
  "value": x
}
```

sendo que cada um dos outros recursos terão o mesmo atributo (com o mesmo nome) em todos eles, possibilitando depois pesquisar por recursos cuja velocidade seja maior do que x. Como são usados vários recursos, os nomes necessitam de estar nos mesmos moldes,nomeadamente *average_speed_edgeId*, em que <*edgeId*> varia de acordo com o valor do *edge* no *OpenStreetMap*. Desta forma, é possível agrega-los todos nas *queries* através de expressões regulares, por exemplo *average_speed_.**, que permite pesquisar em todos os recursos cujo *id* comece por *"average_speed_"*.

Devido a esta limitação, no caso do *M2M* também foi utilizado este tipo de mapeamento para manter a paridade nos resultados entre estes dois *middlewares*.

O *broker* de *M2M* utilizado não permite apagar subscrições, o que constitui um problema em casos que seja necessário apagar subscrições. Isto é apenas um problema do *broker*, visto que standard da *ETSI* do *M2M* define e permite apagar subscrições.

Em termos do cenário de *request/response*, o *FIWARE* é o único que permite *geo queries*. É também necessário implementar as *queries* (filtro dos resultados de acordo com a *query)* de base no *M2M*, ao estilo do *REST*, enquanto que o *FIWARE* já dispõe de operações que fazem o filtro dos resultados automaticamente.

Em relação ao cenário de disponibilização de um *dataset*, qualquer uma destas tecnologias é indicada, não havendo qualquer restrição, a não ser que no caso do *FIWARE* a criação do *dataset* terá que ser faseada se o *dataset* todo apenas num recurso, de modo a não exceder o tamanho máximo do pedido de publicação. Deste modo, em cada fase apenas pode ser publicado no máximo um *JSON* com perto de 1 *MB* de tamanho, ou seja, um pedido com um *Content-Length* inferior ou igual a 1 *MB*.

# Capítulo 5

# Implementação e Desenvolvimento

Neste capítulo é descrito como funcionam os vários componentes da *tool box* desenvolvida. Estes referem-se ao trabalho desenvolvido na base de dados, aos componentes que intervêm nos cenários descritos no capítulo 3 (*publisher* e *subscriber* nos dois primeiros, *requester* no terceiro), à recolha de métricas e ao funcionamento do processo de compilação das métricas recolhidas em ficheiros compactos.

## 5.1 Arquitetura da Tool Box

Os vários componentes que integram a *tool box* estão ilustrados na figura 5.1. Em seguida é feita uma introdução a cada um deles:

- *Database*, a base de dados relacional onde são guardados os dados recolhidos pelos sensores, implementada em *PostgreSQL*[1];
- *Publisher/Requester*, nos dois primeiros cenários este refere-se ao *publisher* e no terceiro ao *requester*. Estes criam ficheiros *CSV (Comma Separated Values)* com as várias métricas ao nível da aplicação referentes a cada cenário após a execução da publicação dos dados nos dois primeiros cenários, ou obtenção dos resultados conforme o pedido *web* efetuado no terceiro cenário;
- *Subscriber*, que apenas participa nos dois primeiros cenários, criando um ficheiro *CSV* com a respetiva métrica ao nível da aplicação;
- *Tcpdump*, referente ao processo da ferramenta *tcpdump*[2] que está a executar em *background* durante a publicação ou obtenção de dados;
- *PcapAnalyser*, uma aplicação *Java* que analisa as capturas da rede fornecidas pelo componente anterior, gerando no fim ficheiros *CSV* com as várias métricas ao nível da rede;

[1] https://www.postgresql.org/
[2] http://www.tcpdump.org/

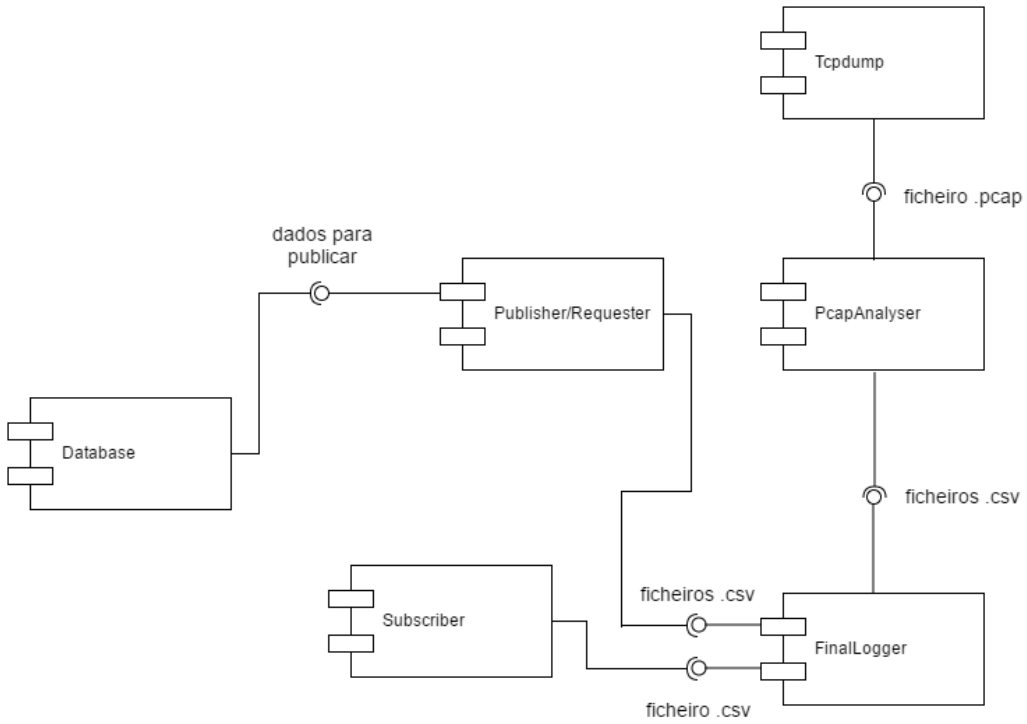



Figura 5.1: Diagrama *UML* de componentes descrevendo a arquitetura da *tool box* desenvolvida

.

- *FinalLogger*, uma aplicação *Java* que junta todos os ficheiros *CSV* criados, produzindo os ficheiros finais a partir dos quais é possível utilizar qualquer ferramenta para a interpretação dos dados gerados, como o *R*.

Nas secções seguintes são apresentados os vários componentes aqui falados mais em detalhe.

## 5.2 Base de Dados

O projeto *Future Cities Project* tem uma base de dados relacional em *PostgreSQL* e foi criada um réplica da base de dados para ser possível trabalhar sobre ela sem afetar a que está *live*. Nesta foram desenvolvidos *scripts* em *PL/pgSQL*[3] que permitem obter a velocidade média em todos *edges* do *OpenStreetMap* na área urbana do Porto tanto na última hora como nas últimas x semanas , sendo possível escolher o número de semanas desejado. Estas velocidades médias ficam armazenadas em quatro tabelas, uma para cada *timeslot* (das 8 às 10 da manhã, das 10 da manhã às 17 da tarde, das 17 da tarde às 19 da tarde e por fim das 19 da tarde às 8 da mnhã) referido no capítulo 3 na secção 3.2.3, com os seguintes campos:

- *way_id*, a chave primária da tabela e o *id* do *edge* do *OpenStreetMap*;
- latitude do *edge*;

[3] https://www.postgresql.org/docs/9.3/static/plpgsql.html

- longitude do *edge*;
- *total_speed*, valor total da soma de todas as leituras de velocidade registadas num dado *edge*;
- *n_values_speed*, número de leituras de velocidade registadas num dado *edge*;
- *weekly_average*, velocidade média registada nas últimas x semanas (por exemplo as 3 últimas semanas), calculada através da divisão entre a coluna *total_speed* e *n_values_speed*;
- *last_hour_average*, a velocidade média registada na última hora;
- *combined_average*, combinação das duas velocidades anteriores, com um peso arbitrário para cada uma.

Nas *queries* efetuadas à base de dados foram utilizados *cursors*[4], um mecanismo com o qual é possível realizar a *query* apenas a umas linhas da tabela em vez de a todas. Isto permite uma forma eficiente de ler milhares de linhas de uma tabela, evitando problemas de memória ao ler grandes quantidades de dados (*memory overrun*). Adicionalmente, também é significativamente mais rápido, visto que após o primeiro *fetch* a uma parte dos dados da tabela, os outros são quase instantâneos, na ordem dos 40 milissegundos, segundo foi observado através das *queries* efetuadas. Neste caso, as duas tabelas que eram usadas para calcular as velocidades médias tinham 12919900 e 6775280 linhas respetivamente. Um *fetch* de 1000 linhas demorava cerca de 45 segundos, e os *fetchs* seguintes (também de 1000 linhas) demoravam na ordem dos 40 milissegundos, reduzindo desta forma significativamente o tempo da *query* quando existem milhões de dados.

## 5.3 Publisher

### 5.3.1 Implementação Usando o Middleware ETSI M2M

Em termos da implementação do *publisher* usando este *middleware*, foi utilizada a biblioteca para a linguagem de programação *Java* já referida anteriormente no capítulo 4.

Os dados utilizados são obtidos a partir de uma base de dados do projeto *Future Cities Project*, já descrita acima, sendo que a conexão à mesma é feita utilizando a biblioteca disponibilizada pelo *PostgreSQL*.

No cenário de *publish/subscribe* com vários pedidos em simultâneo, é utilizada a classe *ExecutorService* em conjunto com o método *newFixedThreadPool*[5], que permite criar uma *pool* (neste caso uma *FixedThreadPool*) com o número de *threads* desejado. Se forem adicionadas mais *threads* do que o limite definido, estas são adicionadas à sua *queue* interna, e quando uma completar, a que estiver no topo da *queue* toma o seu lugar. Traduzindo para o problema em questão, é utilizada uma classe derivada da classe *Thread* do *Java*, que publica os dados sobre um *edge* num recurso. Estas *threads* são criadas ao iterar sobre os resultados vindo da base de dados, isto é, os

[4] https://www.postgresql.org/docs/9.2/static/plpgsql-cursors.html
[5] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html

dados a publicar. É criado o objeto *JSON* para publicar e criada a *thread* para publicar este, sendo prontamente adicionada à *FixedThreadPool*, e esta depois trata do funcionamento das *threads*. No caso as publicações falharem, devido ao *broker* estar sobrecarregado nesse momento, estas são inseridas num *array* para serem no fim publicadas de novo, e só quando todos os dados forem publicados, é que este processo termina realmente. Este processo está ilustrado na figura 5.2.

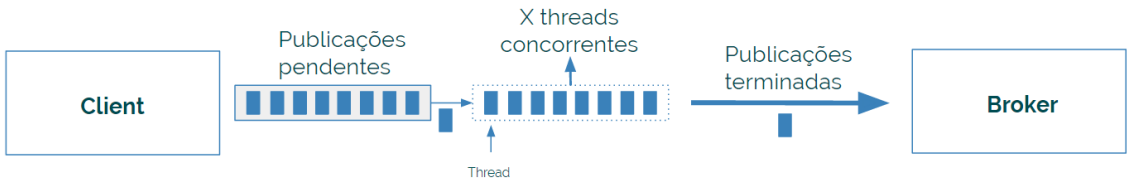



Figura 5.2: Diagrama explicando como funcionam os vários pedidos em simultâneo

.

O *JSON* publicado tem a seguinte estrutura:

```
{
  "edgeId": "x",
    "lat": "y",
    "lon": "z",
    "weekly_average": "w",
    "last_hour_average": "a",
    "combined_average": "b"
}
```

Para cenário de *publish/subscribe* em que são feitos pedidos sequenciais, os pedidos são feitos de forma *back-to-back*, ou seja, só é feito o próximo pedido quando anterior terminar, conforme o programa itera nos resultados retornados pela base de dados. Este processo pode ser verificado na figura 5.3.


Figura 5.3: Diagrama explicando como funcionam os vários pedidos em simultâneo

.

### 5.3.2 Implementação Usando o Middleware FIWARE

Para implementar o *publisher* com este *middleware* foi usada a linguagem de programação *Node.js*[6]. Esta utiliza o *Chrome V8 Javascript engine*, um motor de *Javascript* feito em C++, usado no *browser open-source* da *Google*, o *Google Chrome*. O *Node.js* é uma linguagem assíncrona com um modelo *I/O non-blocking* e é baseado em eventos, isto é, são utilizados *callbacks* nas várias operações nos quais o resultado destas são retornados quando terminam.

As operações disponibilizadas pelo *Orion Context Broker* do *FIWARE* são todas *RESTful* (pedidos *HTTP*), como tal foi utilizada a biblioteca *Unirest*[7], uma biblioteca leve para fazer pedidos *HTTP*, de modo a obter o mínimo de *overhead* ao fazer os pedidos.

Os dados utilizados são obtidos a partir de uma base de dados do projeto *Future Cities Project*, já descrita acima, sendo que a conexão à mesma é feita utilizando as bibliotecas *pg*[8] e *pg-transaction*[9], que permite transações.

Em termos do cenário de *publish/subscribe* em que são feitos vários pedidos em simultâneo, dado que o *Node.js* é assíncrono e não possui *threads* ao contrário do *Java* nem um sistema de *threads* como a classe do *Java FixedThreadPool*, foi necessário implementar as funcionalidades que esta classe providencia. Para emular este comportamento (de modo a manter a paridade entre as duas implementações, *Java* e *Node.js*, e por conseguinte nos resultados), todos os recursos a serem publicados partilham o mesmo *callback* e no início são publicados o número de recursos referente ao número de pedidos em paralelo (50...500). Quando uma publicação termina, dá origem a uma nova publicação, referente ao recurso que está no índice seguinte ao índice atual. Isto é, se forem publicados 500 recursos em simultâneo, o índice começa em 499, logo quando a primeira publicação terminar é publicado o imediamente a seguir, no índice 499+1 = 500. Este processo continua até todos os dados estarem publicados. No caso de as publicações falharem, devido ao *broker* estar sobrecarregado nesse momento, estas são inseridas num *array* para serem no fim publicadas de novo, e só quando todos os recursos forem publicados, é que este processo termina realmente, à semelhança do que acontece na implementação de *M2M* já referida acima. Uma representação gráfica deste processo está disponível na imagem número 5.2.

O *JSON* publicado tem a seguinte estrutura:

```
{
  "id" : "avg_speed_<edgeId>",
    "type" : "averageSpeedEdge",
    "edge_id": {
      "value": x
    },
    "weekly_average": {
        "value": y
    },
```

[6] https://nodejs.org/en/
[7] http://unirest.io/nodejs.html
[8] https://www.npmjs.com/package/pg
[9] https://www.npmjs.com/package/pg-transaction

```
10      "last_hour_average": {
11          "value": w
12      },
13      "combined_average": {
14          "value": z
15      },
16      "location": {
17        "value": "a,b",
18          "type": "geo:point",
19          "metadata": {
20            "crs": {
21                "value": "WGS84"
22              }
23          }
24    }
25  }
```

Como é possível observar, o *FIWARE* impõe uma estrutura definida para os dados a publicar, traduzindo-se num *overhead* considerável. Este advém nomeadamente do facto dos atributos necessitarem de um objeto interior contendo o valor do mesmo, e devido à descrição das variáveis geográficas.

No cenário de *publish/subscribe* no qual são feitos pedidos sequenciais, a publicação do próximo recurso apenas começa quando o anterior terminar. Foi utilizada a biblioteca *async*[10] para simplificar este processo, visto que esta permite executar operações síncronas numa linguagem que é inerentemente assíncrona. Esta disponibiliza uma série de funções, neste caso foi utilizada a função *eachSeries*, que executa a operação que lhe é fornecida de forma sequencial e síncrona. Este processo está ilustrado na figura 5.3.

### 5.3.3 Processo de Recolha de Métricas

Em termos das métricas recolhidas por este componente, estas estão ao nível da aplicação (são recolhidas a partir da própria implementação do código) e são criados cinco ficheiros com os seguintes dados:

1. *epoch time* do *Unix* (o número de segundos que passaram desde 1 de Janeiro de 1970 às 0:00h) no instante em que o pedido da publicação é feito. Esta métrica é importante na medida em que é utilizada para calcular o tempo de subscrição;

2. tempo de publicação;

3. tamanho em *bytes* ocupado pelas variáveis, referentes aos atributos do recurso a publicar. No caso do *Javascript*, foi utilizada a biblioteca *object-sizeof*[11]. Em *Java* foi calculado através do campo *BYTES*, que apenas existe em *Java 8*, que permite calcular o tamanho em *bytes* de uma variável de um tipo de dados primitivo (*int*,*float*,*double*,entre outros);

[10] https://www.npmjs.com/package/async
[11] https://www.npmjs.com/package/object-sizeof

4. número de *retries* ao nível da aplicação, referentes às ocasiões que a publicação falha, nomeadamente por motivos relacionados com o *broker*, como por exemplo o envio de respostas com *status* do protocolo *HTTP* na gama de 500.

5. tempo total de todas as publicações.

Para recolher estas métricas, as três primeiras são adicionadas a um *array*. No fim da publicação de todos os recursos estas são escritas para o respetivo ficheiro. Foi escolhido este método de adicionar o valor ao *array* ao invés de o escrever diretamente no ficheiro. Esta decisão foi efetuada de modo a melhorar a eficiência do mesmo, de modo a não escrever no ficheiro a cada publicação de um recurso, adicionando desta forma um *overhead* considerável. Esta escolha também foi feita por motivos de concorrência, pois existem situações em são publicados muitos recursos ao mesmo tempo, e visto que tanto a primeira e a terceira métrica são calculadas antes da publicação, os acessos aos ficheiros teriam que ser sincronizados, aumentando ainda mais este *overhead*. As últimas duas métricas não seguem este método visto apenas ser necessário escrever 1 valor no respetivo ficheiro.

No caso do *M2M* é criado um ficheiro adicional com o *Content-Length*, visto que este utiliza o protocolo *TLS* e não é possível obtê-lo através da análise das capturas de rede.

Todos estes ficheiros estão na forma de um *CSV* (*comma separated values*), sendo que os três primeiros e o de *Content-Length* do *M2M* estão na forma de <nome da entidade>,<valor>, e os restantes não, dado que apenas é escrito o valor registado.

## 5.4 Subscriber

### 5.4.1 Implementação Usando o Middleware M2M

O *subscriber* executa numa aplicação e porta diferente do *publisher*, dado que no *subscriber* apenas é necessário receber as notificações e visto que numa situação real o *publisher* dos dados não os vai subscrever. Para fazer o *parse* das notificações das subscrições que são recebidas, é criada uma *thread* por cada uma delas. Isto é feito de modo a obter um desempenho ótimo, para que no caso de chegarem muitas ao mesmo tempo seja possível fazer o *parse* correto de todas. É usado o mesmo mecanismo que na publicação dos dados, uma *FixedThreadPool*, para que no caso de chegarem muitas notificações ao mesmo tempo, não estarem demasiadas *threads* ativas, para obter eficiência em termos da utilização de memória e *cpu*.

### 5.4.2 Implementação Usando o Middleware FIWARE

O *subscriber* executa numa aplicação e porta diferente do *publisher*, pelas mesmas razões já apresentadas na secção anterior. Como tal a aplicação do *subscriber* é muito simples (um único *script* de *Node.js* com 29 linhas) e apenas com o essencial para realizar estas duas tarefas, para diminuir o *overhead* e obter o desempenho máximo.

### 5.4.3 Processo de Recolha de Métricas

A sexta métrica ao nível da aplicação é recolhida neste componente, e refere-se ao *epoch time* do recebimento de cada notificação relativa a cada publicação. Esta métrica é recolhida anotando este *epoch time* e adicionando-o a um *array*, para depois no fim, após terem sido publicados todos os recursos, os dados contidos neste *array* serem escritos para um ficheiro *CSV*. Foi escolhido este método de adicionar o valor ao *array* ao invés de o escrever diretamente no ficheiro por duas razões:

- para melhorar a eficiência do mesmo, dado não ser necessário escrever no ficheiro a cada notificação, já referido na secção 5.3.3;
- por motivos de concorrência, pois existem situações em que chegam várias notificações ao mesmo tempo.

## 5.5 Requester

### 5.5.1 Implementação Usando o Middleware M2M

Na implementação deste componente com este *middleware* foi utilizado a *framework Jetty*[12] que permite nomeadamente criar servidores *web* de uma forma simples. Ao mesmo tempo, é executado a aplicação de *M2M*, visto ser necessário para poder fazer as operações necessárias: *get* de recursos e as *content instances* dentro de estes neste caso. É possível fazer *queries* por *edge*, por qualquer uma dos tipos de velocidades (e dentro destas qualquer um dos operadores de comparação: >, >=, <, <=, =), sendo possível combinar estas, e pesquisar por qualquer um dos *timeslots*. Estas *queries* tiveram de ser implementadas de base, visto a biblioteca utilizada não oferecer nenhuma função de filtro ou parecido, ao contrário do *FIWARE*.

### 5.5.2 Implementação Usando o Middleware FIWARE

Foi utilizado a *framework* para o desenvolvimento de aplicações *web* e *APIs Express*[13] de modo a facilitar o desenvolvimento deste componente.

É possível fazer as seguintes operações:

- *queries* por *edge*, na mesma forma como foram descritas na secção anterior do *M2M*. Esta *query* é realizada utilizando *string queries* combinadas com a operação standard *queryContext*;
- *queries* geográficas, que são mutuamente exclusivas com as referidas acima. É possível indicar se se pretende obter resultados dentro da área do circulo/polígono ou fora dela e qual o polígono desejado para mapear a área desejada para fazer a *query*: quadrado, triângulo, entre outros, indicando os seus vértices. No caso do círculo é necessário também indicar

[12] http://www.eclipse.org/jetty/
[13] http://expressjs.com/

qual o raio do mesmo, a latitude e longitude do centro. Esta *query* é realizada usando *geo queries* combinadas com a operação standard *queryContext*.

### 5.5.3 Implementação Usando REST

Nesta tecnologia, os dados são obtidos a partir da base de dados já referida. As *queries* que são possíveis fazer são as que já foram descritas na secção do *M2M*, com a diferença que os filtros são implementados via *queries* à base de dados.

### 5.5.4 Processo de Recolha de Métricas

Neste componente é criado um ficheiro *CSV* com os tempos de resposta a cada pedido *web* efetuado.

## 5.6 Tcpdump

Para as medições ao nível da rede, são capturados *logs* através da ferramenta *tcpdump*[14], uma ferramenta que permite capturar o tráfego da rede através da linha de comandos. Esta ferramenta está a correr em *background* durante a publicação de dados ou a obtenção dos resultados de um pedido *web*, de modo a capturar o tráfego da rede nestes momentos. Para tal, foram utilizados os filtros disponibilizados por esta ferramenta. Estes foram utilizados para filtrar o tráfego pela porta relevante em cada um dos cenários. Nos dois primeiros cenários apenas é capturado o tráfego da comunicação com o *broker* e no terceiro cenário apenas a comunicação com o utilizador que fez o pedido *web*. Estes *logs* são posteriormente interpretados pelo componente seguinte, o *PcapAnalyser*.

## 5.7 PcapAnalyser

Para esta aplicação, foi utilizada uma biblioteca denominada por *jNetPcap*[15], que permite interpretar capturas de rede (ficheiros *.pcap*). Utilizando esta, são criados 4 ficheiros com os seguintes dados:

1. tamanho do *Content-Length*, tamanho do *payload TCP* (*TLS* no caso do *M2M*), *round trip time*, total de *bytes* da comunicação em ambos os sentidos e *goodput* de cada publicação;
2. número de retransmissões de pacotes *TCP* e *HTTP* e o *delay* registado para a retransmissão em causa;
3. número de ocorrências da *flag RST* do protocolo *TCP*;
4. publicações onde se verificaram ocorrências da *flag RST* do protocolo *TCP*.

[14] http://www.tcpdump.org/
[15] http://jnetpcap.com/

Todos estes ficheiros também estão no formato *CSV*, sendo que no primeiro e no último os dados estão indexados por cada recurso. Para o terceiro cenário, apenas é criado o primeiro ficheiro, só com o *content-length* e o total de *bytes* utilizados, visto as outras métricas não serem relevantes neste.

No caso do *M2M*, foi necessário implementar o suporte para esta biblioteca reconhecer pacotes do protocolo *TLS/SSL* [The08] através de um *custom header*, visto que a biblioteca apenas suporta de raiz pacotes do protocolo *ethernet*, *ip*, *tcp* e *http*. Para tal, foi seguido o capítulo 5 do *user guide*[16] do *jNetPcap*, onde é explicado como criar um *custom header* para a biblioteca poder interpretar outros protocolos que não os suportados de origem. Para implementar o *custom header* é necessário perceber qual o tamanho do *header* do protocolo em questão, em que *bytes* desse *header* estão os vários campos do mesmo, e quais são os valores que esses campos podem tomar. No caso do *SSL/TLS* o tamanho do *TLS Record Header* são 5 *bytes* e são os seguintes:

- *byte* 0 refere-se ao tipo de pacote do *SSL/TLS*: *Change Cipher Spec* com o valor 20, *Alert* com o valor 21, *Handshake* com o valor 22 ou *Application Data* com o valor 23;
- *bytes* 1 e 2 referem-se à versão do protocolo, 0x300 significa que é *SSL3*, 0x301 significa que é *TLS 1.0* e 0x303 significa que é *TLS 1.2* (a versão do protocolo que era usada pela biblioteca de *M2M*);
- *bytes* 3 e 4 referem-se ao tamanho do *payload*, excluindo os 5 *bytes* que o *header* ocupa.

A partir deste *custom header*, é possível fazer *parse* de pacotes do protocolo *SSL/TLS*, obtendo todas as informações contidas no *TLS Record header*. Estas informações são as relevantes para este caso, visto apenas ser necessário distinguir que tipo de pacote é e o seu tamanho. Não é necessário descodificar os dados que vêm no *Handshake Protocol Header*, como o tipo de *handshake*, visto que:

- todo o *handshake* é feito aquando da criação da aplicação de *M2M*;
- quando é feita a publicação dos dados apenas existem 4 mensagens de *handshake*: *ClientHello*, *ServerHello* e duas *Encrypted Handshake Messages*

Para as mensagens *Encrypted Handshake Message* foi verificado através deste *custom header* qual é o tipo desta mensagem conforme o *Handshake Protocol Header* define:

- 0, referente à mensagem *Hello Request*;
- 1, referente à mensagem *Client Hello*;
- 2, referente à mensagem *Server Hello*;
- 11, referente à mensagem *Certificate*;
- 12, referente à mensagem *Server Key Exchange*;

[16] http://jnetpcap.com/?q=userguide/ch5

- 13, referente à mensagem *Certificate Request*;
- 14 referente à mensagem *Server Done*;
- 15 referente à mensagem *Certificate Verify*;
- 16 referente à mensagem *Client Key Exchange*;
- 20, referente à mensagem *Finished*.

Estas referem-se ao tipo de mensagem *Finished*, pois o valor observado foi 20. Os dados do *Alert Protocol* também não é necessário verificar, dado que apenas é enviado um antes do fecho de cada ligação, indicando o fecho da mesma (*Closure Alert*).

## 5.8 FinalLogger

Este componente destina-se apenas aos primeiros dois cenários, visto que este os dados estão contidos em vários ficheiros separados, além de ser necessário ler os ficheiros para calcular nomeadamente o tempo de subscrição. No terceiro cenário este componente não é necessário visto que toda a informação já se encontra bem discriminada e pronta para analisar. Isto é devido ao facto de todas as métricas deste cenário já estarem contidas nos ficheiros criados pelos componentes anteriores prontas para análise.

Para compilar a informação dos dois primeiros cenários e juntá-la, é utilizada mais uma aplicação *Java*. Existem duas classes essenciais nesta aplicação:

- *Logger*, que junta os ficheiros vindos do *publisher*, *subscriber* e do componente anterior num só. Além disto, é aqui também calculado o tempo de subscrição, através da diferença entre o *epoch time* do recebimento da notificação e do envio do pedido de publicação de um recurso. É criado um ficheiro *CSV* por cada ciclo de publicação (um ciclo de publicação é a publicação dos 19984 dados destes dois cenários). Este ficheiro tem a seguinte estrutura em cada linha: <Recurso>,<TPub>,<EpochPub>,<TSub>,<SensorReadings>,<Content-Length>,<TcpPayload>,<TotalLigCS>,<TotalLigSC>,<RTT>,Goodput>. Estes referem-se ao seguinte:
  - recurso em si;
  - tempo de publicação;
  - *epoch time* do envio do pedido *web* de publicação;
  - tempo de subscrição;
  - tamanho em *bytes* ocupado pelas variáveis relativas aos atributos de cada recurso a publicar;
  - *content-length* do *JSON* do recurso a publicar;
  - tamanho do *payload TCP* do pacote da publicação;

  - total de *bytes* da comunicação no sentido cliente->servidor;
  - total de *bytes* da comunicação no sentido servidor->cliente;
  - *round trip time*;
  - *goodput* da comunicação.

- *AverageLogger*, que junta todos os ficheiros dos ciclos de publicação num único ficheiro *CSV*. Faz a média de todos os valores encontrados em cada ciclo de publicação, além de adicionar neste ficheiro métricas que apenas têm um registo por cada ciclo. Estas referem-se ao tempo total de publicação, aos *retries* ao nível da aplicação e aos *retries* ao nível da rede: o número de *TCP RST flags*,número de retransmissões *TCP* e *HTTP*, e o *delay* da retransmissão. O ficheiro está indexado por cada ciclo de publicação.

No caso do *M2M*, os dados de um ciclo de publicação estão em dois ficheiros diferentes dado que as comunicações são encriptadas. Como tal, não é possível indexar os ficheiros produzidos pelo componente anterior por recurso, logo não é possível discriminar os dados de rede por cada um destes.

## 5.9 Workflow

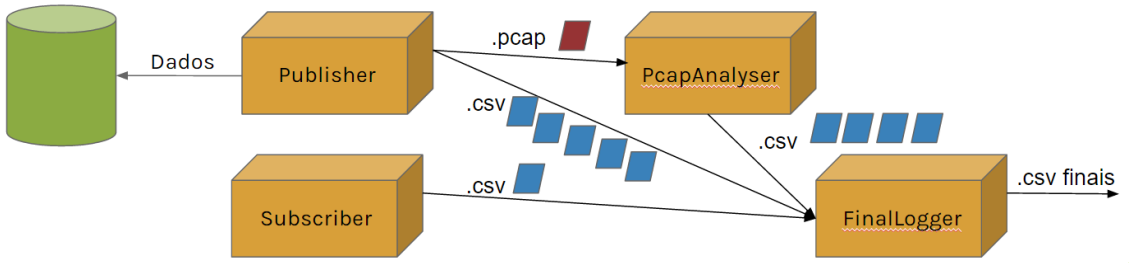



Figura 5.4: *Workflow* de toda as partes involvidas no processo de obtenção de métricas quantitativas para posterior análise e avaliação nos cenários de *publish/subscribe*

.

A figura acima ilustra o *workflow* desta *tool box* nos cenários de *publish/subscribe*. Descrevendo e resumindo o funcionamento deste, o *publisher* ao publicar cria 5 ficheiros e o *subscriber* 1, contendo as várias métricas ao nível da aplicação já referidos anteriormente.

Paralelamente, está a executar em *background* na máquina que está a publicar os dados uma captura de pacotes da rede com recurso à ferramenta *tcpdump*. Apenas são capturados os pacotes relativos à publicação, descartando os demais e os da subscrição, visto que tanto o *publisher* e o *subscriber* estão na mesma máquina. Isto é conseguido através dos filtros que esta ferramenta possui, filtrando os que não são enviados ou recebidos pela porta referente à publicação (as subscrições são recebidas numa porta diferente). Quando a publicação de todos os dados termina, a captura também é terminada, gerando um ficheiro *PCAP* contendo o tráfego da rede capturado.

O *subscriber* vai adicionando as várias notificações de subscrição recebidas a um *array*. Passados uns segundos após a publicação terminar, é enviado um pedido para o servidor *web* que o *subscriber* executa no caso do *FIWARE*, e um comando pela linha de comandos no caso do *M2M*. Este pedido/comando serve para escrever os dados contidos neste *array* para um ficheiro *CSV*, cujo conteúdo já foi referido acima. São esperados alguns segundos para terminar visto que quando o *publisher* termina ainda existem notificações de subscrição a receber. Estas podem demorar mais ou menos tempo a chegar dependendo das condições do *broker* (carga atual) e do estado da ligação de rede.

Estando este processo terminado, a aplicação *Java PcapAnalyser* que processa as capturas de rede entra em cena, processando todas as capturas (uma para cada ciclo de publicação), gerando os quatro ficheiros *CSV* já referidos.

Por fim, estando estes passos todos concluídos é utilizada a aplicação de *Java FinalLogger* cuja função já foi explicada na secção 5.8.

Após estes passos, estão criadas todas as condições para utilizar qualquer ferramenta para a interpretação dos dados gerados, como a linguagem de programação *R*[17], ou mesmo importar os ficheiros para o *Microsoft Excel*.

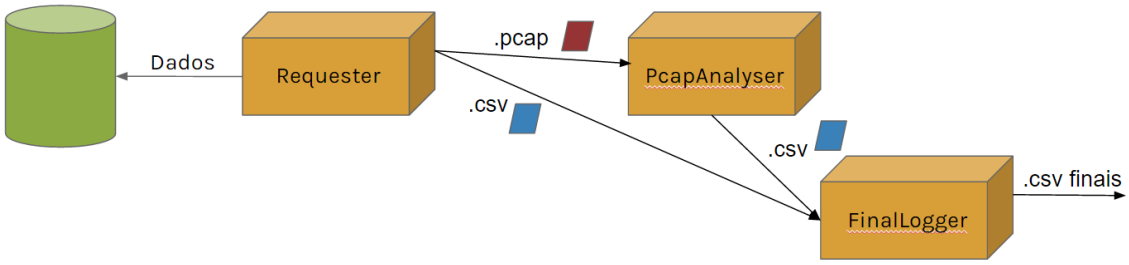



Figura 5.5: *Workflow* de todas as partes involvidas no processo de obtenção de métricas quantitativas para posterior análise e avaliação nos cenários de *request/response*

.

O *workflow* no cenário de *request/response*, como ilustrado na imagem acima, é bastante similar ao de *publish/subscribe*. Apenas difere no facto de apenas ser gerado um ficheiro *CSV* com os tempos de resposta, e no processamento da captura de rede onde são verificados menos parâmetros, sendo apenas criado um ficheiro, com o total de *bytes* para a comunicação, o *Content-Length* e o *goodput*.

Para finalizar, de notar também que esta *tool box* pode ser estendida a novos *middlewares*, adicionando um novo módulo para este, ou a *brokers* diferentes dos *middlewares* já suportados, sendo apenas necessário mudar o ponto de contacto destes.

[17] https://www.r-project.org/

# Capítulo 6

# Resultados

Neste capítulo são descritas as várias experiências realizadas para cada cenário e são analisados os resultados obtidos, interpretando-os e colocando hipóteses para as suas causas, com base nas várias métricas anteriormente definidas no capítulo 3.

## 6.1 Experiências

De modo a obter resultados para a avaliação quantitativa, foram efetuadas experiências para cada um dos *middlewares* em cada um dos cenários.

Para as experiências foi utilizada uma máquina virtual da *DigitalOcean*[1] localizada no *datacenter* de Londres com as seguintes especificações:

- Sistema operativo Ubuntu 14.04 x64;
- 2 cores do *CPU Intel(R) Xeon(R) CPU E5-2650L v3 @ 1.80GHz*;
- 2 *GB* de *RAM*;
- 889.67 *Mbits/s* de *download* e 397.00 *Mbit/s* de *upload*, sendo que esta é partilhada pelas várias máquinas virtuais existentes na máquina física.

### 6.1.1 Cenários Publish/Subscribe

No caso com vários pedidos em simultâneo, foram realizadas medições da parte da manhã, tarde e noite. Em cada uma destas partes foram feitas 3 repetições — cada repetição representa um conjunto de ciclos de publicação, isto é, 10 ciclos de publicação — da mesma medição. Na parte da manhã foram efetuadas das 9h30m às 12h30m, da tarde das 15h30m às 18h30 e por fim da noite das 21h30m às 00h30m. No caso do *M2M*, devido ao tempo total de publicação ser deveras grande, apenas foi efetuada 1 repetição ao longo do dia, ou seja, apenas um conjunto de ciclos de publicação (10 ciclos de publicação), pois não foi possível encaixar mais que 4 ciclos de publicação por cada parte do dia.

[1] https://www.digitalocean.com/

Adicionalmente, foi variado o número de publicações concorrentes, de 50 a 500 — como já referido na secção 5.3 do capítulo 5 — com incrementos de 50. O teto máximo escolhido foi 500 devido aos *brokers* não suportarem mais publicações concorrentes. Esta variação do número de pedidos concorrentes possibilita observar as diferenças de comportamento conforme o número de publicações em simultâneo vai aumentando. Estas referem-se nomeadamente ao aumento do número de *retries*, do tempo de publicação/subscrição, do número de retransmissões, entre outros. O número total de dados a publicar foi 19884.

No caso sequencial, foram efetuados 4 ciclos de publicações *back-to-back*, visto que estes eram bem mais morosos e não era possível escalonar mais que 2 por cada parte do dia e também dado que a variância neste resultados é menor porque que o método é sempre o mesmo.

O *publisher* e o *subscriber* estavam na mesma máquina, já referida anteriormente. Como tal, não foi necessário sincronizar o *publisher* e *subscriber*, nomeadamente com recurso protocolo de sincronização de relógios *NTP*[2], que permite sincronizar o relógio entre máquinas distintas em relação a um relógio comum - como o *UTC* - com um erro de poucos milissegundos.

### 6.1.2 Cenário Request/Response

Neste cenário foram efetuados cerca de 50 pedidos *back-to-back* para obter as métricas já referidas no capítulo 3. Foram efetuadas *filter queries* convencionais incidindo sobre a velocidade média em ruas da cidade do Porto (sendo que esta *query* retornava 4802 resultados de 19884), o único tipo de *query* que todos os *middlewares* suportam, sendo que foi feita a mesma em cada uma deles.

A tecnologia *REST* serve como o caso base para a comparação, dado que esta é a tecnologia inerente para fazer sistemas *request/response*. O objetivo é verificar o desempenho do *FIWARE* e *M2M* neste cenário, fazer uma comparação entre os dois, e depois comparar com o caso base, verificando como é o seu desempenho face a este.

## 6.2 Resultados

### 6.2.1 Cenário Publish/Subscribe com Vários Pedidos em Simultâneo

#### 6.2.1.1 FIWARE

O *broker* utilizado é orientado para testes e disponibilizado de forma pública, desde que se possua uma conta criada no síto *web* do *FIWARE*. Foi escolhido devido a se assemelhar a um *broker* em produção, visto que não havia nenhum deste tipo disponível. Encontra-se em Barcelona e não implementava *HTTPS*, logo todos os dados foram transmitidos sem encriptação.

Foram feitas medições ao longo de um dia neste *broker* e em seguida é apresentado um curto resumo do que foi observado, sendo que posteriormente estes resultados são analisados mais em

[2] http://www.ntp.org/

detalhe. Através das medições ao nível da aplicação, já referidas no capítulo anterior, é possível verificar o seguinte:

- a notificação relativa à subscrição de um dado recurso tanto é enviada antes como após o *publisher* receber o OK da publicação desse recurso, no entanto, normalmente é recebida posteriormente ao OK;
- conforme o número de publicações concorrentes aumenta, o número de *retries* aumenta, o tempo total aumenta e por consequência o *tPub* e *tSub* também;
- o número de *retries* também vai aumentando em geral ao longo do dia. Estes correspondem ao envio de respostas com *status* do protocolo *HTTP* 503, que significa que o serviço estava indisponível de momento.

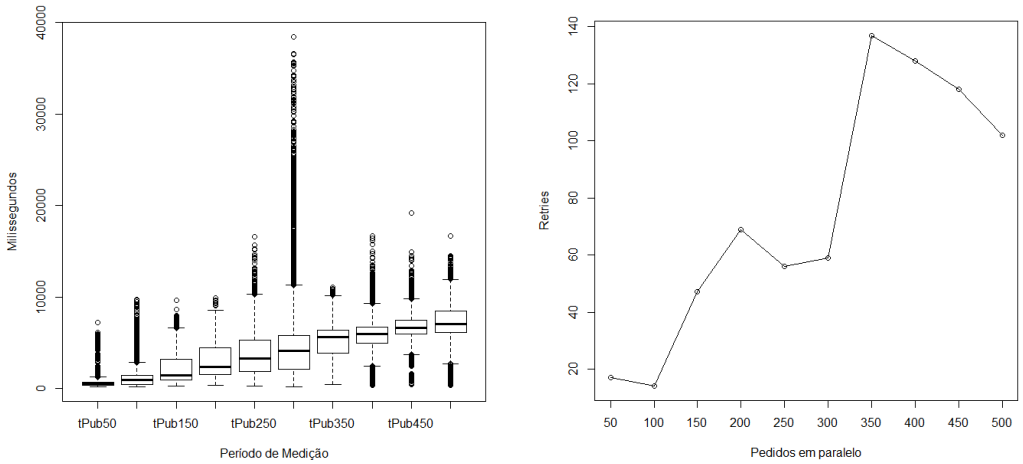
Através das medições ao nível da rede, também já referidas no capítulo anterior, é possível verificar o seguinte:

- o *content-length* tem 251 bytes de tamanho em média, sendo desta forma 348% superior ao tamanho ocupado pelas variáveis relativas aos dados a publicar;
- o tamanho do *payload TCP* tem cerca de 511 *bytes*, um aumento de cerca de 100% em relação ao tamanho do *content-length*;
- São enviados cerca de 850 *bytes* em direção ao servidor, e são recebidos cerca de 689 *bytes* por publicação. A partir desta informação é possível concluir o seguinte:
  - o total de *bytes* recebidos no sentido servidor->cliente é 19% menor quando comparado ao sentido cliente->servidor;
  - o total de *bytes* de uma publicação (850 + 689) é cerca de 200% superior ao tamanho do *payload TCP* do pacote de publicação;
  - para publicar um *JSON* de 251 *bytes* de tamanho são transmitidos 1539 na camada de ligação, um aumento de cerca de 513%.
- foram verificadas retransmissões de pacotes *TCP* e *HTTP*;
- o *Round Trip Time* mantém-se idêntico ao longo do dia.

Para ajudar a compreender mais em detalhe os dados, são aqui apresentados um conjunto de *boxplots* e *lineplots*.

Na figura 6.1a é possível verificar o aumento do tempo de publicação no primeiro ciclo de publicações de manhã, de acordo com o aumento do número de publicações concorrentes. Este aumento deve-se ao facto do número mais elevado de publicações em paralelo colocar um esforço maior no *broker*. Cada *tick* no eixo dos xx representa um ciclo de publicação com um número de publicações concorrentes diferente. Também se reflete no aumento do número de *retries* ao nível

da aplicação, pois o *broker* conforme fica mais sobrecarregado envia mais respostas com *status* 503, como é visível na figura 6.1b.




(a) Evolução do tempo de publicação de acordo com o aumento do número de publicações em paralelo, na parte da manhã no primeiro ciclo de publicação

(b) *Retries* verificados no primeiro ciclo de publicações da parte da manhã

Figura 6.1: Tempos de publicação e *retries* verificados no primeiro ciclo de publicações de manha

Na figura 6.2 é possível verificar a variação do tempo de subscrição ao longo dos vários conjunto de ciclos de publicação ao longo do dia. Cada *tick* do eixo dos xx reflete um conjunto.

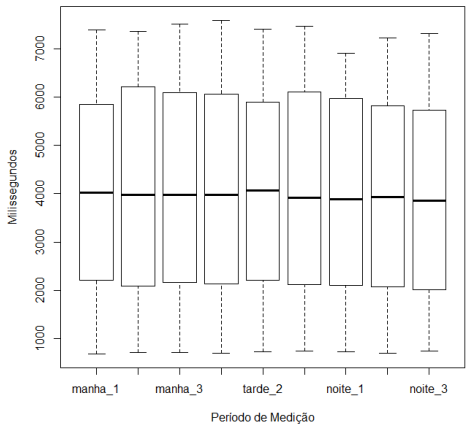



Figura 6.2: Tempo de subscrição ao longo dos vários conjuntos de ciclos de publucação, cada *tick* do eixo do x indica um destes conjuntos, desde manhã até à noite

.

Um dos objetivos era verificar se a notificação da subscrição chegava antes ou depois do OK da publicação, visto que se queria perceber se o *broker* enviava OK da publicação de um recurso

antes do envio da notificação relativa à subcrição desse recurso. Para tal, foi calculada a diferença entre o tempo de publicação e subscrição em cada publicação de um recurso. Se esta diferença for negativa, então significa que a notificação da subscrição de um dado recurso é enviada antes de ser enviado o OK relativo à publicação desse mesmo. Foi observado que tanto chega antes como chega depois do OK. Uma hipótese possível será o facto de existirem dois processos diferentes: um a tratar do *parse* da publicação e outro para o envio da subscrição. Esta hipótese explicaria esta situação, visto que se todo o processo fosse tratado no mesmo processo, as notificações chegariam sempre antes ou depois do OK do pedido da publicação. Esta diferença pode ser verificada na figura 6.3.

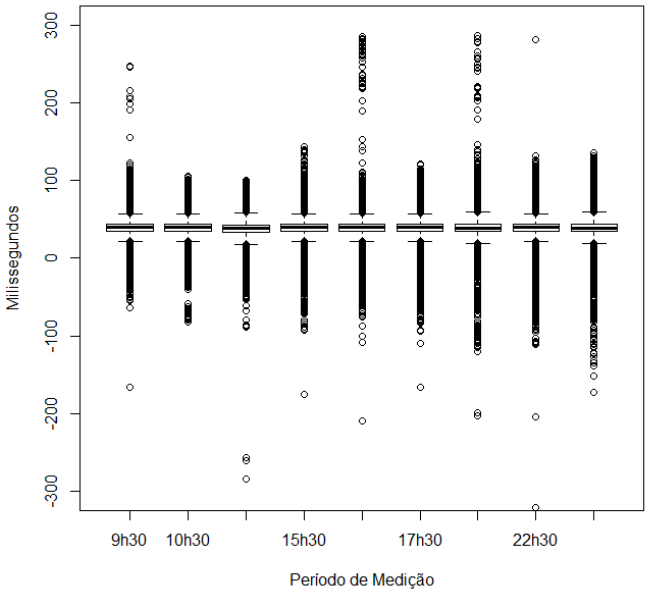
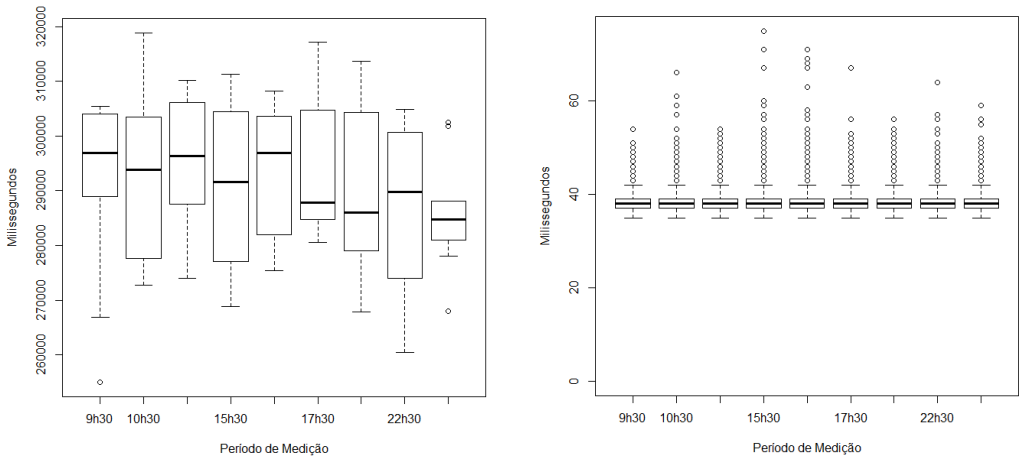



Figura 6.3: *Box plot* sobre a diferença entre o tempo de publicação e o de subscrição, sendo que cada *tick* do eixo do x representa o ciclo de publicação em que são publicados 50 pedidos em simultâneo, começando no primeiro da parte da manhã e terminando no último da parte da noite

.

Foi também possível verificar que o tempo total de um ciclo de publicação é de cerca de 4-5 minutos e tem tendência a decrescer ao longo dia, enquanto o *Round Trip Time* se mantém praticamente constante. Uma hipótese possível é que o esforço atual no *broker* é determinante para o tempo de publicação. Este comportamento está ilustrado nas figuras 6.4a e 6.4b.




(a) *Box plot* sobre a evolução do tempo total ao longo do dia, sendo que cada *tick* do eixo do x representa um ciclo de publicação, começando no primeiro da parte da manhã e terminando no último da parte da noite

(b) *Box plot* sobre a evolução do *Round Trip Time* ao longo do dia, sendo que cada *tick* do eixo do x representa um ciclo de publicação, começando no primeiro da parte da manhã e terminando no último da parte da noite

Figura 6.4: Evolução do tempo total de publicação e *round trip time* ao longo dos vários conjuntos de ciclos de publicação

Adicionalmente, em certos ciclos de publicação foram observadas retransmissões *TCP* incidindo sobre o primeiro *TCP SYN* e *HTTP* em relação ao pacote *HTTP* da publicação. O tempo para as retransmissões verificado foi cerca de 998 milisegundos para retransmissões sobre o pacote *TCP SYN* e 238 milisegundos para retransmissões sobre os pacotes *HTTP* das publicações. No entanto, em termos percentuais estas ocorrências são mínimas, flutuando entre 0% e 0.0007%, não sendo deste modo nada relevantes. A figura 6.6a retrata o número de retransmissões em termos de pacotes *TCP SYN* e a figura 6.6b o número de retransmissões em termos de pacotes *HTTP*.

Foi observado que o número de *retries* ao nível da aplicação foi crescendo ao longo do dia, como ilustrado na figura 6.5. Mesmo este número não sendo grande, situando-se entre os 0.02% e 0.014%, foi posta a hipótese que isto se poderá ter devido ao facto de haver algum *memory leak* no *broker*. Para confirmar esta hipótese, foram realizadas mais medições, ao longo de dois dias consecutivos, e se o número de *retries* fosse crescendo ao longo deste dois dias, esta hipótese seria validada.

De acordo com os dados recolhidos ao longo destes dois dias, foi possível verificar que se manteve a tendência para os *retries* aumentarem ao longo do dia, no entanto, entre um dia e o seguinte essa tendência de aumento não se verificou. Ou seja, os dias revelaram-se idênticos, isto é, começam mais ao menos no mesmo número de *retries*, baixo, aumentando ao longo do dia. Face a estes resultados, a hipótese mais provável continua a ser a existência de um *memory leak*. No entanto, esta hipótese implicaria que o *broker* estivesse a ser reiniciado de madrugada, no período entre a 00h30 e as 9h30, dado que não foram efetuadas medições nesse mesmo período de tempo. Um *box plot* detalhando a evolução dos *retries* ao longo dos dois dias encontra-se na figura 6.7.

Este contém 18 barras, sendo que a partir da décima estas referem-se ao segundo dia.

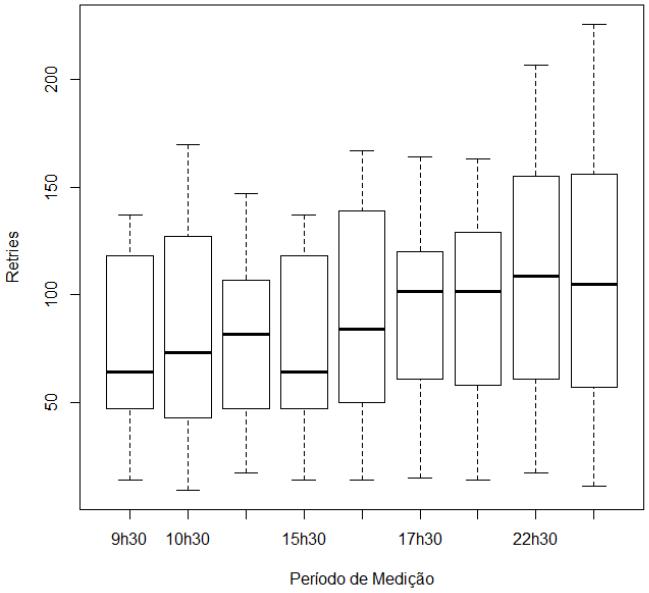
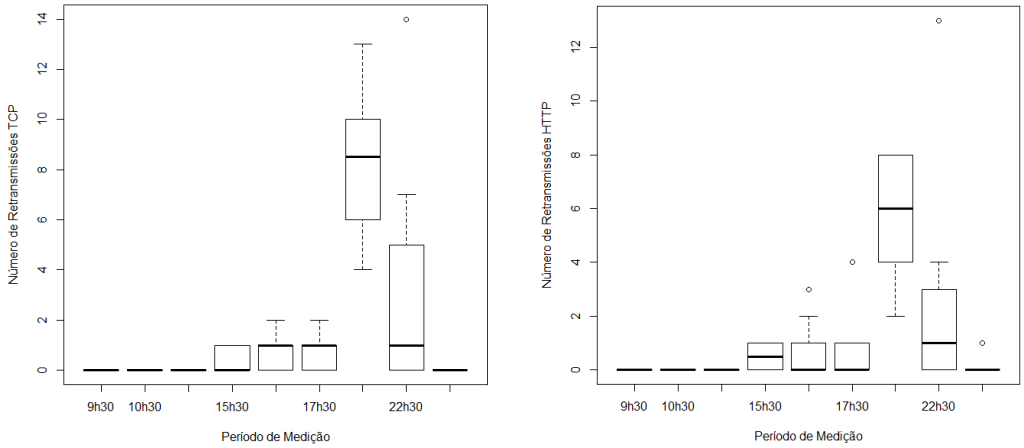



Figura 6.5: Retries ao longo do dia

.




(a) *Box plot* sobre a evolução das retransmissões de pacotes *TCP* ao longo do dia. Cada *tick* do eixo do x representa um ciclo de publicação, começando no primeiro da parte da manhã e terminando no último da parte da noite

(b) *Box plot* sobre a evolução das retransmissões de pacotes *HTTP* ao longo do dia. Cada *tick* do eixo do x representa um ciclo de publicação, começando no primeiro da parte da manhã e terminando no último da parte da noite

Figura 6.6: Evolução das retransmissões *TCP* e *HTTP* ao longo dos vários ciclos de publicação

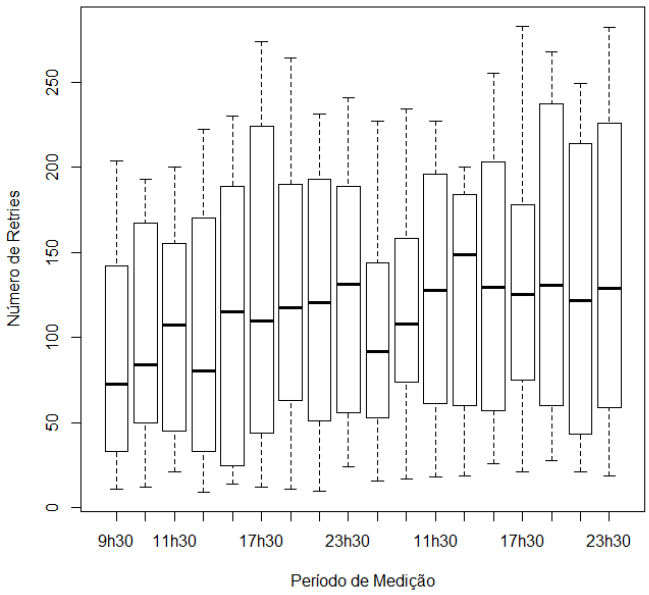


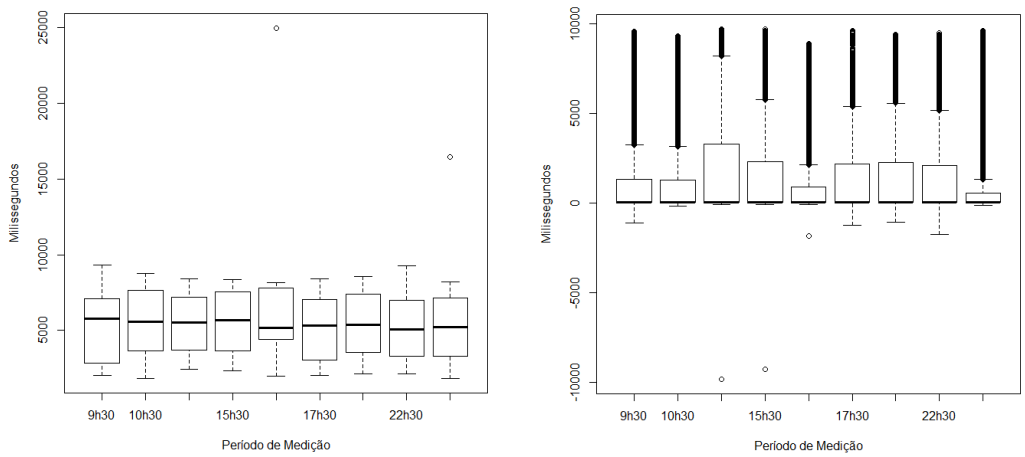

Figura 6.7: *Box plot* sobre a evolução dos *retries* ao longo de dois dias consecutivos, sendo que cada *tick* do eixo do x representa um ciclo de publicação. O segundo dia começa a partir da 10º caixa

.

Para confirmar esta hipótese foram contactados os responsáveis pelo desenvolvimento deste *middleware* via o *email* já referido no capítulo 4. Através deste contacto foi possível perceber que os *retries* ao nível da aplicação, as respostas com *status* 503 do protocolo *HTTP*, são provocados pelo *proxy* de *Steelskin PEP*, já referido no capítulo 2.

No entanto, foi afirmado por estes responsáveis que não tinham conhecimento de nenhuma existência de *memory leak*, de algum reinício do *broker* ou de algum mau funcionamento deste.

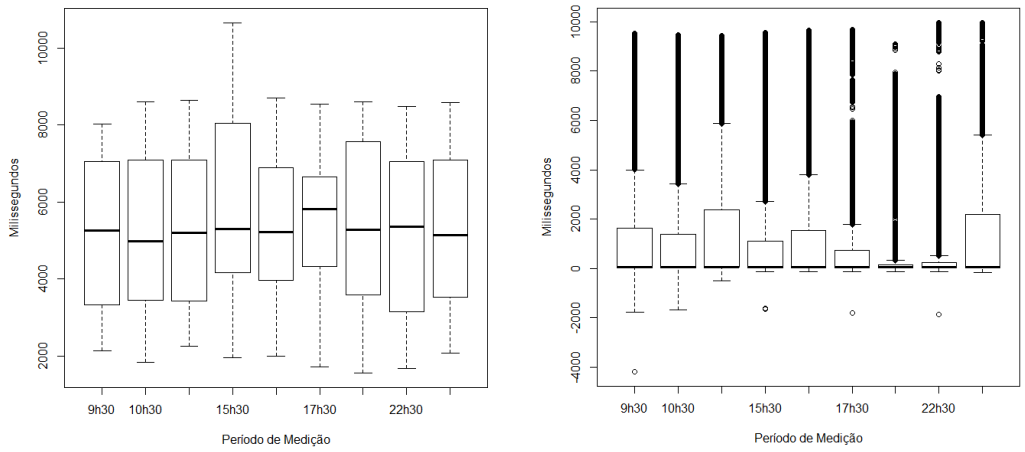
Estas medições adicionais também permitiram verificar que o tempo de subscrição aumentou consideravelmente em relação à medição anterior a estes dois dias. No entanto, o tempo de publicação manteve-se idêntico à primeira medição (medição anterior a esta medição feita durante dois dias consecutivos). Por consequência, a diferença entre o tempo de publicação e subscrição também aumentou. Como não existe nenhum controlo sobre este *broker*, não podendo obter informações de modo a perceber algum aspeto interno no funcionamento do *broker* foi alterado, não é possível afirmar se isto se deve a um aumento da carga no *broker* ou se houveram alterações no seu funcionamento. Estas alterações podem ser verificadas nas figuras 6.8a e 6.8b, que ilustram o tempo de subscrição e a diferença entre o tempo de publicação e subscrição no primeiro dia, respetivamente. Nas figuras 6.9a e 6.9b pode-se observar o tempo de subscrição e a diferença entre o tempo de publicação e subscrição no segundo dia, respetivamente.




(a) Tempo de subscrição ao longo do primeiro dia, cada *tick* do eixo do x indica um conjunto de ciclos de publicação, desde manhã até à noite

(b) Diferença entre o tempo de publicação e o de subscrição em relação ao primeiro dia. Cada *tick* do eixo do x representa o ciclo de publicação com 50 pedidos em simultâneo, começando no primeiro da parte da manhã e terminando no último da parte da noite

Figura 6.8: Evolução do tempo de subscrição e da diferença entre o tempo de publicação e subscrição ao longo dos ciclos de publicação no primeiro dos dois dias




(a) Tempo de subscrição ao longo do segundo dia, cada *tick* do eixo do x indica um conjunto de ciclos de publicação, desde manhã até à noite

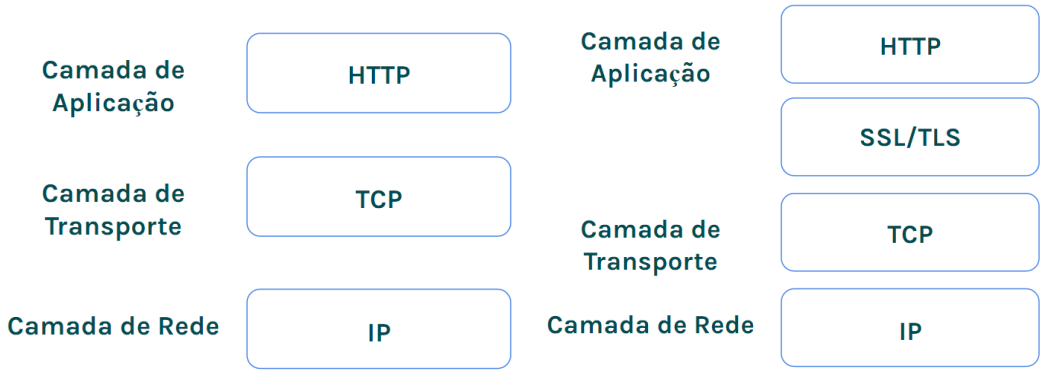
(b) Diferença entre o tempo de publicação e o de subscrição em relação ao segundo dia. Cada *tick* do eixo do x representa o ciclo de publicação com 50 pedidos em simultâneo, começando no primeiro da parte da manhã e terminando no último da parte da noite

Figura 6.9: Evolução do tempo de subscrição e da diferença entre o tempo de publicação e subscrição ao longo dos ciclos de publicação no segundo dos dois dias

Em termos da comparação dos tamanhos do *frame* de publicação e da resposta à mesma, os tamanhos são cerca de 577 e 417 *bytes* respetivamente. Esta diferença deve-se ao facto da resposta não ter nenhum conteúdo, visto que apenas contém os *headers* do protocolo *HTTP*.

#### 6.2.1.2 M2M

O *broker* para o qual foram publicados encontra-se na FEUP, é utilizado por várias pessoas diariamente e implementa o protocolo *HTTPS*. Devido a este *broker* implementar este protocolo, em vez de ser avaliado o tamanho do *payload TCP* do pacote *HTTP* da publicação é avaliado o tamanho do *payload TCP* do pacote *Application Data* do protocolo *TLS*. A figura 6.10 ilustra a diferença no encapsulamento entre o uso de *HTTPS* ou não.




(a) Encapsulamento no caso do *broker* utilizado no *FIWARE*, que não implementava *HTTPS*

(b) Encapsulamento no caso do *broker* utilizado no *M2M*, que implementava *HTTPS*

Figura 6.10: Diferenças nos encapsulamentos dos pacotes nos *brokers* utilizados no *FIWARE* e *M2M*

De seguida é apresentado um resumo do que foi possível observar através das experiências efetuadas. Através das medições ao nível da aplicação é possível verificar o seguinte:

- a notificação relativa à subscrição de um dado recurso é sempre recebida antes do *publisher* receber o OK da publicação desse recurso, funcionando deste modo de uma forma diferente em relação ao *FIWARE*. Isto pode ser um problema no caso em que existam muitas subscrições a um recurso, constituindo um problema de escalabilidade e aumentando desta forma consideravelmente o tempo de publicação;
- conforme o número de publicações concorrentes aumenta, apenas o *tPub* e *tSub* aumentam;
- não se registaram *retries* ao nível da aplicação.

Através das medições ao nível da rede é possível observar o seguinte:

- o *content-length* tem 390 *bytes* em média, bastante superior quando comparado ao do *FIWARE*. É é cerca de 1119% superior ao tamanho ocupado pelas variáveis relativas aos dados a publicar;

- o tamanho do *payload TCP* do pacote *Application Data* do protocolo *TLS* é cerca de 768 *bytes*, como tal este é 426% superior ao tamanho do *Content-Length*;
- são enviados cerca de 2029 *bytes* em direção ao servidor, e são recebidos à volta de 1937 *bytes*. A partir desta informação é possível concluir o seguinte:
  - o número total de *bytes* recebidos no sentido servidor->cliente é 4,5% menor que no sentido cliente->servidor, havendo desta forma maior *overhead* na resposta em relação ao *FIWARE*;
  - o total de *bytes* de uma publicação é cerca de 416% superior ao tamanho do *payload TCP*;
  - para publicar um *JSON* de 146 *bytes* de tamanho são utilizados 3969, um aumento de cerca de 2618%.
- não foram registadas nenhumas retransmissões, tanto de pacotes *TCP* como de pacotes *HTTP*;
- o *Round Trip Time* mantém-se idêntico ao longo do dia.

Observando estes dados mais em detalhe, foi procurado perceber o porquê do aumento do *content-length* em relação ao *FIWARE*, visto que o *M2M* não impõe nenhum tipo de *overhead* na formatação dos dados. Foi verificado que apenas 146 *bytes* são relativos aos dados a publicar, sendo que o restante deve-se a outros parâmetros incluídos no *JSON* impostos pela biblioteca utilizada. Estes parâmetros extra incluídos são por exemplo o *timestamp* da criação do recurso ou *content-type*. Este está erradamente incluído também no *JSON* com os dados a enviar, em vez estar apenas presente nos *headers* do pedido *web (HTTP)*. O facto de este estar incluído também no *JSON* enviado, em vez de estar apenas nos *headers*, constitui um erro de implementação da biblioteca. O tamanho do *JSON* apenas com os dados — 146 *bytes* — é inferior ao *content-length* do *FIWARE* (que apenas tem os dados em si), visto não ser necessário todo o *overhead* incluído por este.

Adicionalmente, foi procurado perceber o facto de não se registarem retries ao nível da aplicação nem ao nível da rede, e verificou-se que que a biblioteca utilizada não suporta mais que duas ligações de rede ativas em simultâneo. Uma ligação ativa de rede é uma ligação em que já foi enviado o pacote *TCP SYN* e a resposta com um pacote *TCP ACK*, mas ainda não foi recebido um pacote *TCP FIN,ACK* a sinalizar o seu término. Isto foi possível verificar através da análise das capturas de rede efetuadas. Deste modo, o número de publicações concorrentes (*threads* ativas) não têm praticamente nenhum impacto no desempenho, visto que independentemente deste número nunca existem mais que 2 ligações de rede ativas ao mesmo tempo.

Visto que nunca existem mais que duas ligações de rede ativas ao mesmo tempo, nunca há esforço suficiente no *broker* para causar estes *retries*. Estas observações também explicam o facto do tempo total se manter idêntico mesmo variando o número de publicações concorrentes, como é possível verificar na figura 6.11. Também explicam o facto do *goodput* não variar com o aumento

do número de publicações concorrentes e ser significativamente superior em relação ao *FIWARE*, e no entanto este ser cerca de 10 vezes mais rápido a publicar, dado que o *M2M* demora cerca de 48 minutos por cada ciclo de publicações.

O *goodput* foi em média 2622 *bytes*/segundo, o percentil 25% situou-se nos 2499 *bytes*/segundo e o percentil 75% nos 2729, muito superior quando comparado ao *FIWARE*. Neste a média do *goodput* com 50 publicações em simultâneo — o ciclo de publicação com menor tempo de publicação/subscrição/total e maior *goodput* – está entre os 1500 e 1600 *bytes*/segundo, quando contabilizados todos os conjunto de ciclos de publicação.

Em termos do tempo de publicação, o valor registado mais baixo, apenas com 50 pedidos em simultâneo, foi de 5992 milissegundos em média, um valor muito maior que o do *FIWARE*, que registou uma média de cerca de 630 milissegundos com o mesmo número de pedidos em simultâneo. Esta diferença é atribuída ao facto de apenas existirem duas ligações de rede ativas ao mesmo tempo, pois o recurso não é logo publicado após a chamada da função que efetua a publicação. Estes tempos (de publicação) continuam a subir, sendo que o valor máximo registado foi com 500 pedidos em simultâneo, com uma média de cerca de 72858 milissegundos, que pode ser observado na figura 6.12. O tempo de subscrição registado seguiu os mesmos moldes do tempo de publicação, sendo que a subcrição foi sempre enviada cerca de 45 milissegundos antes do OK referente à publicação, como tal a média do tempo de subscrição é cerca de 45 milissegundos menor que a do tempo de publicação.

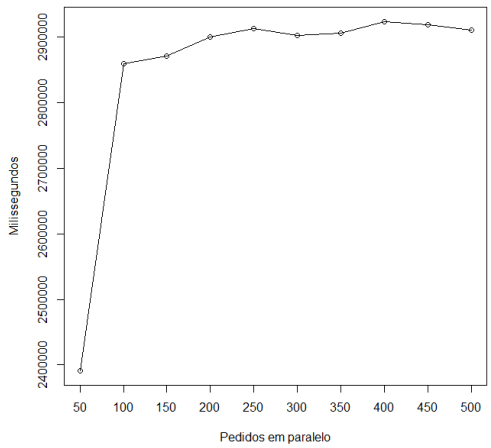



Figura 6.11: Tempo total que cada ciclo de publicação demorou

.

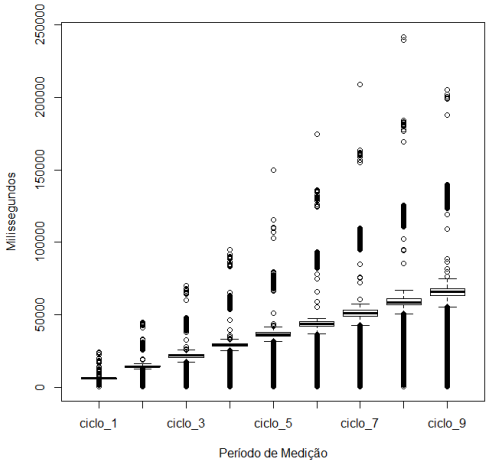



Figura 6.12: Variação do tempo de publicação ao longo dos vários ciclos de publicação

.

Estudando mais em profundidade a quantidade de *bytes* de uma ligação, é possível verificar que, mesmo descontando o *overhead* (tanto no conteúdo das mensagens como nas próprias adicionais introduzidas por este protocolo) introduzido pelo *TLS* (que o *broker* de *FIWARE* utilizado não implementa), esta é significativamente superior à registada no *FIWARE*. Este *overhead* é proveniente das seguintes fontes [The08]:

- Mensagens *Client Hello* e *Server Hello*, enviado do cliente para o servidor e do servidor para o cliente respetivamente e utilizados para estabelecer melhorias na segurança da comunicação entre o cliente e o servidor. Estabelecem os atributos *Protocol Version*, *Session ID*, *Cipher Suite*, *Compression Method* e dois números aleatórios, um no cliente e um no servidor. O *frame* destas mensagens tem 296 e 152 *bytes*, respetivamente.

- Mensagem *Change Cipher Spec*, que indica que as comunicações seguintes serão protegidas com o *cipher spec* e chaves previamente negociadas. É enviada tanto pelo servidor como pelo cliente. O *frame* desta mensagem tem 72 *bytes*.

- Mensagem *Encrypted Handshake Message*, que corresponde à mensagem *Finished* do protocolo de *Handshake* do *TLS*, verificando que a troca de chaves e o processo de autenticação foi bem sucedido e sinalizando que já é possível enviar e receber *application data*. É enviada pelo cliente e o *frame* desta mensagem tem 151 *bytes*.

- Mensagem *Application Data*, onde são enviados os dados da publicação encriptados, que adiciona um *overhead* de cerca de 40 *bytes*[3]. O *frame* desta mensagem tem cerca de 834 *bytes*.

[3] http://netsekure.org/2010/03/tls-overhead/

- Por fim, a mensagem *Encrypted Alert*, neste caso simbolizando um *alert* de fim de comunicação. Neste caso um *Closure Alert*, que indica que quem enviou esta mensagem não enviará mais mensagens nesta ligação, procedendo-se desta forma ao fim da ligação (*TCP FIN,ACK*.

Somando todo este *overhead* introduzido pelo protocolo *TLS*, as suas mensagens e os respetivos pacotes de *acknowledgment*, é possível verificar que adiciona cerca de 892 *bytes* no sentido cliente->servidor e 639 *bytes* de *overhead* no sentido servidor->cliente. Se este for retirado ao total da ligação em cada um dos sentidos, podemos verificar que o total de *bytes* no sentido cliente->servidor seria 1137 e no sentido servidor->cliente seria 1298 *bytes*. Mesmo assim é significativamente superior aos valores do *FIWARE* apresentados anteriormente.

O facto de o total da ligação no servidor->cliente ser superior ao cliente->servidor (neste sentido é onde é contabilizada a publicação dos dados) deve-se ao facto da resposta à publicação ser superior à publicação em si, o que não acontece no *FIWARE*. A publicação tem cerca de 834 *bytes* e vem num só *frame* enquanto que a resposta vem em dois e tem em média 1118 *bytes* (983 + 135). Como tal, a resposta não é um simples OK (com *status* 200/204) mas sim contém uma série de informação, adicionando ineficiência ao *middleware*, em especial em ligações com pouca largura de banda.

#### 6.2.1.3 Comparação

Finalizando e sumariando o que já escrito acima, são aqui descritas as diferenças observadas entre os dois *middlewares*:

- o *M2M* envia as notificações sempre antes do OK ao contrário do *FIWARE*, constituindo um problema de escabilidade no caso de existirem várias subscrições a um dado recurso;

- Os tempos de subscrição e publicação observados no *M2M* também são maiores quando comparados aos do *FIWARE*, cerca de 850% superiores, nomeadamente devido ao facto da limitação da biblioteca em apenas suportar duas ligações de rede ativas ao mesmo tempo. O facto de apenas existirem duas ligações de rede ativas ao mesmo tempo explica a não existência de qualquer tipo de *retries*;

- O *FIWARE* é mais eficiente em termos do tamanho do *content-length* do *JSON* com os dados a publicar, visto que a biblioteca de *M2M* utilizada insere uma série de parâmetros adicionais no *JSON* a enviar além dos dados a publicar. Deste modo, o tamanho *content-length* registado no *M2M* é cerca de 55% superior ao do *FIWARE*;

- o *FIWARE* é mais eficiente no tamanho do *payload TCP*, dado que o *payload* no caso do *M2M* é cerca de 42,5% superior ao registado pelo *FIWARE*. Isto é devido ao facto do tamanho do *content-length* do *M2M* ser superior ao do *FIWARE*.

- o *FIWARE* também é mais eficiente em termos do número de *bytes* totais utilizados para publicar um recurso, tanto no sentido cliente->servidor, em que o *M2M* é 34% mais ineficiente em relação ao *FIWARE*, como no sentido servidor->cliente, no qual o *M2M* é cerca de 88% superior. Esta diferença é devido ao facto da resposta à publicação não ser apenas um *status* 200 *OK* com *content-length* nulo, mas conter dados, nomeadamente o *JSON* do recurso publicado

De notar que as comparações em termos do *payload TCP* e *bytes* totais foram calculadas descontando o *overhead* imposto pelo procotolo *HTTPS* no caso do *M2M*, visto que o *broker* do *FIWARE* não implementava este. Para finalizar, a biblioteca de *M2M* apenas permitia duas ligações de rede ativas ao mesmo tempo, como tal, não existiu nenhum tipo de *retry*, o que também explica o facto do *goodput* ser superior ao registado pelo *FIWARE* mesmo este sendo muito mais rápido a publicar todos os dados (cerca de 10 vezes mais rápido).

As tabelas 6.1 e 6.2 ilustram um apanhado de todas as medições e os seus valores em cada um dos *middlewares*, indicando as suas médias. No caso de estas variarem consoante o número de pedidos ao mesmo tempo, é apresentada a média com 50 e 500 pedidos simultâneo, demonstrando assim a variação das métricas. Certas métricas como o *goodput* e o tempo total de publicação não variam no caso do *M2M* dado que apenas existem duas ligações ativas ao mesmo tempo, logo aumentar o número de pedidos em simultâneo nao afeta o seu desempenho. No caso do *M2M* existem valores dentro de parêntisis, que se referem aos valores sem o *overhead* imposto pela encriptação.

Através das tabelas referidas acima é possível verificar que o *FIWARE* é mais eficiente e rápido em relação ao *M2M* exceto no tamanho *content-length* do *JSON* com os dados. Esta eficiência deve-se ao facto do *M2M* não impor nenhum tipo de formatação obrigatória nos dados como verificado no capítulo 5, não acarretando desta forma um *overhead*.

Tabela 6.1: Resumo das comparações entre o *FIWARE* e *M2M* no segundo cenário

| | tPub | tSub | Tempo total de publicação | Retries | Content-Length |
|---|---|---|---|---|---|
| FIWARE | 630-7400 ms | 650-7430 ms | 4-5 m | 9-283 | 251 b |
| M2M | 5992-72858 ms | 5958-72126 ms | 48 m | 0 | 390 b |

Tabela 6.2: Resumo das comparações entre o *FIWARE* e *M2M* no segundo cenário

| | TCP Payload | bytesTotaisCS | bytesTotaisSC | Retransmissões TCP | Retransmissões HTTP | Goodput |
|---|---|---|---|---|---|---|
| FIWARE | 511 b | 850 b | 689 b | 0-53 | 0-20 | 550-1700 b/s |
| M2M | 768(728) b | 2029(1137) b | 1937(1298) b | 0 | 0 | 2600 b/s |

### 6.2.2 Cenário Publish/Subscribe Sequencial

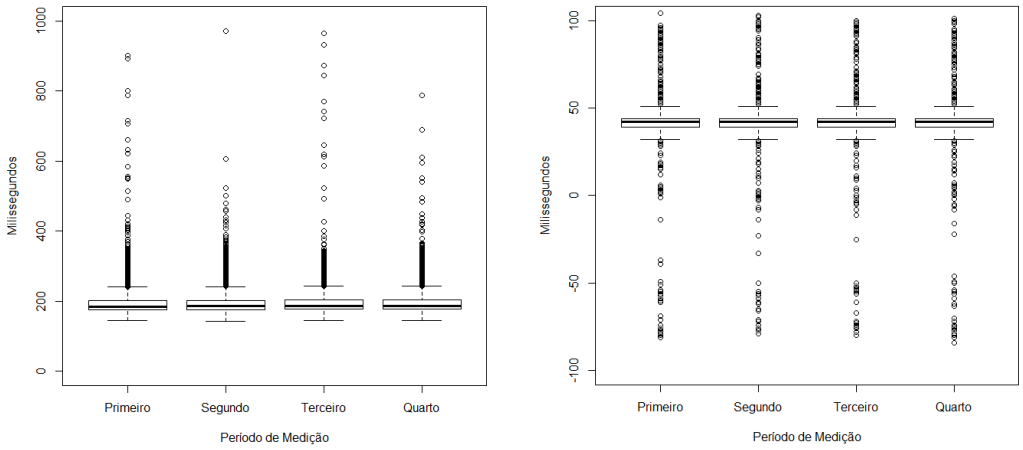
#### 6.2.2.1 FIWARE

Em relação a este cenário, apenas foram registadas diferenças no tempo de publicação, subscrição e total de publicação, em relação ao cenário de *publish/subscribe* com pedidos em simultâneo.

Este aumento do tempo de publicação afeta o *goodput* por consequência, visto que os mesmos dados demoram mais tempo a serem publicados. Os tempos de publicação registados foram bastante inferiores, sendo que a média foi de 186 milissegundos, o percentil 25% situou-se nos 175 milissegundos e o percentil de 75% nos 202. No entanto, existem alguns *outliers*, como é possível verificar na figura 6.13a.

O tempo de subscrição registado também foi bastante inferior. A média situou-se nos 227, o percentil 25% nos 217 e o percentil 75% nos 243 milissegundos. Também foram registados alguns *outliers* nos mesmos moldes que o tempo de publicação. Calculando a diferença entre o tempo de publicação e subscrição, é possível observar a mesma situação que no caso dos pedidos simultâneos, isto é, as notificações das subscrições tanto são enviadas antes ou depois do *OK* da publicação. Esta observação está ilustrada na figura 6.13b.

Por fim, o *goodput* registado foi maior, dado que o tempo de publicação é menor. Como tal, a diferença entre o *timestamp* do primeiro e último pacote é menor, e portanto o valor da divisão é maior, visto a quantidade de dados em cada publicação ser a mesma.




(a) *Box plot* sobre o tempo de publicação no cenário de *publish/subscribe* no modo sequencial, sendo que cada *tick* do eixo do x representa um ciclo de publicação

(b) *Box plot* sobre a diferença entre o tempo de publicação e o de subscrição no cenário de *publish/subscribe* no modo sequencial, sendo que cada *tick* do eixo do x representa um ciclo de publicação

Figura 6.13: Evolução do tempo de subscrição e da diferença entre o tempo de publicação e subscrição ao longo dos ciclos de publicação no cenário de *publish/subscribe* sequencial

#### 6.2.2.2 M2M

Apenas foram observadas diferenças nos tempos de publicação, subscrição, total, e no *goodput*, devido à natureza sequencial deste cenário. Em termos do tempo de publicação, a média foi 282 milissegundos, o percentil 25% situou-se nos 270 e o 75% nos 294. A média do tempo de subscrição foi de 237 milissegundos, o percentil 25% 225 e o 75% 250. Estes tempos são bastante inferiores ao cenário com vários pedidos em simultâneo, muito devido ao facto de a biblioteca

apenas permitir duas ligações de rede ao mesmo tempo. Como tal, o tempo de publicação/subscrição é extremamente alto, visto os pedidos demorarem muito tempo a serem realmente efetuados após a chamada da função que os faz. Também são superiores aos do *FIWARE* neste cenário, mas isso poderá dever-se ao facto do *broker* utilizado pelo *M2M* ser inferior em termos de desempenho e *specs* quando comparado ao do *FIWARE*.

Para finalizar, o *goodput* foi menor, visto que a média situou-se nos 1444 *bytes*/segundo, inferior à média registada com vários pedidos em simultâneo, que foi cerca de 2600. As medições neste cenário foram feitas num dia diferente do cenário anterior, portanto esta diminuição do *goodput* (quando deveria aumentar, visto apenas ser publicado um recurso ao mesmo tempo, em vez de dois, logo o tempo de processamento da publicação devia ser menor) poderá dever-se a motivos internos ao funcionamento do *broker*, ou motivos de aumento do esforço atual no mesmo nesse dia.

#### 6.2.2.3 Comparação

Visto que o tempo total de publicação se situou entre os 63-64 minutos no caso do *FIWARE* e 95 no caso do *M2M*, um aumento de 48%, publicar os dados desta forma não seria prático para o caso de uso em questão. Visto o objetivo nestes cenários é a publicação de hora a hora das velocidades médias — disponibilizando-as publicamente para depois serem utilizadas para outros efeitos, nomeadamente em aplicações de terceiros — o ideal é que esta publicação seja curta (e inferior a 1 hora) para que os utilizadores tenham acesso rápidamente à informação. Deste modo, a publicação com vários pedidos em simultâneo é a forma mais indicada.

Nas tabelas 6.3 e 6.4 encontra-se uma compilação de todas as medições efetuadas em cada um dos *middlewares*. Em cada coluna da tabela está o valor médio das várias métricas avaliadas. Através destas tabelas é possível verificar que o *FIWARE* é mais eficiente e rápido quando comparado ao *M2M*. No caso do *M2M* existem valores dentro de parêntisis, que se referem aos valores sem o *overhead* imposto pela encriptação.

Tabela 6.3: Resumo das comparações entre o *FIWARE* e *M2M* no primeiro cenário

| | tPub | tSub | Tempo total de publicação | Retries | Content-Length |
|---|---|---|---|---|---|
| FIWARE | 186 ms | 227 ms | 63-64 m | 0 | 251 b |
| M2M | 282 ms | 237 ms | 95 m | 0 | 390 b |

Tabela 6.4: Resumo das comparações entre o *FIWARE* e *M2M* no primeiro cenário

| | TCP Payload | bytesTotaisCS | bytesTotaisSC | Retransmissões TCP | Retransmissões HTTP | Goodput |
|---|---|---|---|---|---|---|
| FIWARE | 511 b | 850 b | 689 b | 0 | 0 | 2660 b/s |
| M2M | 768(728) b | 2029(1137) b | 1937(1298) b | 0 | 0 | 1444 b/s |

### 6.2.3 Cenário Request/Response

#### 6.2.3.1 REST

Neste caso, os dados estavam alojados numa base de dados relacional que utiliza *Postgresql*, funcionando como uma aplicação convencional de *REST*. Este é o caso base de comparação, sendo que tanto o *FIWARE* e o *M2M* são comparados com estes resultados de forma a determinar o *overhead* imposto por eles.

Em termos de tempo de resposta, demorou cerca de 557 milissegundos.

Em relação às métricas ao nível da rede, foram enviados 234386 *bytes* no total, sendo que 1620 desses são referentes a *acknowledges TCP* e 644 ao OK ao pedido *web* ao recurso. O *Content-Length*, relativo ao *JSON* com os resultados do *GET*, foi de 223756 *bytes*.

#### 6.2.3.2 FIWARE

Com este *middleware*, os dados estavam alojados na infraestrutura do mesmo. Em termos do tempo de resposta, demorou cerca de 4747 milissegundos a obter os resultados, registando um aumento de 750% quando comparado aos tempos obtidos no *REST*.

Em termos do total de *bytes* enviados, foram enviados 2034454, 768% superior em relação ao caso base, sendo que 15288 são referentes a pacotes de *acknowledge*. Estes totalizam mais *bytes* em comparação ao *REST* visto que são enviados mais pacotes com dados, logo o número de *acknowledges* é superior (283 neste caso e 30 no *REST*). Foram enviados 426 *bytes* para o OK relativo ao pedido *web* ao recurso efetuado. O *Content-length* tem um tamanho de 1946336 *bytes*, correspondendo a um aumento de 770%.

#### 6.2.3.3 M2M

Neste *middleware* foram registadas dificuldades técnicas devido à implementação da biblioteca usada. Neste cenário não foi excepção, sendo que se registaram dois problemas distintos:

- a função que obtém os recursos não retorna todos os recursos, apenas retornando cerca de 9961, pouco mais de metade;
- a obtenção dos recursos é bastante demorada, demorando cerca de 886143 milissegundos, sendo desta forma inviável para este cenário.

#### 6.2.3.4 Comparação

Conforme os resultados aqui apresentados, é possível verificar que o *FIWARE* é consideravelmente mais lento que o caso base, o *REST*, sendo cerca de 750% mais lento que este e enviando cerca de 768% mais *bytes* pela rede para enviar a mesma informação que o *REST*. Isto deve-se à formatação imposta na resposta pelo *FIWARE*, exponenciado pelo facto de serem milhares de resultados (4802). É possível calcular este *overhead* através da diferença entre o *Content-Length*

dos dados do *FIWARE* e *REST*, dividindo-a pelo número de resultados. Foi obtido um valor de cerca de 422 *bytes* de *overhead* no *Content-Length* por cada resultado.

Não foram efetuados testes de carga, visto o objetivo ser perceber se o *middleware* funciona neste cenário, e não desenvolver um sistema robusto e preparado para cenários de carga alta/extrema.

Infelizmente, devido a problemas técnicos coma biblioteca de *M2M*, não foi possível comparar este com o *FIWARE*, o que seria muito interessante, dado que um dos objetivos era perceber como estes dois *middlewares* se comportam face ao caso base, o *REST*.

# Capítulo 7

# Conclusões e Trabalho Futuro

O objetivo desta dissertação consistiu numa avaliação quantitativa e qualitativa de vários *middlewares* para o desenvolvimento de serviços e aplicações para *smart cities*. Para tal, foi desenvolvida uma *tool box* para ajudar a automatizar a avaliação quantitativa. Esta dissertação procura ajudar no processo de decisão aquando da escolha de um *middleware* neste contexto, possibilitando esta escolha com base no desempenho do *middleware*, oferecendo dados concretos para esta decisão.

Foi feita uma revisão bibliográfica de modo a identificar standards para o *benchmark* de *middlewares* aplicados a *Smart Cities* e *Internet of Things*. No entanto, não foram encontrados quaisquer. Como tal, foi proposta uma metodologia para as avaliações quantitativa e qualitativa de *middlewares* neste contexto.

A avaliação qualitativa permitiu definir a aplicabilidade de cada *middleware* em cada cenário, verificar o cumprimento de vários requisitos pelos *middlewares*, os recursos que cada um destes disponibilizam em termos de documentação e suporte, e por fim descobrir falhas. Em relação a estas, foi descoberto por exemplo o facto da biblioteca de *M2M* não permitir o *unsubscribe* de recursos ou o facto do *FIWARE* impor limites no tamanho máximo dos pedidos de publicação e nas notificações. Este limites obrigaram a um *workaround*, publicando apenas um *edge* por recurso, ao invés de publicar todos um *edges* apenas em um recurso.

Em termos da avaliação quantitativa, foi concluído que nos dois primeiros cenários o *middleware M2M* (devido à biblioteca utilizada) tem um problema de escabilidade, visto as notificações da subscrição serem sempre enviadas antes do OK da publicação. O *FIWARE* é mais eficiente em termos dos recursos de rede utilizados bem como em termos da velocidade de publicação/subscrição. Foi descoberto um problema de implementação da biblioteca de *M2M*, visto que esta inclui o *content-type* no *JSON* a enviar no pedido *web*, em vez de apenas o incluir nos *headers* do mesmo. Adicionalmente, foi verificado que o primeiro cenário (em que os dados são publicados de forma sequencial) não se adequa a este caso de uso, visto o seu tempo total de publicação ser demasiado excessivo, pois este é superior a uma hora, para o caso de uso estudado. Como tal, não é uma opção válida para este caso de uso, dado que os dados têm que ser publicados de hora a hora. No terceiro cenário, devido a problemas relacionados com a biblioteca do middleware *M2M* utilizada,

não foi possível obter resultados conclusivos para este, sendo que apenas foi possível comparar o *FIWARE* com o caso base — o *REST* — em vez de comparar o *FIWARE* ao *M2M* e depois comparar estes dois com o caso base.

Os objetivos propostos foram atingidos, na medida em que:

- foi proposta e executada uma metodologia para as avaliações quantitativa e qualitativa de *middlewares* neste contexto;
- foi feito um levantamento de requisitos para estes *middlewares*, bem como uma avaliação qualitativa tendo em conta a sua aplicabilidade nos cenários propostos, as suas limitações e outros aspetos como a sua documentação ou suporte, possibilitando uma caracterização mais aprofundada destes;
- foi desenvolvida uma *tool box* que no futuro poderá evoluir para uma verdadeira *framework* capaz de avaliar um *broker* de um determinado *middleware* e apresentar imediatamente vários resultados (gráficos, tabelas), caracterizando este em termos quantitativos;
- foram apresentados os resultados da avaliação quantitativa nos cenários propostos, caracterizando os *middlewares* em termos de desempenho, bem como ajudando a perceber aspetos obscuros das suas implementações, essenciais aquando da decisão sobre qual *middleware* escolher.

Ao longo do desenvolvimento foram encontradas algumas dificuldades técnicas, em especial no *M2M*. Neste, devido aos vários problemas já abordados acima e nos capítulos anteriores, os resultados relacionados com os tempos (publicação/subscrição/total) e *retries* no dois primeiros cenários—e em geral no terceiro cenário—, não foram tão precisos e completos como poderiam.

## 7.1 Trabalho Futuro

Em termos de trabalho futuro, uma opção possível é transformar esta *tool box* numa *framework*. Isto é, existir uma aplicação para *desktop* (*Windows*, *macOS* e *Linux*), possivelmente desenvolvida com as *frameworks NW*[1] ou *Electron*[2], que permitem desenvolver aplicações multiplataforma usando tecnologias *web*, como *HTML*, *CSS* e *Javascript*. Estas duas *frameworks* seriam utilizadas para não ser necessário desenvolver a *framework* com ferramentas nativas em cada sistema operativo. A aplicação *desktop* permitiria escolher o *middleware* para realizar as medições, qual o endereço e porta do *broker* desejado e também qual o tipo de teste. O *output* seriam os ficheiros *CSV* já referidos anteriormente no capítulo 5 e/ou adicionalmente apresentando logo os gráficos mais relevantes bem como mostrando um *overview* dos resultados.

[1] http://nwjs.io/
[2] http://electron.atom.io/

# Referências


[All12] Open Mobile Alliance. NGSI Context Management. Technical Report May, 2012.

[Ber96a] Philip A. Bernstein. Middleware: A model for distributed system services. *Commun. ACM*, 39(2):86–98, February 1996. URL: http://doi.acm.org/10.1145/230798.230809, doi:10.1145/230798.230809.

[Ber96b] Alex Berson. *Client/Server Architecture (2Nd Ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1996.

[CDN11] Andrea Caragliu, Chiara Del Bo e Peter Nijkamp. Smart Cities in Europe. *Journal of Urban Technology*, 18(2):65–82, 2011. doi:10.1080/10630732.2011.601117.

[Cit16a] CitySDK. Deliverable - D2.1 CitySDK for Pilots, including Documentation (M12). Disponível em http://www.citysdk.eu/wp-content/uploads/2013/09/deliverable_2.1.pdf, acedido em Janeiro 2016.

[Cit16b] CitySDK. Deliverable - D3.1 Participation Pilot Application and it's SDK components. Disponível em http://www.citysdk.eu/wp-content/uploads/2013/09/CitySDK_D31_Participation-Pilot-Application-and-it%E2%80%99s-SDK-components_final.pdf, acedido em Janeiro 2016.

[Cit16c] CitySDK. Deliverable - D3.2 – Participation, Lead and Replication Pilots reports, Smart Participation SDK Components. Disponível em http://www.citysdk.eu/wp-content/uploads/2013/09/D3.2-Smart-Participation-Lead-and-Replication-Pilots-reports-Smart-Participation-SDK-Components_final.pdf, acedido em Janeiro 2016.

[Cit16d] CitySDK. Deliverable - D4.2 – Mobility, Lead and Replication Pilots reports, Smart Mobility SDK Components. Disponível em http://www.citysdk.eu/wp-content/uploads/2013/09/D4.2-Mobility-Lead-and-Replication-Pilots-reports-Smart-Mobility-SDK-Components.pdf, acedido em Janeiro 2016.

[Cit16e] CitySDK. Deliverable - Tourism Pilot Application and its SDK Components. Disponível em http://www.citysdk.eu/wp-content/uploads/2013/09/CitySDK_D5.1.pdf, acedido em Janeiro 2016.

[Com00] Douglas Comer. *Internetworking with TCP/IP*. Prentrice Hall, Fourth edition, 2000.

[DSs12] Yousef Daradkeh e Manfred Sneps-sneppe. M2M Standards : Possible Extensions for Open API from ETSI. *European Journal of Scientific Research*, 72(4):628–637, 2012.

[ECM16] ECMA International. JSON. Disponível em http://www.json.org/, acedido em Janeiro 2016.

[EFGK03] Patrick TH. Eugster, Pascal A. Felber, Rachid Guerraoui e Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003. URL: http://portal.acm.org/citation.cfm?doid=857076.857078, doi:10.1145/857076.857078.

[Ets13] Etsi Tc M2M. ETSI TR 101 584 - Study on Semantic support for M2M Data. Technical report, 2013. URL: http://www.etsi.org/deliver/etsi_tr/101500_101599/101584/02.01.01_60/tr_101584v020101p.pdf.

[ETS16a] ETSI. ETSI TS 102 690 - Machine-to-Machine communications (M2M), Functional architecture, mIa, dIa and mId interfaces. Disponível em http://www.etsi.org/deliver/etsi_ts/102600_102699/102690/02.01.01_60/ts_102690v020101p.pdf, acedido em Fevereiro 2016.

[ETS16b] ETSI. ETSI TS 102 921 - Machine-to-Machine communications (M2M), mIa, dIa and mId interfaces. Disponível em http://www.etsi.org/deliver/etsi_ts/102900_102999/102921/01.02.01_60/ts_102921v010201p.pdf, acedido em Fevereiro 2016.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf, doi:10.1.1.91.2433.

[FIW16] FIWARE. FIWARE.OpenSpecification.Data.ContextBroker. Disponível em https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.OpenSpecification.Data.ContextBroker, acedido em Janeiro 2016.

[For16] Internet Engineering Task Force. Rfc 1122 - requirements for internet hosts - communication layers. Disponível em https://tools.ietf.org/html/rfc1122, acedido em Maio 2016.

[Fut16] Future Cities Project. Future Cities Project inital web page. Disponível em http://futurecities.up.pt/site/, acedido em Janeiro 2016.

[Goo16] Google. Google trends de rest e soap. Disponível em http://www.google.com/trends/explore?hl=en-US#q=soap+api,+rest+api&cmpt=q, acedido em Janeiro 2016.

[GPH05] Yuanbo Guo, Zhengxiang Pan e Jeff Heflin. LUBM : A benchmark for OWL knowledge base systems. 3:158–182, 2005. doi:10.1016/j.websem.2005.06.005.

[IET16] IETF. RFC 1157 - Simple Network Management Protocol (SNMP). Disponível em https://tools.ietf.org/html/rfc1157, acedido em Fevereiro 2016.

[Jef16] Jeffrey Fulmer. siege: HTTP/HTTPS stress tester - Linux man page. Disponível em http://linux.die.net/man/1/siege, acedido em Junho 2016.

[JLD+05] J Jack, Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John Mccalpin, David Bailey e Daisuke Takahashi. Introduction to the HPC Challenge Benchmark Suite. 2005.

[Lab16] Altice Labs. Ngm2. Disponível em http://www.alticelabs.com/content/WP-NGM2M.pdf, acedido em Fevereiro 2016.

[MMY09] J. Meng, S. Mei e Z. Yan. RESTful Web Services: A Solution for Distributed Data Integration. *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, 6(4):1–4, 2009. URL: http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5365234, doi:10.1371/journal.pone.0018636.

[PA14] Carlos Pereira e Ana Aguiar. Towards efficient mobile M2M communications: Survey and open challenges. *Sensors (Switzerland)*, 14(10):19582–19608, 2014. doi:10.3390/s141019582.

[Pin15] Antonio Pinto. NA Library Guide. Technical report, 2015.

[PL03] R. Perrey e M. Lycett. Service-oriented architecture. *IEEE Symposium on Applications and the Internet Workshops*, pages 116–119, 2003. doi:10.1109/SAINTW.2003.1210138.

[Rag16] Raghunath Nambiar - Cisco. Benchmarking Internet of Things (IOT). Disponível em http://blogs.cisco.com/datacenter/industry-standards-for-benchmarking-iot, acedido em Junho 2016.

[Rou05] Matthew Roughan. Fundamental bounds on the accuracy of network performance measurements. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):253, 2005. doi:10.1145/1071690.1064241.

[SAKB] Kai Sachs, Stefan Appel, Samuel Kounev e Alejandro Buchmann. Benchmarking Publish / Subscribe-based Messaging Systems.

[SKB+09] Kai Sachs, Samuel Kounev, Jean Bacon, Alejandro Buchmann e Distributed Systems Group. Performance Evaluation of Middleware using the SPECjms2007 Benchmark. 66:410–434, 2009. doi:10.1016/j.peva.2009.01.003.

[Tel16] Telefonica. Orion Context Broker NGSI API v1 Specification. Disponível em http://telefonicaid.github.io/fiware-orion/api/v1/, acedido em Janeiro 2016.

[The08] The Internet Engineering Task Force. RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2. Disponível em https://tools.ietf.org/html/rfc5246?as_url_id=AAAAAAXUh3l4AXPNWPOv1PkNwni0k86B9nN-C5HX5SkDVOAH0gaBY2fnkF9tAZjTODl8lHyLLSf1qnpQjAZ6L8b8tClI, Agosto 2008.

[The16] The Apache Software Foundation. ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4. Disponível em https://httpd.apache.org/docs/2.4/programs/ab.html, acedido em Junho 2016.

[W3C16a] W3C. HTTP Status Code Definitions. Disponível em https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html, acedido em Janeiro 2016.

[W3C16b] W3C. HTTP/1.1: Header Field Definitions. Disponível em https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html, acedido em Março 2016.

[W3C16c] W3C. Points of Interest Core - W3C Working Draft 12 May 2011. Disponível em https://www.w3.org/TR/poi-core/, acedido em Janeiro 2016.

[Web16] Programmable Web. Página inicial. Disponível em http://www.programmableweb.com/, acedido em Janeiro 2016.

[WQA+02] Yi-Min Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson e Helen J. Wang. Subscription Partitioning and Routing in Content-based Publish/Subscribe Systems. *16th International Symposium on DIStributed Computing (DISC'02)*, page 8, 2002. URL: http://cis.poly.edu/westlab/papers/cntdstrb/disc2002.pdf.

# Anexo A

# Anexos

Requisitos para aplicações de *Internet of Things* da *Internet of Things Architecture*[1]:

- UNI.001 - O sistema deve oferecer meios para permitir às pessoas usar serviços de *Internet of Things* anonimamente.
- UNI.002 - Utilizadores humanos têm controlo de como os seus dados são expostos para outros utilizadores.
- UNI.005 - O sistema deve suportar comunicação baseada em eventos, periódica e/ou autónoma.
- UNI.008 - Os serviços e aplicações do sistema devem ser interoperáveis.
- UNI.016 - O sistema deve suportar localizações geográficas.
- UNI.022 - O sistema deve oferecer acesso seguro aos recursos.
- UNI.023 - O sistema deve oferecer acesso a informação externa.
- UNI.030 - O sistema deve providenciar uma infraestrutura de resolução de nomes, endereçamento e atribuição de entidades virtuais e serviços.
- UNI.036 - O sistema deve permitir obter a auto-descrição das coisas/dispositivos (poder perceber quando seja necessário mudar alguma coisa com base nos seus atributos).
- UNI.047 - O sistema deve assegurar interoperabilidade entre objetos ou aplicações.
- UNI.048 - O sistema deve permitir interoperabilidade entre nomes e endereços.
- UNI.067 - O sistema deve oferecer várias permissões de acessos aos dados.
- UNI.071 - O sistema deve oferecer comunicação semântica e standardizada entre serviços.

[1] http://www.iot-a.eu/public

- UNI.073 - O sistema deve permitir a descrição semântica de entidades físicas e serviços por parte de um utilizador.
- UNI.092 - Serviços remotos devem ser acessíveis por utilizadores humanos (por exemplo aceder aos serviços via *smartphone*).
- UNI.094 - O modelo de referência da arquitetura deve suportar qualquer cenário de *Internet of Things*.
- UNI.240 - O sistema deve oferecer interfaces unificadas para aceder e fazer *queries* aos metadados das entidades/recursos.
- UNI.245 - Deve ser possível criar novas aplicações (serviços compostos por serviços simples).
- UNI.405 - O sistema deve possibilitar usar mais do que um sistema de coordenadas e permitir converter de um para outro.
- UNI.406 - Deve ser possível fazer *queries* geográficas: por posição, entidades mais próximas, *navigational queries* (obter informações sobre as entidades com base na sua velocidade e direção) e alcance.
- UNI.426 - O sistema deve permitir *queries* semânticas e retornar os recursos/serviços de acordo com o seu resultado.
- UNI.607 - O sistema deve oferecer confidencialidade na comunicação.
- UNI.608 - O sistema deve oferecer integridade na comunicação.