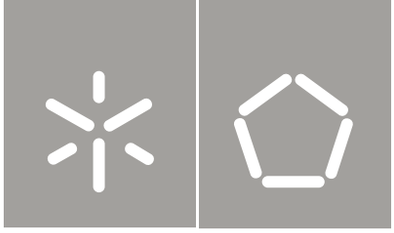
Universidade do Minho
Escola de Engenharia

Manuel Frederico da Costa Dias Pereira

Framework de sincronização para aplicações em ecrãs públicos

Framework de sincronização para aplicações em ecrãs públicos
Manuel Frederico da Costa Dias Pereira
UMinho | 2013

outubro de 2013



Universidade do Minho
Escola de Engenharia

Manuel Frederico da Costa Dias Pereira

Framework de sincronização para aplicações em ecrãs públicos

Tese de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Comunicações

Trabalho efetuado sob a orientação de
Professora Doutora Helena Rodrigues
Professora Doutora Maria João Nicolau

outubro de 2013

DECLARAÇÃO

Nome: Manuel Frederico da Costa Dias Pereira

Correio electrónico: mfdcdp@gmail.com

Tel./Tlm.: 918115937

Número do Bilhete de Identidade:13221150-5-ZZ8

Título da dissertação:

Framework de sincronização para aplicações em ecrãs públicos

Ano de conclusão: 2013

Orientadores:

Helena Cristina Coutinho Duarte Rodrigues

Maria João Mesquita Rodrigues da Cunha Nicolau Pinto

Designação do Mestrado:

Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia de Comunicações

Área de Especialização: Engenharia de Comunicações

Escola:Engenharia

Departamento: Sistemas de Informação

Guimarães, ___/___/_____

Assinatura: _____

Agradecimentos

Gostaria de agradecer à professora Helena Rodrigues pela orientação, apoio e aconselhamento, dedicação e paciência ao longo de todo este projecto, bem como à professora Maria João pelas mesmas razões.

Quero também deixar uma grande palavra de agradecimento à minha namorada Luísa por todo o apoio demonstrado ao longo de todo o projeto. Pela paciência e pelas constantes palavras de incentivo em especial nos momentos de maiores adversidades.

Agradeço também à minha família, em especial ao meu irmão Alcino e à minha cunhada Sofia pelo grande apoio demonstrado.

A todos os meus amigos agradeço o apoio, em especial ao Ricardo, Pedro, Rafael e Henrique que me acompanharam ao longo de todo o percurso académico que culminou com este projeto.

Não poderia deixar de agradecer também as palavras de apoio e incentivo dadas pelos meus patrões Paulo e Cristina.

Resumo

As redes de ecrãs públicos de grande escala são um paradigma emergente e constituem uma transformação radical na forma como se encara a disseminação de informação em espaços públicos. Apesar de existirem já várias soluções proprietárias para ecrãs públicos, estas acabam por ter uma utilidade limitada devido ao facto de constituírem soluções fechadas.

Neste trabalho de mestrado, estudamos os principais requisitos na sincronização de aplicações em ecrãs públicos e apresentamos um *Framework*, que inclui não só um modelo de sincronização, mas também uma API (*Aplication Programming Interface*) para programadores de aplicações para ecrãs públicos. Este *Framework* é baseado no PubSubHubbub, um protocolo de comunicação distribuída para a Internet, baseado no modelo Produtor/Subscritor.

Para exemplificar e avaliar a utilidade da Framework proposta foi desenvolvida uma aplicação distribuída que faz uso da mesma, constituindo assim uma prova de conceito do trabalho realizado. A aplicação desenvolvida foi colocada em servidores públicos e partilhada por um conjunto de instalações de redes de ecrãs.

Abstract

Large-scale pervasive public displays networks are becoming an emerging paradigm and represent a radical transformation in the way we think about information dissemination in public spaces. One of the features of pervasive public display systems is their ability to create experiences that span across multiple displays in a coordinated fashion. Proprietary single site display solutions exist but these are not open to third-party developers.

On the other hand, scalable open systems that enable large-scale, synchronized and multi-screen experiences, spanning multiple networks domains will call for the definition of multiple administrative boundaries that accommodate function partitioning. In our research, we have studied the key requirements involved in this open application synchronization and present our work on designing a Framework - a synchronization model and Application Programming Interface-for public displays application developers that is built on top of the PubSubHubbub protocol, an open protocol for distributed publish/subscribe communication on the Internet.

At the end, we also describe the design and implementation of a proof-of-concept that consists of a pervasive public display system formed by a display application, deployed in independent public servers and shared by a set of public displays installations.

Conteúdo

Lista de Figuras

Lista de Tabelas

Acrónimos

API	Application Programming Interface
HTML	HyperText Markup Language
MMS	Multimedia Messaging Service
NTP	Network Time Protocol
PHP	Personal Home Page
REST	Representational State Transfer
RSS	Really Simple Syndication
SLAP	Simple Lightweight Announcement Protocol
SMS	Short Message Service
SUP	Simple Update Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
VM	Virtual Machine
WWW	Word Wide Web
XML	eXtensible Markup Language

1. Introdução

1.1. Enquadramento

Nos dias de hoje, é comum encontrar ecrãs públicos nos mais diversos locais, como por exemplo, estações de transportes públicos, centros comerciais, estádios, museus ou até mesmo em ruas de grandes centros de algumas cidades. As redes de ecrãs não são mais do que um conjunto de elementos ligados entre si, numa rede local ou através da Internet. Podem partilhar conteúdos ou cada um apresentar um conteúdo específico, ao local onde se encontra instalado o terminal. Atualmente, o objetivo principal dos ecrãs públicos é a difusão de informação (publicidade, avisos, informações, etc.). As redes de ecrãs públicos podem ser vistas como um bom exemplo de um sistema distribuído já que herdam destes sistemas características importantes, como por exemplo: sistemas descentralizados, elementos ligados através de uma rede (local ou com acesso externo), conteúdos partilhados, constante expansão e evolução.

Atualmente o utilizador comum em geral está cada vez mais familiarizado com a presença de ecrãs, embora na sua maioria lhes reconheça pouca utilidade. Tal facto é um ponto bastante negativo e deve ser tido em conta. Apesar do custo do *hardware* descer de uma forma acentuada, se não forem desenvolvidas aplicações que motivem os utilizadores, a sua utilização será cada vez mais baixa. Sem dúvida que o grande desafio no desenvolvimento de redes de ecrãs é a motivação dos utilizadores para a sua utilização. Além de disponibilizarem um baixo número de funcionalidades, as redes de ecrãs de uma forma geral constituem soluções fechadas. No entanto, num futuro próximo, as redes de ecrãs públicos têm potencial para se tornarem nas novas redes sociais, sendo que para isso é necessário alterar a estratégia de desenvolvimento de aplicações. Atualmente, as aplicações para redes de ecrãs não permitem grandes interações por parte dos utilizadores, não indo muito além de ecrãs táteis. No entanto a tecnologia já vai muito para além disso, permitindo a incorporação de sensores capazes de recolher informação das mais diversas formas e dos pontos mais variados. Submissão de imagens por Bluetooth, sensores de temperatura, de movimento são bons exemplos de *hardware* que pode ser introduzido nas redes de ecrãs, contribuindo para a introdução de novas funcionalidades e aplicações. O segredo para a evolução das redes de ecrãs pode passar pela adoção de soluções abertas que permitam que todos contribuam.

Do ponto de vista de quem desenvolve as aplicações para redes de ecrãs públicos surgem novos desafios. Dadas as características deste tipo de redes, existe um conjunto de fatores que influenciam o desenvolvimento. As redes de ecrãs poderão incluir uma grande diversidade de *hardware* com as mais variadas características como sensores, ecrãs com diferente resolução e dimensão, a forma como os vários elementos estão ligados entre si e até mesmo a sua posição física. Pensando num exemplo concreto, desenvolver uma aplicação equivalente á típica proteção de ecrã "bola saltitante" para uma rede de ecrãs, implica ter em conta vários aspetos. Dado que a bola deverá passar de um ecrã para outro de acordo com uma lógica espacial, será necessário identificar como é que os vários ecrãs se encontram posicionados fisicamente. Da mesma forma será necessário sincronizar os vários movimentos da bola, para que no momento em que a bola começa a sair de um ecrã, dê entrada no ecrã seguinte na mesma posição, independentemente da resolução ou dimensão dos ecrãs. Por outro lado, deverá ser tida em conta como é que os vários ecrãs comunicam entre si. Finalmente será necessário prever que, enquanto a aplicação corre, poderá ser escalonado outro conteúdo para um ou vários ecrãs, podendo essa informação sobrepor-se à existente. De todos os desafios apresentados, através do exemplo da proteção de ecrã "bola saltitante", é notória a necessidade de sincronização nas redes de ecrãs. O exemplo descrito apresenta um cenário em que é necessário sincronizar, caso contrário poderemos obter situações de erro, como por exemplo, a bola não se encontrar em nenhum ecrã, haver duas bolas a circular em simultâneo nos ecrãs, entre outros.

Um outro tipo de aplicações, que poderá tornar-se popular nas redes de ecrãs, são os jogos, em especial jogos que promovam a interação entre dois ou mais utilizadores. Neste tipo de jogos, os movimentos ou interações de um jogador (através de comandos ou ecrãs táteis), alteram o resultado apresentado no ecrã dos restantes jogadores. Pensando num jogo de ténis, se a bola é projetada na direção do ecrã oposto, tal movimento terá que ser apresentado nesse ecrã. Este requisito de sincronização está presente já em jogos comuns, utilizados em computadores ou consolas, no entanto, no contexto das redes de ecrãs é necessário ajustar o cenário/movimentos nos vários ecrãs envolvidos. Além disso como já foi dito anteriormente, a diversidade do *hardware* existente, torna este desafio ainda mais complexo.

Além das aplicações referidas, existem inúmeros exemplos de aplicações para redes de ecrãs onde é notória a necessidade de sincronização, independentemente do tipo de aplicação, do conteúdo que apresenta ou dos próprios recursos disponíveis. Por isso é

necessário definir uma estrutura genérica capaz de dar resposta aos mais variados requisitos nos diferentes cenários. Este tema será focado com mais detalhe ao longo do trabalho, sendo apresentados vários cenários, requisitos de sincronização e respetiva solução para o problema.

1.2. Objetivos/Contribuições

O principal objetivo desde trabalho é o desenvolvimento uma API (Application Programming Interface) que ofereça às aplicações para ecrãs públicos, mecanismos que lhes permitam sincronizarem entre si, a apresentação de conteúdos. Estas aplicações devem apresentar vários tipos de requisitos de sincronização, temporais ou de reação a eventos.

Numa primeira fase serão analisadas as abordagens usadas noutros domínios para posteriormente se efetuar uma analogia com as redes de ecrãs públicos. Após o estudo e descrição dos requisitos de sincronização específicos das aplicações para redes de ecrãs públicos, será desenvolvida uma API de sincronização que seja capaz de dar resposta aos diferentes requisitos de sincronização identificados.

Por último é pretendido o desenvolvimento de uma pequena aplicação Web que seja passível de ser executada numa rede de ecrãs em que se coloquem alguns requisitos de sincronização, e que faça uso da API desenvolvida de forma a validar sua a utilidade e eficiência.

1.3. Estrutura da dissertação

A presente dissertação é composta por sete capítulos, sendo que o primeiro capítulo inclui uma breve introdução ao tema do projeto, aos seus principais objetivos e motivação.

No capítulo dois, são apresentados os vários requisitos de sincronização presentes num conjunto de cenários, para posteriormente se analisarem e escolherem os mais relevantes e comuns aos vários cenários.

No capítulo três é descrito o estado da arte que se encontra dividido em duas partes: redes de ecrãs públicos e sistemas distribuídos. Na primeira parte apresenta-se um conjunto de projetos já existentes. Na segunda, é feita uma análise às soluções/algoritmos de sincronização presentes nos sistemas distribuídos.

No capítulo quatro é feita uma descrição do funcionamento do protocolo produtor/subscritor PubSubHubbub. Além da apresentação do protocolo é apresentada uma análise e justificação da escolha do PubSubHubhub, em relação a outras alternativas

existentes.

No capítulo cinco é apresentada a Framework de sincronização desenvolvida. Numa fase inicial é feita uma descrição geral das funcionalidades e é apresentado o modelo de eventos. Numa segunda fase são descritos em pormenor os métodos e respetivos protótipos disponibilizados pela API. No final é apresentada uma breve descrição da linguagem e plataforma adotada.

No capítulo seis apresentam-se os testes e resultados obtidos. Para avaliar a Framework de sincronização foi desenvolvida uma aplicação de demonstração que é aqui descrita sucintamente juntamente com o conjunto de requisitos a que a mesma deve responder. No final do capítulo são apresentados dois cenários de testes distintos e respetivos resultados.

No capítulo sete são apresentadas as conclusões do projeto, algumas propostas de trabalhos futuros e possíveis melhorias.

2. Análise de requisitos

A nossa atenção debruçar-se-á neste capítulo sobre os diversos requisitos de sincronização de um conjunto de cenários típicos de redes de ecrãs públicos. Nesse sentido, será realizada a análise desses requisitos bem como a comparação entre os vários cenários de forma a obter os requisitos comuns.

2.1. Cenários de sincronização

Nos cenários aqui descritos, os requisitos de sincronização são evidentes, uma vez que, as instalações de ecrãs públicos, distintas e independentes, necessitam de coordenar ações entre elas. Assume-se que todas as instalações participantes, previamente procuraram e subscreveram as aplicações que suportam os diferentes cenários, provavelmente disponíveis numa *application store* (no capítulo 3 será apresentado o estado da arte em arquiteturas para redes de ecrãs públicos, onde será apresentado o componente *application store*). O processo de subscrição está fora do âmbito deste documento, mas espera-se que cada instalação seja conFigurada com o(s) URL(s) da(s) aplicação(ões) (um endpoint Web otimizado para ecrãs públicos) e um conjunto de restrições que são definidas pelos gestores da instalação. Este conjunto de restrições deverá incluir, pelo menos, as regras de escalonamento da aplicação e alguma indicação sobre o comportamento baseado em eventos da aplicação (no capítulo 3 será apresentado o componente de escalonamento de uma arquitetura para ecrãs públicos).

Cenário 1 – Dia Internacional da SIDA

Ação Internacional – Dia Internacional da SIDA – É o 1.º de Dezembro – Dia Internacional da SIDA. Em toda a Europa decorrem iniciativas que exploram o que significa SIDA para as diferentes comunidades: inclui informação pública de como se pode contrair a SIDA; documentários sobre vários doentes infetados em África; e conteúdo de grupos como escolas ou organizações que angariam fundos de caridade para combater a SIDA. Os cidadãos Europeus estão unidos por uma experiência interativa, criada por um conjunto de artistas internacionais e apresentado em todos os ecrãs participantes de toda a Europa.

No exemplo apresentado os ecrãs são utilizados de forma a partilhar um conjunto de experiências que sensibilizem a população Europeia para o grande problema que é a SIDA.

Como exemplos de requisitos de sincronização, podemos considerar a situação em que uma foto tirada a um grupo de pessoas em frente a um ecrã, é imediatamente difundida por todos os ecrãs participantes. Genericamente podemos considerar situações em que todos os ecrãs deverão sincronizar a disponibilização do mesmo conteúdo.

Cenário 2 – Simpósio com ecrãs informativos

De 27 a 30 de maio de 2030 decorre o simpósio sobre a saúde pública. Durante o evento são debatidos diversos temas sobre a área, o que está feito e o que pode ser feito para melhorar a saúde pública. O evento decorre na Universidade do Minho, contendo vários auditórios com temáticas diferentes a decorrerem em simultâneo. Por todo o campus estão instalados ecrãs que fornecerem um conjunto de informações úteis. Informam o que está a decorrer e onde, indicam direções a tomar de forma a chegar a um determinado auditório, anunciam o fim ou início das apresentações, intervalos, etc.

No cenário descrito a aplicação terá que adaptar o conteúdo apresentado a várias circunstâncias. Tal facto verifica-se porque a aplicação tem como objetivo apresentar informações úteis como: hora de início, hora de fim, indicações do local de almoço ou de *coffe break*. Dessa forma, todos os ecrãs que executem a aplicação devem estar preparados para executar em simultâneo o mesmo conteúdo ou conteúdos personalizados. No caso de simples notificações de início ou fim de uma apresentação num determinado instante, estas serão comuns a todos os ecrãs. Já no caso de se tratar de indicações do local do almoço com setas indicativas, cada ecrã terá que apresentar o conteúdo correto de forma a coincidir com as instruções esperadas.

Cenário 3 – Cenário de emergência

Imaginemos que num determinado centro comercial uma criança é dada como desaparecida. De imediato a mãe da criança entra em contacto com um dos seguranças do centro comercial e fornece-lhe um conjunto de dados sobre a criança. Em poucos segundos todos os ecrãs do centro comercial apresentam uma mensagem, contendo a foto da criança, descrição da roupa que veste, contacto da mãe, entre outros dados. Em poucos minutos alguém identifica a criança descrita num dos ecrãs do centro comercial.

A aplicação de emergência terá garantir suporte à submissão de conteúdos através de várias formas, como por exemplo: SMS, MMS, E-mail, etc. No cenário descrito existe o requisito de que todos os ecrãs do centro comercial apresentem um determinado conteúdo. Há a

necessidade de classificar os conteúdos a serem apresentados, para que os gestores de cada ecrã decidam o que é apresentado primeiro. Nestes casos não poderá ser linear em termos temporais, vistos que podem surgir conteúdos como é caso de uma emergência que se irá sobrepor a qualquer outro conteúdo. Surge, assim, um requisito de sincronização de conteúdos em simultâneo, mesmo que cada ecrã disponha de bases de dados independentes com conteúdos a apresentar. A aplicação gestora do ecrã terá que suportar a introdução de conteúdos externos com prioridade sobre o conteúdo já existente.

2.2. Definição dos requisitos relevantes

Após uma análise aos cenários apresentados foram definidos três requisitos principais:

- Apresentação de conteúdo em simultâneo num conjunto de ecrãs;
- Apresentação de conteúdo num instante pré-definido;
- Reação a eventos;

O primeiro requisito apresentado é comum a cenários de emergência, publicidade, etc. O seu objetivo principal é difundir por um conjunto de ecrãs em simultâneo o mesmo conteúdo (imagens, vídeos, anúncios, músicas, etc.). Utilizando o exemplo de um caso de emergência do cenário número 3, é necessário que com a maior rapidez possível os ecrãs de todo o centro comercial apresentem uma mensagem com uma foto e um número de contacto. Com o passar do tempo o perímetro em que é apresentada essa mensagem deve ser alargado devido à possibilidade da criança sair da zona do centro comercial. Uma vez que estamos num ambiente Web temos um modelo cliente-servidor em que os clientes são todos os ecrãs que executem uma instância de uma dada aplicação. Cada ecrã estará associado a um componente de escalonamento que irá definir o que cada ecrã irá apresentar. Sempre que surjam situações de alerta a aplicação terá que de alguma forma notificar todos os clientes que se manifestaram interessados em tal conteúdo.

No que diz respeito ao segundo requisito, pode imaginar-se um grande número de aplicações que necessitam de executar ou apresentar algo num instante de tempo concreto. A aplicação apresentada no cenário 2 é um bom exemplo disso da mesma forma que o cenário 1 também dispõe desse mesmo requisito. Utilizando o cenário 2, as informações ao longo do dia irão variar consoante a hora, daí a aplicação terá que ser dotada de um mecanismo tal que permita apresentar um determinado conteúdo num determinado instante. Neste caso, a aplicação terá de ser notificada, provavelmente pelo componente de

escalonamento, no instante em que deve apresentar o novo conteúdo. Tal necessidade no cenário em causa advém do facto de ser impossível prever o tempo exato de uma apresentação. Existe uma grande probabilidade de surgirem atrasos, daí não ser possível pré-definir instantes de execução de determinado conteúdo.

O terceiro requisito é a reação a eventos, ou seja, caso aconteça um determinado evento, espera-se que a aplicação execute uma determinada ação. Tal requisito está no cenário 1. Dado que se trata de uma aplicação interativa e permite a participação dos utilizadores, a submissão de novos conteúdos, implicará uma reação por parte de todas as instâncias da aplicação. Quando um utilizador submete um novo conteúdo, essa mesma ocorrência corresponderá a um evento. Cada instância da aplicação (ou até mesmo de diferentes aplicações) deverá estar dotada de um mecanismo que permita detetar o evento e produzir novo conteúdo.

Apesar de terem sido apresentados três requisitos distintos, podemos concluir que corresponde, a diferentes variações do terceiro requisito. Se uma aplicação reagir a eventos, poderá responder a qualquer requisito de sincronização. Tal é possível desde que o evento esteja devidamente classificado de modo que a aplicação possa processar a informação e reagir a esse evento. Dessa forma cada evento será caracterizado pelo seu tipo e pelos seus atributos.

3. Estado da Arte

O presente capítulo deter-se-á, essencialmente, sobre o estado atual dos ecrãs públicos dando especial atenção ao problema da sincronização. A sincronização/coordenação de componentes em sistemas distribuídos e a problemática de sincronização/coordenação entre componentes à escala da *World Wide Web*, nomeadamente o conceito de *Real time Web,* serão, igualmente, objeto de estudo. Finalmente serão enunciadas e descritas, de forma sucinta, alguns exemplos de redes de ecrãs públicos já desenvolvidas, como são o caso do Instant Places [1] [2], e E-campus [3] [4] e UBI-Hotspot [5] [6].

3.1. Redes de ecrãs Públicos

As redes de ecrãs públicos são um paradigma em grande expansão. Atualmente, é possível encontrar ecrãs em vários locais públicos, nomeadamente, em centros comerciais, aeroportos, estações de transportes públicos e ruas principais de algumas cidades. Este progresso veio revolucionar a forma como entendemos a difusão de informação em locais públicos e, essencialmente, em redes de ecrãs públicos. Com a criação de redes de ecrãs abertas, dotadas de mais funcionalidades e mais interativas, torna-se mais cativante o seu uso para o público em geral. Dada a evolução de toda a tecnologia envolvente no paradigma de ecrãs públicos acredita-se que num futuro próximo, seja possível obter redes que possam interagir de uma forma síncrona com dispositivos móveis independentes. Instant Places [1] [2], e E-campus [3] [4] e UBI-Hotspot [5] [6] são exemplos de plataformas de redes de ecrãs públicos.

3.1.1. Arquitetura genérica

Na Figura 3.1 são apresentados os principais blocos e interações para obter uma rede de ecrãs públicos, podendo ser instanciados de diferentes formas e replicados caso necessário [7]. Torna-se imperativo a realização, ainda que breve, de uma descrição da arquitetura das redes de ecrãs públicos, detendo-nos, sobretudo, nos principais aspetos arquiteturais que enquadram o trabalho apresentado neste documento. O ator principal do domínio aplicacional é o programador. Este é responsável pelo desenvolvimento da aplicação, descrição e registo da implementação. O modelo aplicacional ainda é uma questão em aberto. No desenvolvimento deste trabalho, partilhamos o modelo aplicacional fornecido pelo Instant Places, uma infraestrutura de ecrãs públicos específica [2], a qual será descrita

em mais detalhe na secção 3.1.2. Uma aplicação Instante Places é uma aplicação Web que processa e expõe os mais variados conteúdos [8]. As aplicações são baseadas em tecnologias e standards web como o caso de: HTML, JavaScript e CSS. São desenvolvidos em servidores públicos para que possam ser utilizados em qualquer ecrã público. As aplicações são desenvolvidas para que sejam capazes de executar em diferentes *browsers*, diferentes situações, com acesso a variados recursos e diferentes padrões de utilização. Os sistemas Instant Places permitem a integração de aplicações desenvolvidas por terceiros, alojadas em qualquer sítio na internet, tornando assim a arquitetura aberta e escalável. Instant Places oferece um modelo para apresentação de conteúdos que tem em conta a informação local associada ao ecrã, como por exemplo vindo de sensores, interações com utilizadores e conFigurações específicas das aplicações. Esta abordagem permite assim obter um conteúdo bastante personalizado, refletido no comportamento dinâmico das aplicações web de ecrãs públicos [9].

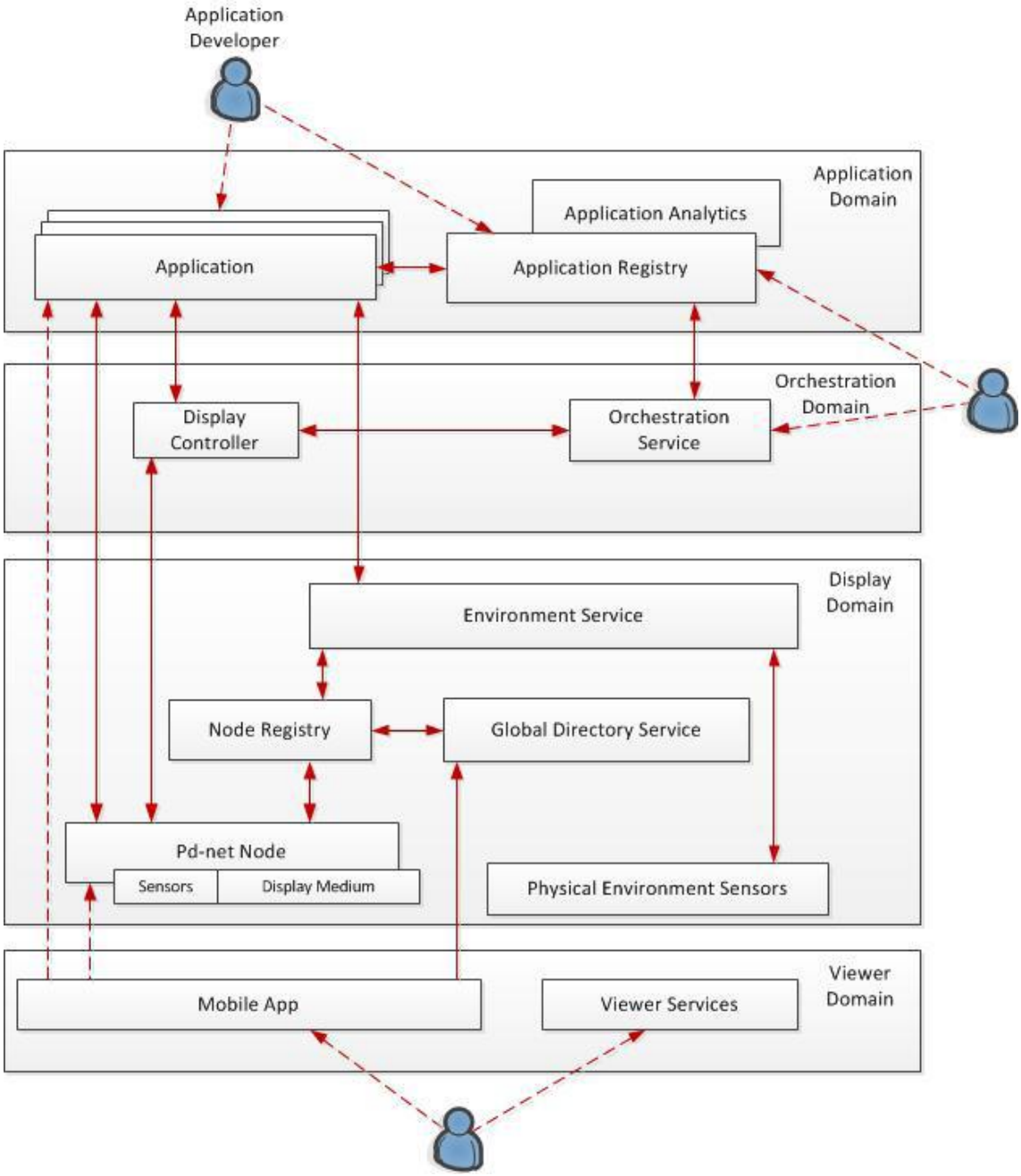


Fig. 3.1 – Arquitetura de uma rede de ecrãs públicos

O registo da aplicação no bloco *Application Registry* é a base para modelo aplicacional distribuído. Fazendo uma analogia com o contexto *Smarth Phones*, *Clinch* apresenta um conjunto de considerações para *applications stores* de aplicações para ecrãs públicos. [10].

Application stores para ecrãs públicos têm uma grande potencialidade para promover atividades colaborativas e sincronizadas em espaços públicos, uma vez que ideias inovadores tendem a surgir em ambientes abertos de inovação.

Os gestores de ecrãs acedem a uma ou várias *applications stores*, registam o ecrã e instalam

as aplicações. Interagem com o *Orchestration Service*, para definir que aplicações estarão disponíveis no ecrã. Quem desenvolve o *software* poderá também definir restrições na instalação e utilização e modo de apresentação das aplicações. Em último caso, os utilizadores também poderão influenciar a forma como os conteúdos são apresentados através de interações locais. A informação de orquestração *(Orchestration)* é a base do *Display Controller* para decidir que conteúdo ou aplicação deve ser apresentado, em que parte do *display* e quando – modelo de escalonamento - deve ser apresentado. Modelos de escalonamento para ecrãs públicos tradicionais, tipicamente multiplexam um conjunto de itens ao longo do tempo e vão alterando o conteúdo apresentado periodicamente. As redes de ecrãs públicos devem ajustar-se a novas formas de escalonamento dinâmico que emerge de conteúdos dinâmicos. Em particular, o problema de sincronização entre aplicações, o escalonamento dinâmico, pode estar associado com a necessidade de reagir a eventos que acontecem em ecrãs diferentes.

Onde é que fisicamente deve estar localizada a aplicação que o ecrã está a executar? é a questão sobre redes em ecrãs públicos que agora se coloca. As potencialidades de redes de ecrãs de larga escala são caracterizadas por uma inerente inovação, mudança contínua do número de utilizadores, detentores de ecrãs, conteúdo produzido, número de nós, itens das aplicações, tipos de conteúdos, modos de interação, sensores e conexões que podem conduzir a grandes problemas de escalabilidade e performance. Neste contexto, técnicas de escalabilidade podem ser utilizadas, em particular em ambientes Web. Podem ser utilizadas técnicas de replicação, ou VM dinâmicas suportadas por *cloudlets* [11].

3.1.2. Instant Places

Atualmente as redes de ecrãs públicos estão em grande crescimento e fazem-se notar, conseguindo captar a atenção do público-alvo. Em muitos casos as redes de ecrãs apresentam aplicações interativas que esperam por contribuições dos utilizadores. Estas podem ser feitas de diversas formas desde o envio de SMSs, MMSs, através de uma ligação por Bluetooth, envio de E-mail, ou interação por toque. No entanto, apesar de existirem muitas formas de interação, nem sempre existem as melhores e mais simples que estimulem a sua utilização. A definição de muitos parâmetros ou sistemas pouco intuitivos podem desmotivar a participação dos utilizadores. Neste contexto, surgiu o projeto denominado de Instant Places, com o objetivo de criar novos conceitos de publicação em ecrãs públicos e interação entre ecrãs e os utilizadores [1] [2]. Foi também criado um mecanismo que

permite controlar a identidade dos utilizadores que participam. O serviço Instant Places define três novos conceitos: os *places*, as *personal identities*, e *display applications*. Um *place* é constituído por um ou vários ecrãs, sendo que representa um cenário simbólico, suficientemente significativo de interações sociais. Uma *Personal Identity* identifica um utilizador, permitindo-lhe controlar o conteúdo exposto. Através da página web do projeto os utilizadores podem criar identidades e posteriormente publicar de conteúdos. As aplicações são as aplicações Web que os gestores do ecrã podem associar ao mesmo. Todas devem estar preparadas para se adaptar às condições impostas e aos recursos existentes e podem aceder a conteúdo partilhado naquele local, através dos recursos locais.

Fig. 3.2 – Exemplo de posters [1]

A título de exemplo, na Figura 3.2 é apresentada uma aplicação que disponibiliza posters/imagens submetidas no local que integra o ecrã público. A aplicação apresenta cada um dos posters com o seu título e o respetivo autor.

Uma visão de alto nível da arquitetura do *Instant places* é apresentada na Figura 3.3.

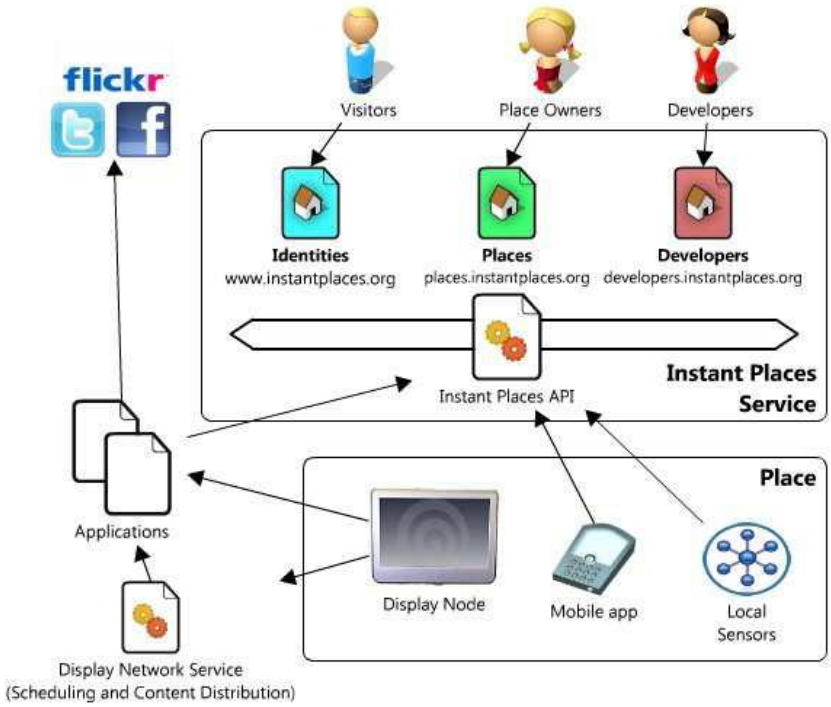


Fig. 3.3 – Arquitetura Instante places [2]

A arquitetura é composta por um conjunto de serviços e a respetiva API (*Application Programming Interface*) que em conjunto oferecem serviços aos gestores de ecrãs, programadores e utilizadores, nomeadamente conFiguração de um local e aplicações desse local, definição de parâmetros de escalonamento, acesso aos recursos locais (como sensores, ecrãs e dispositivos móveis) e abstrações de interação. As aplicações *Instant places* são desenvolvidas por entidades terceiras e encontram-se alojadas em servidores públicos na Internet.

A problemática de sincronização de aplicações não foi abordada no Instant Places. A sincronização ficará sempre ao cargo de quem desenvolve as aplicações. Essa solução é aceitável no contexto mais fechado de publicação de aplicações, mas não se adequa a contextos mais abertos de desenvolvimento e publicação de aplicações.

3.1.3. E-campus

O projeto E-campus foi desenvolvido na Universidade de Lancaster no Reino Unido, com o objetivo de criar uma rede de ecrãs públicos de larga escala [3].

Vários *displays* com diferentes características foram instalados em todo o campus, o que veio a proporcionar um conjunto de instalações interessantes. A primeira instalação decorreu durante o sexto *workshop IEEE* sobre sistemas de computação e aplicações móveis. A instalação é constituída por quatro ecrãs junto as entradas do auditório. O

sistema ia apresentando, ao longo do dia, informações sobre as apresentações, nomeadamente: informações horárias, localizações, informação, ou indicação de outras atividades. A segunda instalação foi desenvolvida na exposição do centro de artes de Brewery. A instalação era constituída por uma apresentação interativa sobre factos ocorridos no local durante a II guerra mundial. Era composta por um conjunto de quatro elementos: conjunto de três ecrãs, um diário em vídeo, um diário web e "Kirlian Table"- uma exibição interativa de arte. Finalmente, a terceira instalação aconteceu na estação subterrâneo de autocarros denominada "Underpass", no campus da Universidade de *Lancaster*. Tinha como objetivo fornecer um conjunto de informações e conteúdo interativo aos passageiros que se encontravam à espera do autocarro. Ao contrário das duas experiencias anteriores, a instalação dos elementos foi feita com a intenção de se manter a longo prazo.

A infraestrutura de ecrãs públicos do E-campus evolui ao longo da realização das três instalações. Cada uma das três instalações apresentava características diferentes, além da localização física dos ecrãs e público-alvo. As principais diferenças focavam-se na disponibilização de conteúdos por entidades terceiras e os requisitos de escalonamento e sincronização.

Na primeira instalação, existia um maior controlo por parte dos gestores da infraestrutura. Os conteúdos poderiam ser criados por terceiros, mas numa fase anterior à realização da instalação. O autor tinha a possibilidade de indicar um conjunto de restrições, nomeadamente temporais ("não inicia antes do Instante T1", "Não termina depois do instante T2", etc), de duração, ecrãs alvo, prioridades e de coordenação entre ecrãs. A partir daqui, o *scheduler* associado a cada ecrã, tenta construir uma "time line", tendo em conta as restrições de cada pedido. No caso de conteúdos com prioridade seria executado um protocolo distribuído de forma a convergirem para uma hora acordada mutuamente. A abordagem escolhida falha em vários pontos. Por exemplo, o *scheduler* não é escalável devido à sua complexidade, sendo que a solução é apenas funcional para um baixo número de ecrãs.

A evolução para a terceira instalação introduziu fortes requisitos de publicação dinâmica de conteúdos. O sistema deveria estar preparado para apresentar conteúdos em determinados instantes e lidar com o conteúdo aleatório, o que torna complexo o escalonamento de conteúdo com a mais variada duração. Neste contexto a "timeline" acordada entre os *schedulers* terá que ser alterada em tempo real e dinamicamente a cada novo conteúdo que

é apresentado. Tal processo torna-se bastante complexo devido à necessidade de manter os vários ecrãs sincronizados entre si. Estes requisitos tornaram-se ainda mais complexos na idealização da terceira instalação, pois sendo esta idealizada para uma longa duração, criava condições para a necessidade de evolução do sistema, tanto a nível de conteúdos como ecrãs. Além disso perspetivava-se já também, a introdução de atores como os gestores de cada local.

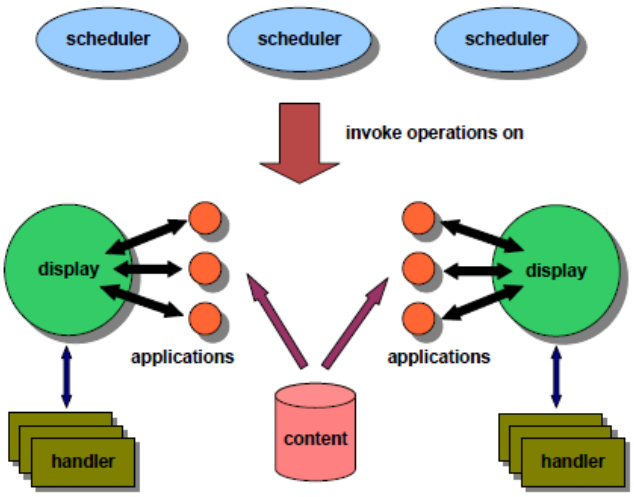


Fig. 3.4 – Modelo computacional E-Campus [4]

Na Figura 3.4 é apresentado o modelo computacional do sistema E-Campus. No sistema E-Campus os principais atores são: as aplicações *schedulers*, ecrãs e *handlers*. Sempre que surge um novo requisito e os *schedulers* existentes não podem dar resposta, é introduzido um novo no sistema. É sua função, através da *scheduling API* gerar e controlar aplicações que processam os conteúdos nos respetivos ecrãs com as devidas restrições definidas. As aplicações são os softwares utilizados nos ecrãs, que processam a informação. Quando é criada uma nova aplicação pelo *scheduler*, o sistema verifica se o conteúdo e os recursos físicos se encontram disponíveis. Por sua vez foram criados os *handlers* de forma a resolver conflitos entre os recursos físicos utilizados. Os atores denominados por ecrãs podem representar vários elementos, sendo possível fazer uma agregação de vários recursos entre ecrãs e computadores.

A comunicação é feita recorrendo a um protocolo de pedido/resposta de uma forma assíncrona através do modelo produtor/subscritor baseado em eventos. O canal de eventos suporta 1:N comunicações. Os componentes subscrevem os eventos do seu conteúdo

identificados pelo par (nome, valor). Posteriormente os eventos são entregues a todas subscrições que correspondem.

3.1.4. UBI-Hotspot

UBI-Hotspot é denominação de uma rede interativa de ecrãs públicos embebida noutros recursos computacionais. Esta rede foi instalada fora de um campus universitário, mais concretamente no centro da cidade de Oulu, na Finlândia. Dessa forma foi obtida uma experiência de computação ubíqua num cenário mais aproximado da realidade, atingindo um maior e mais variado número de utilizadores. Antes da implementação foram inqueridas algumas pessoas, pedindo sugestões de locais para os ecrãs e exemplos de serviços que lhes poderiam ser úteis.

Em termos de infraestrutura de hardware os hotspots estão divididos em dois componentes um para interiores e outro para exteriores. Foram instalados doze *hotsopts* sendo que seis foram em interiores e os restantes seis foram instalados nas ruas do centro da cidade [5] [6].

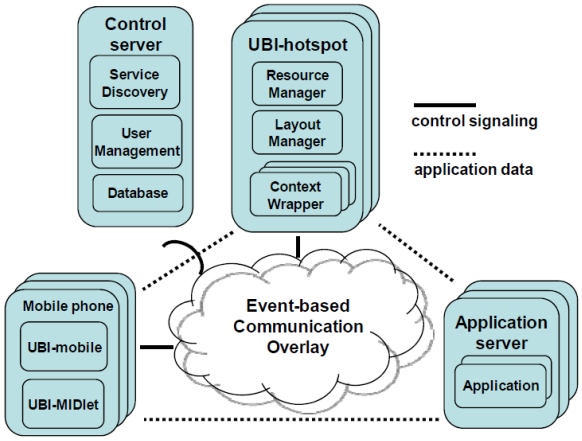


Fig. 3.5 – Arquitetura conceptual de software [5]

Na Figura 3.5 é apresentado a arquitetura conceptual de software, sendo que é constituída por quatro atores: *server control*, *UBI-hotspot*, *aplication server* e *Mobilhe Phone*. UBI-Hotspot é constituído por um ecrã público, colocado juntamente com outros recursos como pontos de acessos sem fios e oferece um conjunto de funcionalidades fornecidas por componentes conhecidos da internet. *Control server* é responsável por um conjunto de mecanismos de gestão de toda a infraestrutura como o exemplo de alojamento de aplicações, registo de utilizadores, etc. *Mobile Phone* é um terminal móvel que pode aceder

aos recursos do sistema e introduzir conteúdos num *Hotspot* em particular. Toda a lógica aplicacional encontra-se no último elemento denominado de *Application Server*. A comunicação entre os vários elementos é efetuada através de uma rede overlay, baseada no modelo de eventos, permitindo a subscrição e publicação de conteúdos relacionados com o contexto em que se encontra. Apesar da arquitetura de *software* ter sido desenvolvido de forma a permitir o seu funcionamento em rede e ao mesmo tempo individualmente, contextualizando as aplicações ao local e contexto em que se encontram os terminais. Para a interação entre os vários elementos foi utilizada a arquitetura *Fuego* [12] que é baseada no conceito produtor/subscritor.

O projeto UBI-Hotspot talvez seja dos mais realistas em redes de ecrãs, dado a sua proximidade com o mundo real. Ao contrário de outros projetos foi desenvolvido fora da "zona de conforto", que seria o meio académico. Além de envolver uma infraestrutura bastante mais complexa devido a sua dimensão, abrange questões nomeadamente como cativar o interesse dos utilizadores e incentivar a sua participação. Dessa forma o seu estudo foi focado maioritariamente à integração dos terminais nos locais, de forma a incentivar a sua utilização. Não sendo feito qualquer estudo a nível de sincronização, nem é abordado qualquer requisito de sincronização.

3.2. Middleware para aplicações distribuídas

3.2.1. Ordem global em sistemas distribuídos

O problema de sincronização entre os componentes de um sistema distribuído diz respeito à sincronização temporal. A sincronização de relógios entre os vários componentes de um sistema distribuído pode ser complexo. Tendo em conta que cada componente terá o seu próprio relógio, facilmente podem surgir desfasamentos entre os vários relógios. Mesmo que num determinando instante os relógios possam estar sincronizados, com o passar do tempo acabam por ficar desfasados entre si, uma vez que a oscilação dos cristais não é a mesma em todos os componentes. Ao contrário de um sistema centralizado, os componentes de um sistema distribuído dispõem de dados e processos independentes. Em certas situações, como por exemplo: bases de dados replicadas, execução em simultâneo ou por períodos pré-definidos, caso não exista uma sincronização entre os vários componentes, poder-se-á originar situações de falha devido à falta de consistência e coerência do sistema.

Para a sincronização de relógios são apresentados vários algoritmos entre eles o algoritmo

de *Cristian*, de *Berkeley* e o protocolo NTP. Os algoritmos de *Cristian* e *Berkeley* destinam-se a redes locais. O protocolo NTP é um protocolo definido à escala da Internet.

O algoritmo de *Cristian* talvez seja o mais simples, sendo que dentro do sistema distribuído existirá um componente que contém o relógio de referência. Periodicamente o componente cliente irá requisitar junto do componente de referência a hora de referência e assim acertar o relógio. Apesar de aparentemente eficiente e simples, o algoritmo dispõem de alguns problemas. Se a hora do cliente for t1 e se a hora do elemento de referencia for t0, sendo t0<t1, não será possível decrementar a hora do cliente. Por outro lado durante o processo de pedido resposta entre os elementos intervenientes, o tempo continua a avançar e é necessário contabilizar esse período da mesma forma que os tempos de processamento utilizados pelos dois componentes.

O algoritmo de *Berkeley* baseia-se num *deamon* que executa em segundo plano, periodicamente questiona todos os componentes do sistema sobre a sua hora atual. Por sua vez, todos os componentes respondem com a sua hora atual. De seguida o *deamon* indica a todos os componentes como devem acertar os seus relógios. De salientar que o sistema irá estar sincronizado entre si com uma hora que poderá não corresponder à hora externa.

O protocolo NTP (Network Time Protocol) é orientado a sistemas à escala da Internet. Trata-se de uma rede de servidores em que sincronizam entre si de uma forma fiável com compensação de atrasos entre pedidos e resposta. Os servidores centrais da rede sincronizam-se utilizando fontes UTC (Universal Coordinated Time), por sua vez os servidores secundários, sincronizam-se com referência nos servidores centrais.

Poderão também ser utilizados algoritmos de ordenação de eventos através de identificadores gerados pelos processos, no entanto apenas será obtida uma ordenação relativa e nunca uma ordenação total. Já que caso sejam identificados com um *timestamp* será difícil entre duas entidade diferentes escolher qual foi o evento gerado em primeiro lugar, caso por exemplo, tenham sido identificados com o mesmo instante.

3.2.2. Produtor – Subscritor

Nos dias de hoje a Internet revolucionou os sistemas distribuídos, pois tornou a sua presença mais notória a uma escala global englobando centenas ou milhares de entidades. Com tal evolução, a versatilidade dos sistemas produtor/subscritor tornou-se numa solução interessante para sistemas de larga escala. De uma forma geral o paradigma produtor/subscritor consiste na manifestação de interesse num determinado tópico,

conteúdo, ou padrão de conteúdo por parte de um componente subscritor e na receção assíncrona do tópico, conteúdo, ou padrão de conteúdo subscrito assim que produzido por um componente produtor. Tem como grande vantagem a comunicação assíncrona de 1:N, N:1 e N:N, dando assim resposta a vários tipos de desafios. O paradigma produtor/subscritor é composto por três elementos sendo eles: o Produtor, o Subscritor e o Serviço de Eventos [13] [14]. O Subscritor é dotado de um mecanismo que permite manifestar o interesse num dado evento ou num grupo de eventos, por outro lado o produtor dispõem de uma funcionalidade que lhe permite publicar um dado evento. Não existe conhecimento entre quem publica (produtor) e quem subscreve (subscritor). O elemento que constitui o elo de ligação entre ambos será o Serviço de Eventos que recebe as subscrições dos subscritores e as publicações dos Produtores. Posteriormente notificará todos os interessados num dado evento quando o mesmo ocorrer. A informação pretendida por uma dada aplicação é denominada de *evento* e o procedimento de entrega do mesmo conteúdo à respetiva entidade é denominado de *notificação*.

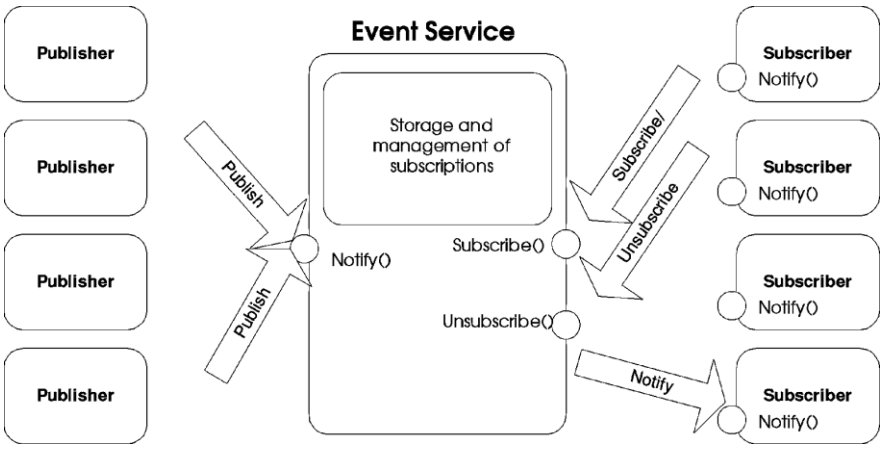


Fig. 3.6 – Sistema Produtor/Subscritor [13]

Na 3.6 é possível verificar os vários elementos intervenientes e as respetivas relações entre os mesmos. Um único subscritor poderá subscrever vários eventos, da mesma forma que vários subscritores poderão subscrever o mesmo evento. Existem inúmeras vantagens a destacar das características de um sistema produtor/subscritor comum como o apresentado na 3.6. No entanto, as que se destacam são o facto de que quem consome não necessita de conhecer quem produz e vice-versa. Outra característica importante deve-se ao facto dos subscritores não necessitarem de questionar periodicamente os produtores sobre um novo evento. Tais vantagens são bastante relevantes para sistemas de grande escala permitindo assim a escalabilidade dos mesmos. A seguinte analogia ilustra como um sistema

produtor/subscritor permite uma maior escalabilidade dos sistemas. “Imaginemos um autocarro com sessenta passageiros, e todos os passageiros perguntam periodicamente se já chegaram ao destino”. Tal problema tornaria a viagem dolorosa e insustentável para o motorista. A solução poderia passar por um elemento intermediário, por exemplo um display que apresentaria um tempo estimado para a chegada. Dessa forma todos os ocupantes do autocarro estariam informados e o motorista apenas teria que periodicamente atualizar o valor do display.

O sistema produtor/subscritor dispõem de três variantes, sendo elas *Topic-based*, *Content-based* e *Type-based*. Num sistema *topic-based* os produtores publicam os eventos em tópicos individuais que são identificados através de palavras-chave. Os subscritores recebem todos os eventos dos tópicos que subscreveram. Os produtores são responsáveis por definir os vários tópicos (palavras-chave) que os subscritores poderão subscrever.

Nos sistemas *content-based* os eventos serão entregues aos subscritores apenas se respeitarem as condições impostas pelos mesmos. No caso de sistemas *topic-based* os eventos são filtradas segundo critérios externos, por outro lado, os sistemas *content-based* dependem de características internas e particulares do próprio evento, como nome de variáveis, valores das mesmas, etc. Tais filtros são implementados com linguagens como XPath ou SQL, por exemplo.

Um sistema *type-based* é um sistema híbrido que utiliza as duas outras variantes. Neste caso os produtores publicam a informação segundo determinados tópicos que são definidos pelos mesmos. Por outro lado os subscritores subscrevem os tópicos especificando que tipo de conteúdo esperam receber dos mesmos. Ou seja estabelecem um conjunto de condições que o evento com origem num determinado tópico deve reunir.

3.3. Real-time web

O conceito *real-time web* [14] tem com objetivo difundir alterações de conteúdo Web de forma expedita. Pode também ser definido como um conjunto de mecanismos capazes de entregar às entidades interessadas, um determinado conteúdo acabado de produzir pelo autor. Aplicações com necessidades de atualização constante são um bom caso de utilização desta tecnologia. Páginas de notícias, blogs, páginas de leilões, redes sociais entre outros são só alguns exemplos em que existe necessidade de atualização de conteúdos. Em alguns casos os autores podem até nem conhecer quem está interessado nos seus conteúdos.

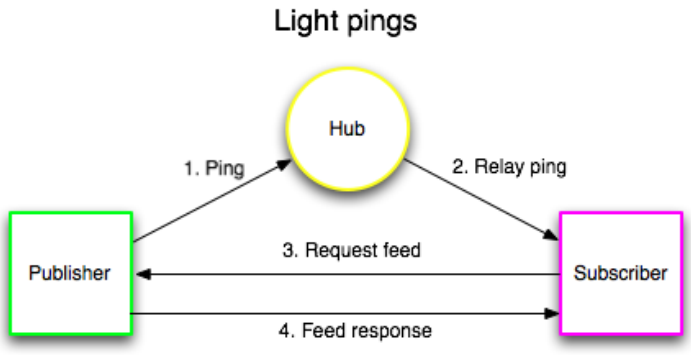
Perante as atuais necessidades de aplicações em tempo real, surge então o conceito de *feed*.

Um *feed* não é mais do que uma lista de atualizações de uma dada página. Cada página, blog ou outro tipo de aplicação web associa a si um *feed* em forma de um link. O utilizador através de um agregador de *feeds*, poderá subscrever um conjunto de *feeds*. Desta forma o agregador/leitor de *feeds* irá questionar periodicamente as fontes se contêm novas atualizações. Os arquivos *feeds* são ficheiros descritos em XML. Atualmente existem três especificações para a criação de ficheiros *feeds*: RSS 1.0, RSS 2.0 [15] e Atom [16].

3.3.1. Sistemas Produtor/Subscritor

No presente capítulo serão analisados alguns sistemas produtores/subscritores (publish/subscribe) à escala web. A escolha de sistemas à escala web deve-se os requisitos impostos pelo projeto descritos no capítulo 2. Sistemas produtores/subscritores locais não são escaláveis, não sendo assim possível obter redes de ecrãs globais e independentes. Desta forma, a nossa análise deter-se-á apenas nos sistemas orientados segundo o modelo web. A informação de alteração de conteúdos é modelada através de web *feeds*, que são disponibilizados por um URL. Nos exemplos apresentados ao longo do presente capítulo o serviço de eventos será denominado por *Hub*.

Atualmente existe um conjunto de sistemas produtores/subscritores (pubsub) dotados de um conjunto de mecanismos que permitem manter os subscritores atualizados sobre alterações feitas pelos produtores (*publishers*). Estes sistemas podem ser do tipo *Light pings* ou *Fat pings* [17]. Quando um produtor gera uma nova atualização informa o Hub (Serviço de eventos) dessa atualização. Por sua vez o *Hub* informará o subscritor da existência dessa atualização através do *Url* da mesma. Tendo assim o subscritor que contactar diretamente o produtor para obter o conteúdo, este sistema é denominado de *Light ping*. O seu funcionamento encontra-se ilustrado na 3.8.



[17]

Fig. 3.7 – Ilustração de uma arquitetura light ping

Num sistema *Fat ping* o próprio *Hub* obtém o conteúdo junto do produtor e envia o

conteúdo diretamente para o subscritor. Na Figura 3.9 é possível verificar o funcionamento do protocolo PubSubHubbub [18] que é um exemplo de um sistema *fat ping*.

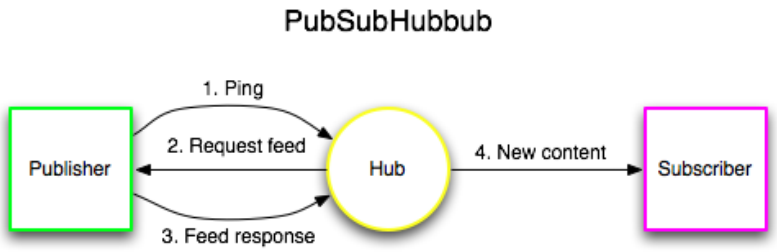


Fig. 3.8 – Ilustração de uma arquitetura fat ping [17]

Além do PubSubHubbub outro exemplo de um protocolo *fat ping* é XMPP pubsub. Como exemplo de sistemas *light ping*, podemos apresentar também o sistema rssCloud, SUP e SLAP. Seguidamente será feita uma análise a alguns dos exemplos apresentados.

XMPP pubsub (XEP-0060) [19] é um protocolo produtor/subscritor. Contém uma entidade central com capacidade de receber um conjunto de subscrições e ao mesmo tempo receber notificações dos produtores e posteriormente encaminhar o novo conteúdo para os subscritores interessados. A entidade central que desempenha tais funções é denominada de "nó". Além das funções típicas de mediador de um sistema produtor/subscritor, dispõem também de um histórico dos eventos ocorridos e de outros serviços que complementam o modelo original. O princípio em que assenta o seu funcionamento é básico. Os subscritores/utilizadores manifestam o seu interesse num determinado conteúdo junto de um nó. Da mesma forma que os produtores publicam no nó todas as alterações surgidas. O nó por sua vez encaminhará o respetivo conteúdo para os subscritores interessados.

SUP (Simple update protocol) [20], consiste num protocolo que dispõem de um conjunto de mecanismos que permite aos produtores informarem os subscritores que existem novas alterações ao conteúdo subscrito. Neste caso, as interações são efetuadas através de *polling*, sendo que os clientes (subscritores) periodicamente questionam os produtores se existem alterações. O protocolo dispõe de um documento denominado de "documento de alterações (*updates*)" que é um objeto do tipo JSON [21]. Um subscritor questiona um produtor se num determinado período surgiu alguma alteração ao conteúdo subscrito. Se não surgiram alterações o documento de alterações estará vazio, por outro lado, se existirem alterações estas estarão descritas no documento de alterações. Desta forma, é evitado o mecanismo de

polling e simultaneamente, uma vez que é enviado apenas o conteúdo que foi alterado e não os recursos na sua forma integral, minimiza o aumento de tráfego e consequentemente um menor uso da largura de banda.

O protocolo SLAP (para *Simple Lightweight Announcement Protocol*) [22], de uma forma similar aos restantes protocolos, tem como objetivo notificar os subscritores de novo conteúdo ou de alterações nos produtores, sem utilizar o mecanismo de *polling*. Ao contrário de outros protocolos, o SLAP não é orientado à conexão e utiliza o protocolo UDP. O protocolo anuncia as alterações em relação aos *feeds* que se encontram identificados por um URI e aos *posts* associados a esses mesmos *feeds*. O protocolo funciona de uma forma simples, visto que não é orientado à conexão. Os produtores criam um *feed* de conteúdos. Sempre que o *feed* for alterado, será gerado um evento SLAP que é enviado para cada subscritor. Os subscritores que recebam o anúncio devem responder com um *acknowledgement*. Os produtores irão reenviar as mensagens de anúncio até receberem uma resposta afirmativa ou expirar um período de repetição. O protocolo não considera relevante a perda de alguns anúncios. Por outro lado apenas tem como objetivo anunciar que existe um novo conteúdo ou que foi alterado. Não contém mecanismos de envio do conteúdo integral. Apenas fornece os endereços onde os subscritores podem obter o conteúdo desejado.

O Weblogs.com [23] é um serviço de notificações que automaticamente informa os subscritores quando o conteúdo de um *blog* ou de uma página é alterado. De uma forma genérica, à semelhança de outros protocolos, funciona como uma entidade central entre os subscritores e os produtores. Neste caso concreto é denominado de *Hub*. Os produtores quando dispõem de novo conteúdos publicam no *Hub* que por sua vez se encarregará de entregar esse mesmo conteúdo aos subscritores que tenham manifestado interesse em tal conteúdo. Permite assim uma maior eficiência e rapidez na distribuição de conteúdos, baixando o atraso ocorrido entre a ocorrência de um alteração e a entrega da notificação da mesma. O serviço está disponível em duas interfaces distintas: REST (**Re**presentational **S**tate **T**ransfer) e XML-RPC.

PubSubHubbub [18] é um protocolo produtor/subscritor. Os conteúdos são organizados por tópicos, sendo que cada tópico tem associado a si um URL. O protocolo define três elementos: os produtores, os subscritores e o *Hub*. O *Hub* é o elemento intermediário entre os produtores e os subscritores. Sempre que existir um novo conteúdo para publicar, este será publicado no *Hub*. Da mesma forma que quando um subscritor pretende manifestar

interesse de um determinado conteúdo (tópico), este irá fazê-lo junto do *Hub*. Desta forma, os componentes de um sistema distribuído não necessitam de se conhecerem uns aos outros, promovendo a escalabilidade do sistema. Tal característica advém do facto do próprio *Hub*, após ser notificado da existência de um novo conteúdo, obter este conteúdo junto do produtor. Posteriormente, o *Hub* encaminha o conteúdo para o subscritor. Noutros protocolos, o *Hub* apenas notifica os subscritores sobre novas alterações e posteriormente o subscritor terá que obter essas alterações junto do produtor. Permite assim uma redução de tráfego e uma redução do atraso entre a ocorrência de novo conteúdo e a notificação do subscritor de tal ocorrência. A comunicação entre o *Hub* e os subscritores é baseada num protocolo de *multicast* de forma a difundir simultaneamente as alterações ocorridas por todos os subscritores. Além do protocolo de *multicast*, são utilizadas estratégias de otimização adicionais Por exemplo, se um subscritor tiver subscrito diferentes, poderá receber o conteúdo respetivo numa única notificação. O protocolo de comunicação utilizado entre os vários elementos é o HTTP e o conteúdo das mensagens é modelado em RSS/Atom [Anexo A]. No capítulo 4 o protocolo PubSubHubbub será analisado em maior detalhe e apresentadas as razões para a sua escolha.

3.4. Análise

As redes de ecrãs públicos irão partilhar as características mais importantes dos sistemas distribuídos de grande escala: serão descentralizados de várias formas, desenvolvidas, e utilizadas por uma variedade de entidades, em diferentes contextos, com diferentes necessidades de conexão e em constante expansão. O tipo de aplicações que poderão utilizar as redes de ecrãs são os mais variados, desde aplicações informativas, aplicações públicas de emergência, aplicação privadas difundindo publicidade e até mesmo jogos que permitam ocupar pessoas em salas de espera. Um dos pontos fundamentais deste tipo de sistema é a autonomia na instalação e conFiguração de cada ecrã. Cada instalação será provavelmente gerida por diferentes entidades e poderá não partilhar nenhuma componente funcional da rede de ecrãs com exceção da *application store* e, em alguns casos, do servidor web onde se encontram alojadas as aplicações. Ainda que algum tipo de *schedulers* poderá estar preparado para gerir um conjunto de ecrãs, cumprindo assim os requisitos de sincronização enunciados no capítulo 2 deste documento [4], é nossa convicção de que esta abordagem poderia violar a autonomia das instalações de redes de ecrãs públicos.

No nosso trabalho abordamos a problemática de sincronização entre aplicações em ecrãs e

desenvolvemos um *Framework* baseado em ferramentas, serviços e protocolos do domínio Web que preserva as características principais das redes de ecrãs públicos, mais concretamente a autonomia entre as diferentes instalações de ecrãs públicos e a escala ao nível da Web. Desta forma, optámos pelo sistema PubSubHubbub, um sistema produtor/subscritor, escalável ao contexto da Web e adequado à comunicação entre componentes de um sistema distribuído geridas por entidades diferentes e a executar em contextos arquiteturas independentes.

Soluções baseadas em *middlewares* para sistemas distribuídos como por exemplo protocolos ou algoritmos para sincronização de relógios ou eventos temporais mostraram-se inadequadas, como podemos ver em [4] uma vez que se baseiam numa forte interligação entre os componentes (aplicações) e os *schedulers*, não sendo adequados a sistemas de grande escala.

4. O sistema PubSubHubbub

4.1. Descrição geral

PubSubHubbub é um sistema que integra um protocolo de comunicação remota segundo o paradigma produtor/subscritor, aberto, à escala da World Wide Web e a uma implementação de referência direcionada para o motor de aplicações do Google [18]. A comunicação entre dois componentes de um sistema distribuído, normalmente designados de produtor e subscritor, é efetuada através do protocolo HTTP. Os dados das mensagens são representados através dos formatos RSS e Atom [15] [16]. PubsubHubbub é adequado às aplicações que necessitem de manter constantemente atualizada a informação sobre uma determinada fonte. Permite manter conteúdos atualizados de várias entidades em simultâneo, sem necessidade dos subscritores questionarem os produtores se efetivamente dispõem de conteúdo novo. O mecanismo em que subscritores de informação questionam periodicamente os produtores dessa informação sobre se dispõem de conteúdo novo é denominado por *polling*. Esta situação implica uma maior largura de banda, grandes atrasos na entrega das notificações, necessidade de maiores recursos físicos, resultando assim em sistemas não escaláveis. Este sistema introduz um terceiro componente na comunicação remota entre produtores e subscritores: o *Hub*. O *Hub* será o elo de ligação entre ambos permitindo assim que não exista a necessidade de um conhecimento total da arquitetura por parte de todos os componentes do sistema. Sempre que um subscritor (*subscriber*) está interessado num determinado conteúdo, irá manifestar o mesmo interesse junto do *Hub*. Por outro lado, sempre que um produtor dispõe de novo conteúdo, irá notificar o *Hub* de tal ocorrência. Por sua vez o *Hub* irá obter junto do produtor o conteúdo e entregar esse mesmo conteúdo a todos os componentes que o subscreveram. Desta forma, obtemos um protocolo descentralizado, sendo que o número de *Hubs* poderá ser ajustado ao sistema, e não existe conhecimento direto entre produtor e subscritor evitando necessidade de grandes recursos em ambas partes.

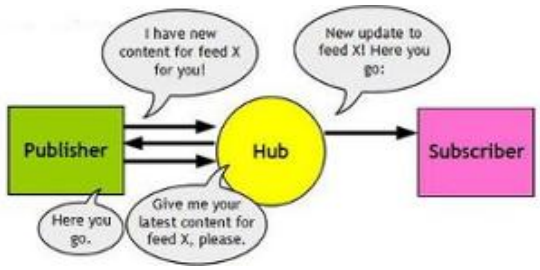


Fig. 4.1 – PubSubHubbub [18]

Na Figura 4.1 é possível verificar as interações básicas entre os vários elementos intervenientes no protocolo PubSubHubbub. Pode constatar-se que o produtor (*Publisher*) não conhece o subscritor (*subscriber*) e vice-versa. Tal como indicado anteriormente o produtor informa o *Hub* que contém novo conteúdo, por sua vez o *Hub* pede-lhe o novo conteúdo, que posteriormente envia para o subscritor (*subscriber*).

4.2. Descrição do protocolo

4.2.1. Funcionamento e elementos intervenientes

O sistema PubSubHubbub é um sistema produtor/subscritor, constituído por três elementos, sendo eles: Produtor (*Publisher*), Subscritor (*Subscriber*) e *Hub*. O produtor produz informação para o sistema, sendo esta consumida pelos componentes subscritores. O *Hub* funciona como mediador entre ambos, tal como se pode verificar na Figura 14.

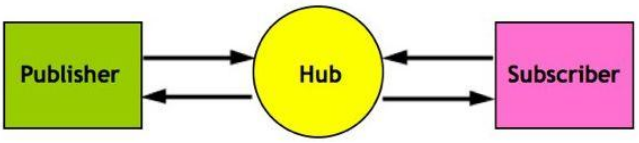


Fig. 4.2 – Elementos presentes no PubSubHubbub [18]

Um produtor sempre que dispõem de novo conteúdo informa o(s) Hub(s) através de um *Web feed*. Um *Web feed* é um formato de dados que permite compilar as várias alterações ocorridas numa determinada página ou blog. É frequentemente utilizado em páginas com atualizações constantes. *Web feed* é um ficheiro XML, contendo todas as alterações efetuadas por uma determinada entidade. Esse mesmo ficheiro é disponibilizado através de um URL, permitindo assim a terceiras entidades subscreverem as alterações através de um agregador de *feeds*. Atualmente existem duas especificações de web feeds, sendo elas Atom e RSS. Dado que os conceitos de *web feed*, Atom e RSS não se encontram no âmbito do presente projeto, é possível encontrar mais esclarecimentos no anexo A.

Num sistema topic-based como é o caso do PubSubHubbub, um *web feed* irá conter as

alterações de um determinado tópico. Por sua vez o URL de um determinado *web feed* será denominado de *topic URL*. Um qualquer subscritor que esteja interessado num determinado tópico, poderá aceder às mesmas efetuando a subscrição do tópico disponível através do endereço *topic URL*.

Todas as interações entre os vários intervenientes serão efetuadas com a utilização do protocolo HTTP. Quando o produtor informa o *Hub* de uma nova atualização, não indica o que foi alterado nem tem qualquer conhecimento de quem são os subscritores das suas atualizações, isso faz com que o produtor seja apenas uma fonte de informação. Quando um *Hub* é notificado pelo produtor que contém novo conteúdo, irá enviar-lhe um HTTP GET de forma a obter o novo conteúdo. Em situações normais será sempre o produtor a informar o *Hub* da ocorrência de alterações num determinado tópico. No entanto, caso o produtor não informe o *Hub* sobre a ocorrência de novos eventos durante longos períodos, este pode questionar o Hub sobre novas atualizações através do mecanismo de *polling*. Na Figura 4.3 é possível ver uma ilustração do processo de publicação de um conteúdo. Apesar de na Figura apenas aparecer um *Hub*, o produtor poderá publicar os seus conteúdos em vários *Hubs*. Por outro lado, os *Hubs* podem pertencer a uma entidade pública ou podem também ser *Hubs* privados, pertencentes associados à aplicação produtora. Como já foi descrito no secção 3.3.1 o PubSubHubbub é um sistema *fat ping*, ou seja, obtém o conteúdo junto dos produtores e posteriormente entrega esse mesmo conteúdo junto dos subscritores. Dessa forma evita mais interações entre os vários intervenientes, (Produtor, Subscritor, Hub e feed). Ao contrário de uma implementação light ping, especificada no subcapítulo 3.3.1, que apenas fornece ao subscritor o URL do *web feed* que foi atualizado.

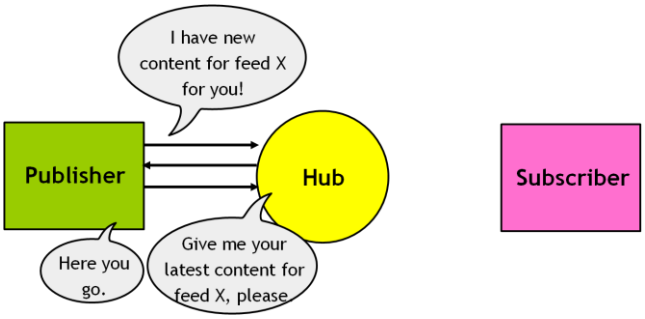


Fig. 4.3 – Exemplo de publicação de conteúdo [18]

Dado que se trata de um sistema *fat ping*, o *Hub* após ser notificado pelo produtor, irá obter o conteúdo junto dele, tal como ilustrado na Figura 4.3. Só após obter a informação é que enviará a mesma aos subscritores do *Web feed* correspondente ao tópico que foi alterado, tal como apresentado na Figura 4.4.

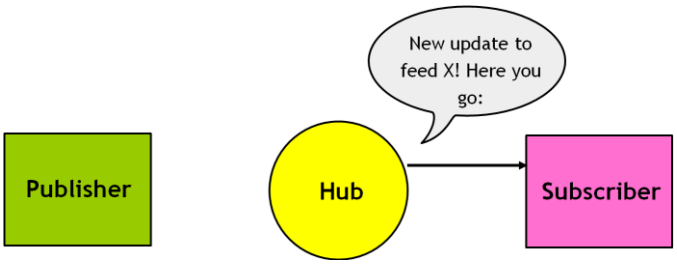


Fig. 4.4 – Envia de dados do Hub para o subscritor [18]

Para tal ser possível, é necessário que anteriormente já tenha sido efetuada uma subscrição por parte do subscritor.

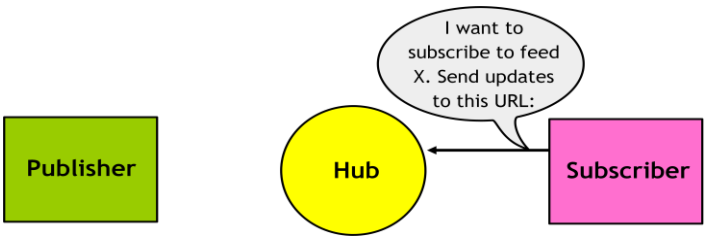


Fig. 4.5 – Ilustração de subscrição [18]

Na Figura 4.5 é possível verificar a ilustração do primeiro passo de uma subscrição, ou seja o subscritor manifesta junto do *Hub* o seu interesse num determinado *Web feed*. Os subscritores terão que estar acessíveis através da internet.

Uma subscrição é um POST HTTP enviado para o endereço do *Hub*. O corpo da mensagem HTTP é composto pelos seguintes parâmetros:

- *CallBack URL* – é obrigatório e é o endereço para o qual o Hub envia as notificações;
- Mode – é obrigatório e indica se pretende subscrever ou remover a subscrição;
- Topic url – é obrigatório e é a descrição do que pretende subscrever;
- Verify – é obrigatório e suporta verificação de uma forma síncrona ou assíncrona;
- Lease_seconds – é opcional e corresponde ao número de segundos que o subscritor pretende ter o pedido ativo;
- Secret – é opcional e corresponde a um conjunto de caracteres que possibilitam o cálculo do HMAC;
- Verify_token – é opcional e é um símbolo "opaco" que permite ao subscritor verificar quando é iniciada a verificação.

O "CallBack Url" identifica o componente do subscritor que recebe as notificações das

atualizações dos *Web feeds* subscritos. Deste modo terá de estar acessível publicamente. Este endereço terá de ser verificado pelo *Hub*, de modo a garantir ao sistema que a subscrição foi efetivamente efetuada pelo componente que disponibiliza o "callback url". Este mecanismo está representado na Figura 4.6.

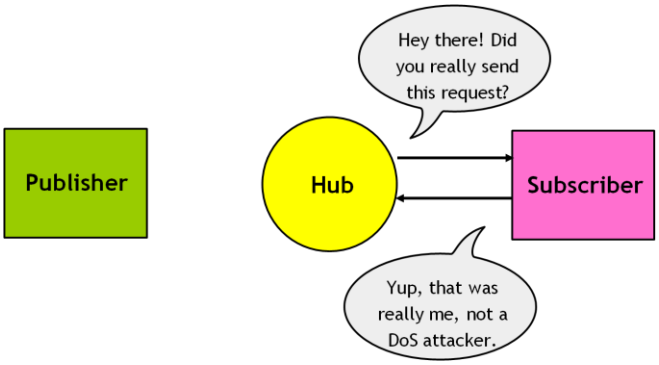


Fig. 4.6 – Ilustração do envio do desafio lançado pelo Hub e respetiva resposta [18]

Como é descrito na Figura 4.6, o *Hub* submete um desafio ao "Callback Url" de forma a verificar se de facto foi esse subscritor que efetuou a subscrição. O desafio é uma agregação de um conjunto de informações relativas ao pedido de subscrição enviada ao subscritor. O subscritor, caso seja o responsável pela subscrição descrita pelo desafio, deverá responder afirmativamente ao *Hub*. No caso do parâmetro *mode*, indica se o subscritor está subscrever ou a remover a subscrição, sendo os modos representados por *subscribe* e *unsubscribe*, respetivamente. No caso do parâmetro *topic url*, corresponde ao endereço url do web feed do tópico correspondente. Ao efetuar a subscrição, o subscritor deverá indicar que tipo de verificação é que suporta, que é representado pelo parâmetro *verify*. No caso de suportar verificação antes da resposta do *Hub* ao pedido de subscrição, devera indicar *sync*. Já no caso de suportar apenas após a resposta do *Hub* deverá indicar o valor *async*. No entanto alguns *Hubs* que suportem os dos modos de verificação, poderão repetir este parâmetro pela ordem preferencial de verificação. O parâmetro *lease seconds* por sua vez indica o tempo total em segundos, que o subscritor pretende que a subscrição se mantenha ativa. Dado que se trata de um parâmetro opcional, caso não seja preenchido, a subscrição ficará ativa permanentemente, até ao *Hub* determinar que já não será útil para o subscritor. O parâmetro *secret* é opcional é composto por uma string que permite o cálculo de uma assinatura HMAC (Hash-based Message Authentication Code). A assinatura é colocada nos cabeçalhos dos vários pedidos de subscrição. Dessa forma garante a autenticidade de conteúdos. O parâmetro verify_token é apenas um símbolo que o subscritor fornece ao

Hub, que posteriormente o encaminha de novo para o subscritor no momento que efetuar a verificação. Dessa forma o subscritor fica a saber do início da verificação. Dado que se trata de um parâmetro opcional, caso não seja indicado, o Hub não enviará nenhum outro símbolo.

4.2.2. Códigos de estado

Dado que a comunicação é efetuada através de HTTP [24], as respostas resultantes de cada operação utilizam os códigos de erro HTTP. Na Tabela 4.1 pode verificar-se a associação entre códigos HTTP e respetiva descrição.

Código	Descrição
1XX	Resposta provisória/informativa
2XX	Bem-sucedido
3XX	Redirecionado
4XX	Erro de solicitação
5XX	Erro no servidor

Tabela 4.1 – Códigos HTTP [24]

Quando é efetuada uma subscrição e esta fica ativa, o *Hub* devolve o código HTTP 204 que corresponde a "sem conteúdo". Por outro lado se a subscrição ainda não tiver sido ativa, ou seja, o pedido foi feito em modo *assíncrono*, o *Hub* devolve o código 202. Se eventualmente for detetado qualquer erro no processo, a resposta será das gamas de valores 4XX ou 5XX. No caso de ocorrer um erro o *Hub* retorna no corpo da mensagem de resposta a descrição do erro. Os *Hubs* têm autonomia para rejeitar determinados subscritores devido às suas próprias políticas. De forma a evitar ataques ou subscrições fraudulentas, sempre que é recebida uma nova subscrição o *Hub* envia um HTTP GET ao "callback url" com os parâmetros da subscrição de forma a garantir que de facto é um pedido válido. Se os parâmetros enviados pelo *Hub* ao subscritor corresponderem ao pedido enviado ao *Hub*, é porque está correto e responde de forma positiva com o código HTTP 2XX. Caso contrario, é devolvido o código de erro 404. O produtor dispondo de atualizações informa o *Hub* através de um HTTP POST, sendo que no mesmo POST poderá conter várias atualizações, ou seja, na mensagem irá ter vários tópicos. A alteração de um determinado conteúdo, poderá afetar um conjunto de tópicos subscritores. Neste caso, o *Hub* dispõe de um mecanismo que permite analisar a ocorrência e verificar a que tópicos

correspondem. O *Hub*, de forma a maximizar a eficiência das notificações aos subscritores, poderá agregar um conjunto que contenha em comum o mesmo destinatário.

4.3. Análise

Pretende-se utilizar um sistema produtor/subscritor que dê suporte a uma API de sincronização para aplicações em ecrãs públicos. Existem uma série de requisitos necessários para o seu bom funcionamento. Os principais exemplos são:

- Latência reduzida
- Garantia de entrega
- Segurança
- Simplicidade
- Escalabilidade

Nenhum dos requisitos enunciados pode ser descartado, no entanto dado o objetivo da API, terá que ser dada especial atenção à escalabilidade e latência reduzida. A API segue o modelo web escalável/distribuído e como tal o protocolo web selecionado também o deve seguir. É necessário que o protocolo garanta a escalabilidade do sistema, caso contrário a API falhará. Analisando os vários exemplos de protolocos apresentados na secção 3.3.1, todos os que pertençam ao grupo light ping estão excluídos. Tal facto deve-se a que como foi indicado, a sua principal característica é o *Hub* em vez de entregar o conteúdo da alteração, apenas entrega o URL do *web feed* que foi alterado. Implica que cada subscritor contacte o produtor para obter as atualizações, isto deita por terra a possibilidade de ter um sistema escalável e introduziria imensos atrasos. Assim sendo restam os protocolos PubSubHubbub e XMPP pubsub. Os dois protocolos restantes apresentam um conjunto de características em comum, como a garantia de baixa latência, distribuição através de *Push* em vez de polling, possibilidade de conter notificações seguras. Contudo, a escolha do XMPP tornou-se inviável pelo formato de dados utilizado. (XMPP). PubSubHubbub por sua vez utiliza como formato Atom e Rss que permitem uma fácil modelação e geração do respetivo ficheiro de dados. Por outro lado, o facto de o PubSubHubbub utilizar HTTP, como protocolo de transporte garante uma maior facilidade no desenvolvimento dos produtores e subscritores. Apenas necessitarão de métodos simples que possam enviar e receber pedidos HTTP.

5. Framework de sincronização para aplicações em ecrãs públicos

5.1. Descrição geral

A Framework desenvolvida no âmbito deste projeto apresenta como principais componentes *Hubs* e as aplicações web. Os *Hubs* são servidores web que permitem agregar conteúdo gerado pelos produtores e difundir "instantaneamente" por todos os subscritores que subscreveram tal conteúdo. As aplicações poderão ser produtoras ou subscritoras de conteúdos ou ambos os casos em simultâneo. Poderão ser do mais variado tipo, utilizando uma qualquer Framework web, no entanto a API desenvolvida no decorrer deste projeto é em PHP. A base de comunicação entre aplicações assenta no modelo de eventos. Se as aplicações produtoras devem modelar, criar e publicar o evento as aplicações subscritoras devem modelar, subscrever e processar o evento recebido. Um evento é caracterizado por um conjunto de parâmetros, que permitem identificar toda a informação que possa ser relevante para as aplicações subscritoras, parâmetros esses que poderão ser textos, datas, números, etc. A API de Sincronização fornece um conjunto de mecanismos que permitem efetuar todas as operações apresentadas anteriormente.

5.2. Descrição geral da API

A API de sincronização consiste numa biblioteca PHP e JavaScript que oferece um conjunto de métodos que dão resposta a um conjunto de requisitos de sincronização. Dado que foi adotado um modelo baseado em produtor/subscritor a API terá que dispor de dois métodos básicos que permitem publicar e subscrever. Além dos métodos básicos são também necessários dois métodos para a modelação e processamento dos eventos. Por outras palavras, são necessários métodos que permitam criar e receber os eventos. Por isso a API de sincronização dispõe do seguinte conjunto de métodos:

- create_event()
- publish_event()
- subscribe_event()
- setInitInfo()
- waitForEvent()

O método criar evento - create_event() - permite às aplicações, ou às suas componentes, criar a descrição de um evento para ser publicado em um ou vários Hubs. Um evento é descrito por um conjunto de parâmetros, sendo que esses parâmetros podem ser de qualquer tipo (array, integer, char, etc). Esse mesmo conjunto de parâmetros compõem um array php que é o único argumento que o método recebe.

```
function creat_event($parametros)
```

Fig. 5.1 – Protótipo do método create_event()

Na Figura 5.1 é representado o protótipo do método create_event(), tal como já referenciado recebe como argumento um *array* de parâmetros denominado de *$parametros*. A execução do método devolve um booleano (*true* ou *false*) que define respetivamente o sucesso ou insucesso da sua execução. Sucesso da operação implica que tenha sido adicionado uma nova entrada ao ficheiro eventos correspondente.

O método publicar evento publish_event() permite às aplicações, ou às suas componentes, publicar um ou vários eventos num Hub.

```
public function publish_event($topic_url, $hub_url)
```

Fig. 5.2 – Protótipo do método publish_event()

Através do protótipo do método *publish_event()* apresentado na Figura 5.2 é possível verificar que o método recebe dois argumentos. Sendo que o argumento *$topic_url* representa o endereço do *web feed* que se pretende publicar. O argumento *$hub_url* representa o endereço do *Hub* onde se pretende publicar os eventos. À semelhança do método *create_event(),* o método *publish_event()* tem como resultado um booleano que representa o sucesso ou insucesso da operação.

O método subscrever evento *subscribe_event()* permite às aplicações subscrever os eventos publicados por outras aplicações e consequentemente terem acesso ao conteúdo produzido por essas aplicações.

```
public function subscribe_event($hub_url, $callback_url, $topic_url,
                                 $lease_seconds)
```

Fig. 5.3 – Protótipo do método subscribe()

Na Figura 5.3 é apresentado o protótipo do método *subscribe_event()* e através do mesmo pode verificar-se que o método recebe quatro argumentos. Dos quatro argumentos dois deles são os mesmos já apresentados em métodos anteriores. Dos argumentos adicionais o callback_url corresponde ao endereço onde é pretendido receber as notificações (endpoint),

das subscrições efetuadas. O quarto argumento corresponde ao período em que a subscrição se irá manter ativa (em segundos).

```
function setInitInfo(end, tipo, envia, vars)
```

Fig. 5.4 – Protótipo do método setInitInfo()

Na Figura 5.4 pode verificar-se o protótipo do método *setInitInfo()* que recebe como parâmetro: endereço do *endpoint*, tipo de evento, endereço para onde são enviados os dados e o array de conteúdos que pretendemos obter, pela ordem respetiva. O argumento do endereço do *endpoint* refere-se à script *endpoint.php* fornecido pela API de sincronização. Em relação ao tipo refere-se a que tipo de evento é, uma aplicação poderá gerar e subscrever um conjunto de eventos de tipos diferentes associados a um só web feed. Contém também o endereço para onde são encaminhadas as notificações recebidas, após a devida filtragem. O *array* de conteúdos é um conjunto de parâmetros que a aplicação pretende saber sobre o tipo de evento subscrito. O protocolo produtor/subscritor escolhido, é baseado em *topic-based*, o que implica que todos os eventos ocorridos que correspondam ao tópico serão entregues aos subscritores. Ainda assim, os vários subscritores podem não pretender os eventos todos e surge a necessidade de aplicar filtragem aos conteúdos. Facilmente se poderão imaginar cenários, em que nem todas as notificações serão úteis a todos os subscritores. Uma aplicação quando efetua a subscrição de um determinado evento, terá conhecimento dos tipos de eventos que uma determinada aplicação gera. A primeira etapa de filtragem corresponde ao tipo de evento. Com a definição do tipo de evento, a aplicação que subscreve apenas irá receber a informação relativa ao tipo de evento selecionado. Numa segunda fase, serão definidos todos os parâmetros do evento que pretendemos receber. Imaginando um evento do tipo notificação, que contém os seguintes parâmetros: hora, mensagem, autor, título. Para uma aplicação subscritora em específico poderá ser suficiente a mensagem, para uma outra aplicação poder necessitar do conjunto de informação como por exemplo: autor, hora e mensagem. Dessa forma torna possível filtrar a informação segundo os interesses de cada aplicação subscritora. Sempre que for recebida informação que corresponda às características pretendidas o método encaminhará apenas a informação útil para o local indicado pela aplicação subscritora. Qualquer outra notificação que seja recebida que não preencha os requisitos definidos será descartada sem que a aplicação subscritora tenha conhecimento de tal.

```
function waitForEvent()
```

Fig. 5.5 – Protótipo do método waitForEvent()

Na Figura 5.5 é apresentado o protótipo do método *waitForEvent* que não recebe qualquer parâmetro nem devolve. Tem como objetivo esperar em segundo plano, pela chegada de conteúdo ao *endpoint*, aplicar os devidos filtros indicados pelo método *setInitInfo* e enviar para o local indicado pelo mesmo método.

Como em qualquer aplicação existem um conjunto de possíveis erros que necessitam de ser identificados da melhor forma. Dado que se trata de uma API orientada ao modelo web, para o tratamento de erros foram adotados os códigos de erro utilizado pelo HTTP que podem ser consultados com mais detalhe no capítulo 4.2.2. De forma a abranger alguns erros particulares da API de sincronização foi introduzida uma nova série de erros denominada de 6XX. No caso do método auxiliar caso um dos quatro parâmetros seja inválido ou nulo, retornará o erro 60X em que X corresponde à posição do argumento. No caso de outro erro inesperado retorna o valor 666 e aborta a execução. No método ler evento, retornará às mensagens de erro geradas pelo AJAX[1].

5.3. Descrição detalhada da API

5.3.1. Eventos

No contexto da API de Sincronização, um evento representa a informação em que um dado subscritor está interessado. Dependendo do tipo de aplicação um evento poderá representar vários tipos de informação, no entanto no âmbito deste projeto um evento poderá ser a execução de uma aplicação específica, inserção de novo conteúdo, alteração do extrato de texto apresentado num ecrã, etc. Um evento é caracterizado por um conjunto de parâmetros que definem convenientemente em que condição ocorreu, o que motivou a ocorrência do mesmo, onde ocorreu, quando ocorreu. A escolha desses mesmos parâmetros é efetuada pela aplicação, assim como também a definição das situações em que deve ser criado um novo evento.

Pensando num pequeno exemplo em que contemos duas aplicações. Uma aplicação efetua a medição periódica de temperatura dos vários espaços de um edifício, uma segunda aplicação efetua cálculos estatísticos.

No exemplo apresentado será gerado um evento sempre que a temperatura passe um dos limites (superior ou inferior), de forma a informar as duas aplicações interessadas nessa

[1] Asynchronous JavaScript and XML

ocorrência. Perante o exemplo são colocadas duas questões: "Que parâmetros definem o evento?", "Em que condições deve ser criado um novo evento?". No exemplo descrito o evento será uma notificação que será gerada para posteriormente ser difundida por todos os interessados. Os parâmetros que descrevem o evento terão que dar resposta a um conjunto de questões inerentes a qualquer tipo de evento. Questões como: "O quê?", "Onde?", "Quem?", "Quando?", são apenas exemplos de questões a que o conjunto de parâmetros do evento têm que dar resposta.

- Tipo – "Alerta" / "Informação"
- Valor –"Valor que atingiu"
- Local – "Define o local da ocorrência"
- Entidade –"Quem gerou o evento"
- Oscilação – "Tipo de variação"
- Data –"Data da ocorrência"
- Hora – "Hora da ocorrência"
- Onde – "Onde ocorreu"

A lista de parâmetros apresentada é apenas um exemplo, sendo que estes são apenas os essenciais para uma caracterização do tipo de evento. De salientar que o parâmetro tipo será obrigatória, devido a variedade de eventos dentro do mesmo contexto. No entanto caso necessário podem ser acrescentados outros parâmetros que possam ser úteis. Além dos parâmetros que definem o evento, é necessário definir quando é que esse mesmo evento deve ser criado. Neste caso a aplicação de monitorização terá que estabelecer um conjunto de condições (limites de temperatura).

```
 (...)
$fim =  false;
$var = "";
$temp = 0;

while($fim!=true)  {
   $temp = getTemp();
   if($temp<18){
      $fim = true;
      $min = true;
      }
      if($temp>25){
         $fim = true;
         $var = "max";
      }
      else{
         $parametros = array("tipo" => "Informacao",
        "desEvento" => "Nota informativa sobre a temperatura atual",
        "local" => "sala Y",
        "temperatura"=>$temp, "oscilacao"=>"");
        $ res = creat_event($parametros);
         sleep(5);
      }
 $parametros = array("tipo" => "Alerta",
"desEvento" => "subida de temperatura muito acentuada",
"local" => "sala Y", "temperatura"=>$temp, "oscilacao"=>$var);
 $ res = creat_event($parametros);

 (...)
```

Fig. 5.6 – Exemplo de condições para a criação de um novo evento

Na Figura 5.6 é possível analisar um pequeno exemplo de código php que exemplifica de como podem ser definidas condições para a criação de um novo evento. Utilizando o cenário de monotorização de temperaturas, a aplicação deverá periodicamente efetuar uma medição da temperatura. No exemplo são definidas duas situações distintas em que é criado um evento, enquanto não é passado um dos limites da temperatura (>18 e <25) e quando passa esses limites. Assim quando não passa os limites será criado um evento do tipo "informativo" apenas para informar a temperatura atual a possíveis interessados e continua a monitorização. Já no caso se verifique um pico sairá do ciclo de monitorização periódica

e cria um evento do tipo "alerta" e com os restantes parâmetros necessários para a situação.

5.3.2. Criar evento

Na Figura 5.1 é possível visualizar o protótipo do método *criar evento* que tem como objetivo criar um evento sempre que estejam reunidas as condições para tal. Antes de invocar o método *creat_event()*, é necessário criar um objeto do tipo *evento*. Implica por isso a utilização do método construtor *Eventos ($tit,$link, $desc)* disponibilizado pela classe *eventos.php*. Recebe como argumentos o título da Aplicação, o link da mesma e uma breve descrição. Os parâmetros indicados são os obrigatórios apenas para a criação do ficheiro RSS feed. A informação detalhada sobre os campos presentes num ficheiro *feed* encontra-se no Anexo A do presente documento. O método recebe um único argumento que se trata do conjunto de parâmetros que definem o evento. O número e o tipo de parâmetros são variáveis, no entanto dentro da lista de parâmetros deve estar indicado o tipo de evento através do parâmetro "tipo", tal como indicado no secção 5.3.1. Devido ao protocolo produtor/subscritor[2] adotado é necessário a existência de um web feed que representa um conjunto de informação modelada num ficheiro xml. No contexto do presente trabalho, esse conjunto de informações serão denominadas por eventos. Cada alteração indicará um novo evento (uma nova entrada no ficheiro web *feed*). Por sua vez como indicado no capítulo 5.3.1 dentro do mesmo contexto podem surgir vários eventos de tipos diferentes. Assim sendo um *web feed* que contenha um determinado tema, poderá conter N eventos com tipos diferentes. Daí surge a necessidade de introdução de filtragem, descrita com detalhe no secção 5.3.5. Será a partir do endereço URL desse mesmo ficheiro que será difundida a informação entre os três elementos intervenientes (produtor, subscritor, *Hub*). Cada aplicação terá assim que conter a si associado um ou vários *web feeds*. Os ficheiros *web feed*, poderão encontrar-se em qualquer lugar na internet, desde que se encontra acessível ao *Hub*. A geração de um evento passará por dois processos distintos: identificação dos parâmetros que definem o evento e posteriormente adicionar uma nova entrada ao ficheiro com a informação indicada.

[2] PubSubHubbub

```
$parametros = array(
            "tipo" => "Alerta",
            "desEvento" => "subida de temperatura muito acentuada",
            "local" => "Sala Y",
            "temperatura"=>$temp,
            "oscilacao"=>$var);
```

Fig. 5.7 – Exemplo de um conjunto de parâmetros

Na Figura 5.7 é possível ver um pequeno exemplo contendo um conjunto de parâmetros alusivo ao exemplo anterior. Os parâmetros poderão ser de vários tipos, como por exemplo: *array*, *String*, *inteiro*, *booleano*, etc. Esse conjunto de dados será armazenado num array para que posteriormente o método criar evento o receba como argumento. Além do conjunto de parâmetros que definem o evento e variam consoante a aplicação serão introduzidos dois parâmetros fixos: a origem do evento (IP) e a data/hora de criação. Para a obtenção do objeto "evento" é efetuada a junção num só *array* dos parâmetros recebidos com os dois argumentos fixos. O passo seguinte será acrescentar uma nova entrada no ficheiro web *feed* com todos os parâmetros recebidos. Para a atualização das entradas no ficheiro de eventos, é utilizado o método auxiliar *create_feed_rss*.

```
function create_feed_rss($title, $link, $description, $evt)
```

Fig. 5.8 – Protótipo do método auxiliar create_feed_rss

Na Figura 5.8 é apresentado o protótipo do método create_feed_rss que recebe quatro argumentos, sendo que um deles representa o evento ($evt). Os restantes representam um conjunto de tags do *web feed* que são obrigatórias de forma a respeitar a estrutura básica de um *feed*, tal como foi descrito no anexo A. Cada um dos parâmetros representará uma tag XML, sendo que os nomes das tags obrigatórios são fixas e iguais para qualquer *web feed* em RSS.

```
<?xml version='1.0' encoding='UTF-8'?>
        <rss version='2.0'>
        <channel>
          <title>Aplicacao de temperatura</title>
          <link>www.emergencias.pt</link>
            <description>Aplicacao de monitorizacao de temperaturas e consumos
</description>
```

Fig. 5.9 – Tags obrigatórias de um ficheiro feed

Na Figura 5.9 é possível verificar a forma como é constituído um qualquer ficheiro *web feed* e tal como indicado todos os nomes das tags são fixos.

No caso das restantes tags a informação contida no array na forma "key"=>"value", será adicionada ao ficheiro xml na forma "<key>value</key>".

```
<item>
    <tipo>Alerta</tipo>
    <descEvento>subida de temperatura muito acentuada</descEvento>
    <local>sala Y</local>
    <oscilacao>max</oscilacao>
    <temperatura>30</temperatura>
    <data>2013/03/26-22:10:47</data>
    <origem>46.189.205.167</origem>
</item>
```

Fig. 5.10 – Tags resultantes de um exemplo de array de parâmetros

Na Figura 5.10 é ilustrada uma entrada no ficheiro eventos utilizando os parâmetros do evento apresentado no exemplo da Figura 5.7. Surgiu um tag denominada de "item" que simboliza cada entrada introduzida no ficheiro. Cada ficheiro eventos (*web feed*) terá tantas tags item quantos os eventos que forem gerados.

No caso de todo o processo descrito ser bem-sucedido o método retornará o valor "600". Caso o argumento que o método recebe seja inválido ou inexistente o método terminara e devolve o valor 601. Noutros possíveis erros durante a execução o método retornará o valor "666" e termina a execução. Todos os erros serão registados num ficheiro de log desenvolvido para o efeito. Nesse mesmo ficheiro ficarão registados todos os detalhes existentes sobre o erro ocorrido além do instante em que ocorreu e do método que deu origem.

5.3.3. Publicar evento

O método *publish_event()* - publicar evento – permitirá as aplicações publicar eventos ocorridos. O método será invocado sempre que surgir um novo evento, sendo que o produtor é indiferente a possíveis interessados. Limitar-se-á a informar o *Hub* da ocorrência de um determinado evento através do protocolo PubSubHubbub e por sua vez o *Hub* notificará todos os interessados.

Na Figura 5.2 é apresentado o protótipo do método *publish_event* que como se pode verificar recebe dois argumentos: evento que pretendemos publicar (topic_url) e por sua

vez o endereço do Hub onde pretendemos publicar.

Numa primeira fase é efetuada uma validação ao endereço do *Hub*, de forma a certificar que de facto é um endereço válido. Caso o endereço seja válido é também testado o endereço do evento (*web feed*) a ser publicado. Caso alguma destas operações falhe a execução termina não sendo efetuada a publicação pretendida. Por sua vez o método devolverá o valor "60X"caso detete alguma anomalia com o argumento X que neste caso terá os valores de 1 ou 2. A publicação do evento consiste num POST HTTP em que no seu conteúdo terá que conter um conjunto de parâmetros para que a publicação seja válida, entre esses parâmetros estará o endereço do evento. O endereço do evento deve conter um ficheiro *web feed rss* com a estrutura correta de um feed genérico. Caso o ficheiro não respeite a estrutura de um *web feed rss*, poderá ser aceite pelo *Hub* no entanto não será processado e posteriormente encaminhado para todos os interessados. A resposta do *Hub* à publicação será um código HTTP adequado a situação como descrito no capítulo 4.2.2 na Tabela 4.1. O método por sua vez devolverá o valor recebido do *Hub*, mantendo o seu significado como já anteriormente indicado. Desta forma permitirá ao produtor detetar em que estado ficou a publicação.

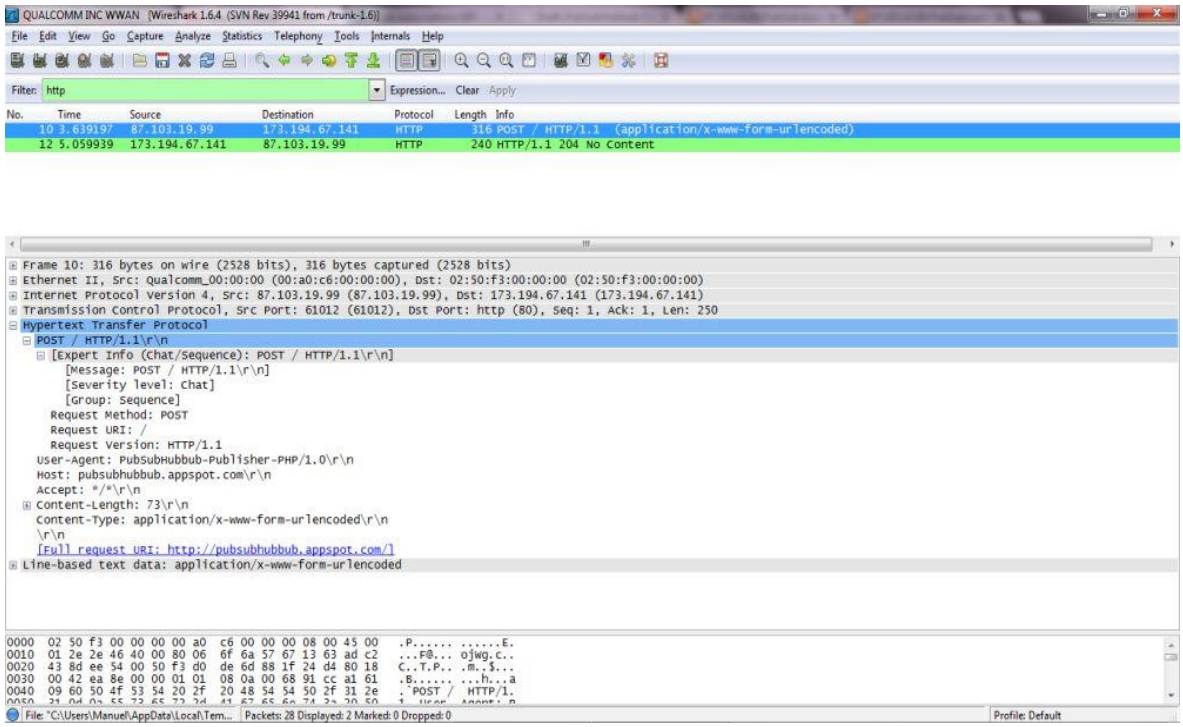


Fig. 5.11 – Exemplo de um Post efetuado pelo Produtor

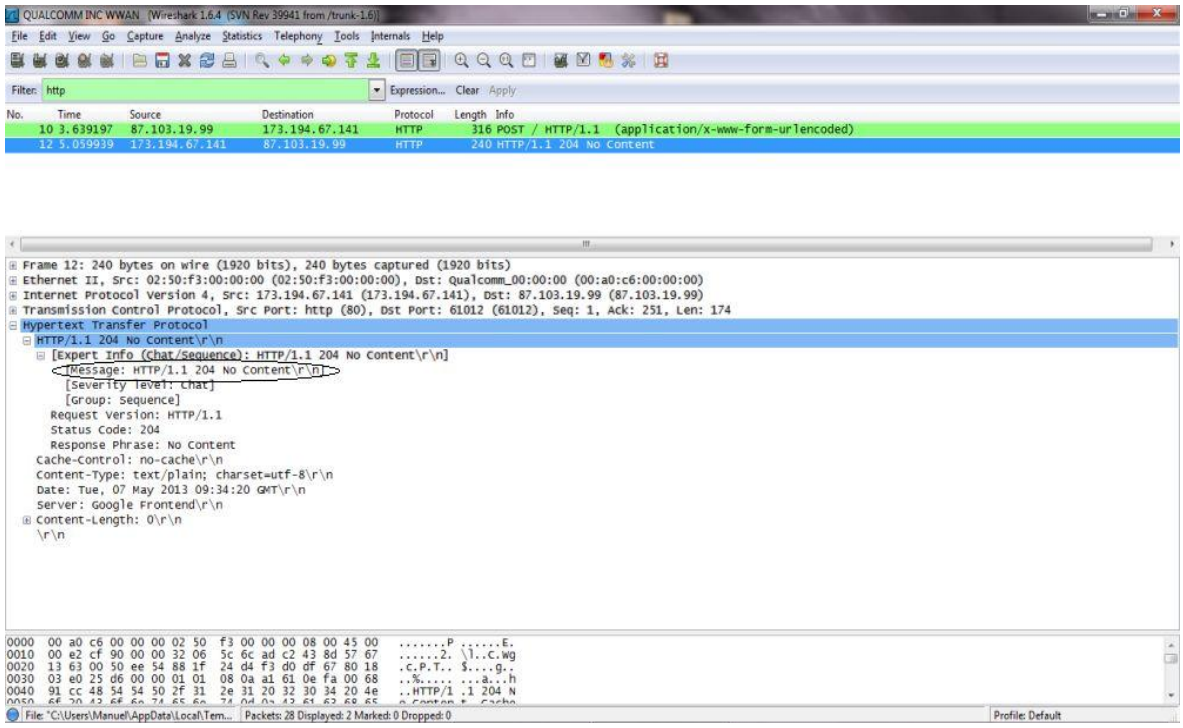


Fig. 5.12 – Resposta do Hub ao Post do produtor

As Figuras 5.11 e 5.12 ilustram uma publicação bem-sucedida através da captura de pacotes http, efetuado pela ferramenta wireshark. Na Figura 5.11 podemos verificar um post efetuado pelo subscritor neste caso para o *Hub* de referência do Google [25]. Pela resposta do *Hub* representada na Figura 5.12 pode verificar-se que a publicação foi aceite, tal pode comprovar-se pela área selecionada da Figura 5.12 que contém a mensagem HTTP "204 – No Content".

Sempre que for publicado um evento em que nenhuma entidade tenha manifestado interesse na mesma, o *Hub* descartara a publicação automaticamente. Apesar disso o *Hub* retornará a mesma mensagem de uma publicação efetuada com sucesso (HTTP "204 – No Content").

5.3.4. Subscrever evento

O método *subscribe_event()* – subscrever evento – permitirá as aplicações subscrever um evento de uma dada aplicação.

Na Figura 5.3 está ilustrado o protótipo do método *subscribe_event* e pode verificar-se que o método recebe os seguintes três argumentos: hub_url, callback_url, topic_url, lease_seconds.

Da mesma forma que em métodos anteriores o argumento *hub_url* é o endereço do *Hub*, que neste caso será onde pretendemos subscrever o evento. Quando invocado o método, a

variável que contém o endereço do *Hub* é validada e caso não contenha nenhum conteúdo ou caso não seja um *url* válido será abortada a operação. O parâmetro *callback_url* refere-se ao endereço da aplicação/componente que pretendemos que receba a notificação da ocorrência do evento que subscrevemos (*endpoint*). Tal como o endereço do Hub, este também será validado de forma a verificar se é efetivamente um endereço correto. O terceiro parâmetro do método contém o endereço do evento que pretendemos subscrever. Da mesma forma que o endereço do *hub* e o endereço do *endpoint* será também submetido a validação de forma a verificar se a variável contém efetivamente um endereço e se é um endereço válido. À semelhança do método *publish_event* a execução do método retornará o valor "60X" em caso de um dos três argumentos conter algum problema, sendo que X tomará o valor de 1, 2 ou 3 referindo-se ao primeiro, segundo e terceiro argumento respetivamente. Além de retornar o código de erro, introduzirá no ficheiro de log uma nova entrada por cada erro detetado. O quarto argumento ao contrário dos restantes ter a particularidade de não ser obrigatório. O valor deverá ser um inteiro que represente o número de segundos que a aplicação pretende manter a subscrição ativa. Caso não seja introduzido um valor, a API de Sincronização por defeito indicará o valor de 86400 segundos que representa uma subscrição válida durante um dia.

Para uma subscrição ser bem-sucedida além de ser submetida a várias validações já enunciadas necessita de dispor de um *endpoint*. A entidade denominada de *endpoint* (script php presente na API de Sincronização) terá que ser acessível do exterior para que possa receber notificações provenientes do *Hub*. Terá duas tarefas importantes: Confirmar a subscrição e receber notificações do *Hub*;

Sempre que for feita uma subscrição o *Hub* enviará um desafio ao *endpoint*, de forma a verificar se efetivamente a subscrição foi feita pela mesma entidade. Caso isso não se verifique ou o *endpoint* não responda o Hub descartará a subscrição. Caso se confirme a subscrição o *Hub* ativará a mesma. A script "endpoint" disponibilizada pela API de sincronização dispõe de mecanismos capazes de responder ao desafio lançado pelo *Hub*, para que a subscrição fique ativa.

Tal como indicado anteriormente a API de Sincronização adotou os códigos de estado HTTP, daí qualquer subscrição submetida ao *Hub* retornará códigos de estado HTTP. Os códigos retornados são os adequados à situação do estado em que se encontra a subscrição: aceite, não aceite, pendente ou poderá ocorrer um qualquer erro que nada tenha a ver com o protocolo. Os detalhes sobre o significado de cada código são apresentados na secção 4.2.2,

na Tabela 4.1. A API de Sincronização da mesma forma que em métodos anteriores limitar-se-á a retornar os códigos recebidos do *Hub*. Se o *Hub* responder com um "204 – No Content" o método retornará o valor "204".

Apesar de uma subscrição ser bem-sucedida não significa que a aplicação que a efetuou venha a ser notificada pela ocorrência do evento. Existem dois motivos para tal acontecer, um deles é não haver qualquer garantia que o evento X ocorra alguma vez. O outro motivo é pelo facto de a subscrição poder expirar. Para prevenir este segundo motivo, deve ser definido um tempo para a subscrição se manter ativa e caso necessário a aplicação poderá desenvolver um mecanismo de renovação da subscrição. Esse mecanismo não é mais do que efetuar nova subscrição com os mesmos argumentos. O *Hub* sempre que detetar uma subscrição proveniente do mesmo subscritor a subscrever o mesmo evento renovará automaticamente, ficando apenas ativa a ultima subscrição bem-sucedida.

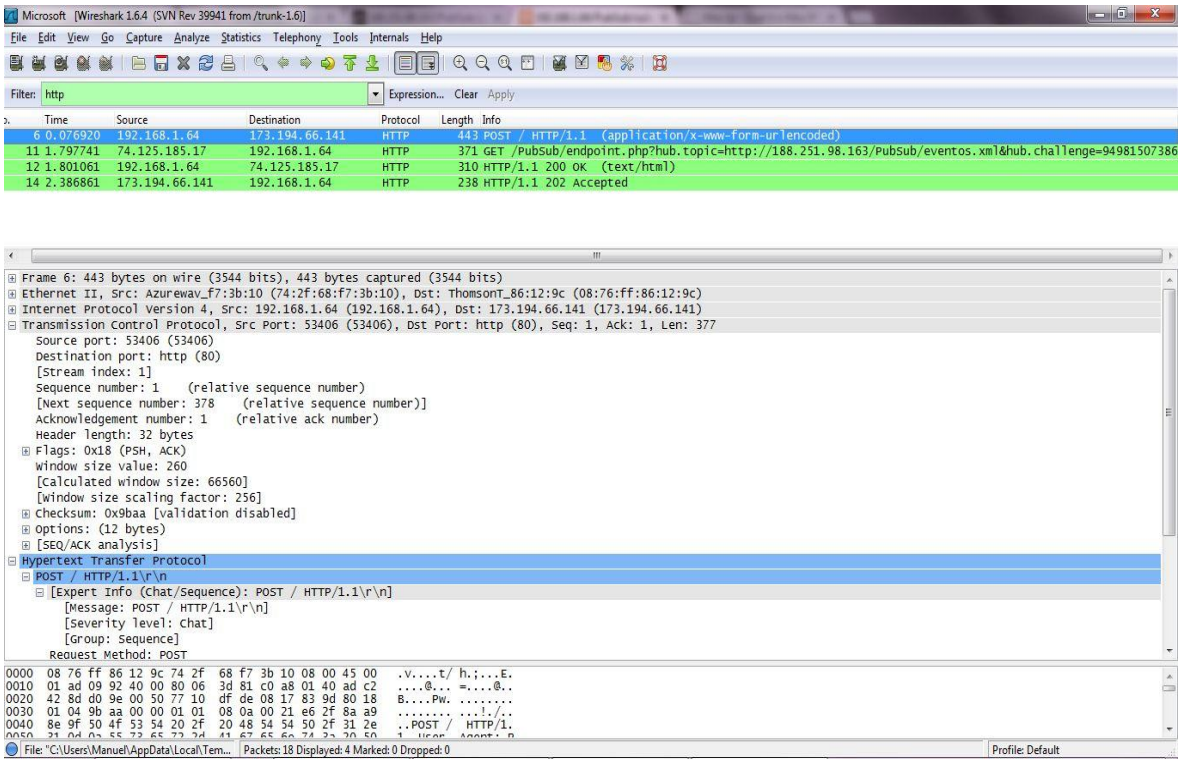


Fig. 5.13 – Exemplo de troca de mensagens durante uma subscrição

Na Figura 5.13 é possível verificar as várias mensagens trocadas entre o subscritor e o *Hub*. Numa primeira mensagem é efetuado o POST no endereço do Hub que posteriormente responde com um desafio ao *endpoint* indicado na subscrição.

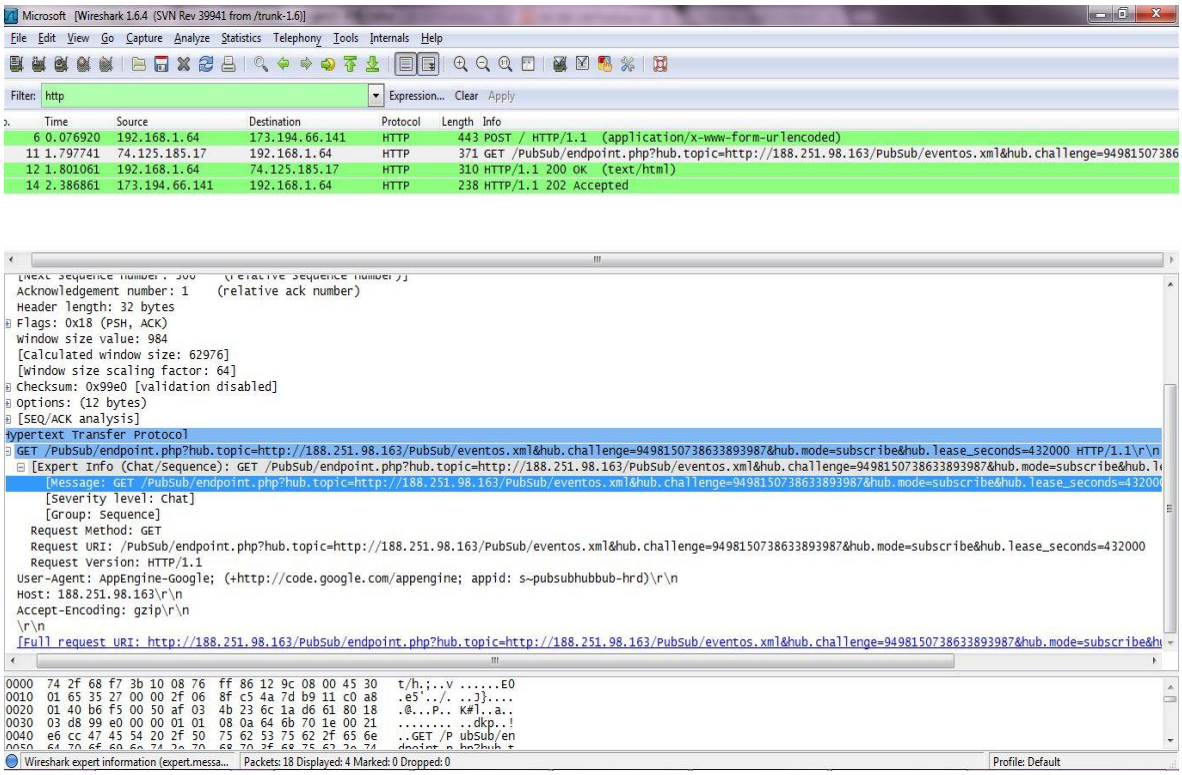


Fig. 5.14 – Exemplo de desafio enviado pelo Hub ao endpoint

Na Figura 5.14 é possível verificar um exemplo de um desafio enviado pelo *Hub* ao *endpoint*, após o POST a efetuar a subscrição. No exemplo apresentado na Figura 5.14 a subscrição foi efetuada com sucesso após *Hub* confirmar a subscrição junto do *endpoint* envia a mensagem de "Accepted", confirmando assim que a subscrição está ativa. A partir desse momento quando ocorrer o evento subscrito, o subscritor será devidamente notificado.

5.3.5. Ler evento

A funcionalidade de ler eventos é composta por dois métodos, sendo eles *setInitInfo* e *waitForEvent*. Tal como indicado no método *subscribe_event*, a componente/aplicação que utilizar a API de Sincronização terá que dispor de um mecanismo capaz de ler e filtrar notificações enviadas pelo *Hub*. Assim surgiu a necessidade de criar um método capaz estar em constante escuta de novas notificações e ao mesmo tempo verificar se de facto interessam ou não à aplicação. O processo de subscrição é iniciado pela subscrição de um dado conteúdo que é efetuado através do método *subscribe_event()*. Após a ativação da subscrição, o *Hub* notificara o *endpoint* sempre que surgirem alterações/ocorrências no tema subscrito. Nesta fase serão recebidas todas as notificações sem qualquer filtragem, associadas ao *web feed* subscrito. A cada 100 milissegundos o método *waitForEvent* irá

questionar o *endpoint* se recebeu nova informação. Em caso afirmativo procede a verificação se o tipo de evento que recebeu corresponde ao evento pretendido pela aplicação subscritora. Se de facto corresponder, irá efetuar uma seleção dos atributos pretendidos e enviar essa informação para o local indicado. Por outro lado irá ativar uma flag booleana com o valor "true". A flag tem como nome "flag". Tal como indicado pelo protótipo do método *setInitInfo* ilustrado na Figura 5.4, irá apresentar os dados necessários para a filtragem, tais como: tipo de evento, local de envio da informação e parâmetros pretendidos. De salientar que sem a definição dos parâmetros iniciais através do método *setInitInfo* o método waitForEvent não funcionará.

Utilizando o exemplo apresentado no secção 5.3.1 vamos supor que existe uma terceira aplicação denominada de Alertas, que pretende apenas ser notificada em caso de violação de um dos limites. Por outras palavras apenas interessam eventos da aplicação de monitorização do tipo "alerta", todos os restantes tipos de eventos não interessam a aplicação Alertas.

```
params[0] = "local";
params[1] = "temperatura";
params[2] = "oscilacao";
setInitInfo("endpoint.php", "alerta", "envia.php", params);
waitForEvent();
```

Fig. 5.15 – Exemplo de definição de condições de filtragem

Na Figura 5.15 é possível verificar um *array* de parâmetros em *JavaScript* que representam o conjunto de condições e informações pretendidas pela aplicação Alertas. Tem como condição que o tipo de evento seja "alerta" e pretendido saber onde ocorreu, a temperatura que atingiu e se foi ultrapassado o limite superior ou inferior. O *array* de parâmetros apresentado na Figura 5.15 é um dos argumentos do método *setInitInfo()*. No exemplo concreto o método ler evento irá verificar se foi recebido novos eventos na página local "endpoint.php". No caso de encontrar informação que corresponda ao pretendido encaminhará para a página "envia.php".

As notificações enviadas pelo *Hub* indicando que ocorreu um determinado evento, serão efetuadas através de um Post HTTP. Dado que o método *waitForEvent* foi desenvolvido em ajax permite que toda a aplicação execute normalmente e em segundo plano estejam a executar outras operações. Sendo que apenas irá interromper a aplicação em caso de

receber um evento dentro das características recebidas. No caso de não corresponder, descarta a mensagem recebida e continua com a execução. No caso de encontrar uma mensagem do tipo pretendido armazena num *array* os atributos pretendidos que no exemplo são: local, a temperatura e a oscilação. Após obter a informação o método enviará a mesma através de um post.

```
$res = array(
            "0" => "Sala B",
            "1" => "28",
            "2" => "max");
```

Fig. 5.16 – Array resultante da execução do método read_event

Na Figura 5.16 pode verificar-se o *array* resultante da receção de um evento que corresponde às condições impostas. A informação associada a cada índice do *array* retornado, é a correspondente a pretendida no *array* de parâmetros indicados no método *setInitInfo()*. No exemplo concreto o índice "0" corresponde ao local que obteve como resultado "Sala B".

Como já foi enunciado no subcapítulo 5.3.4, uma subscrição bem-sucedida de um evento não é sinónimo de ocorrência do mesmo. Para evitar esperas infinitas a componente da aplicação que invocar o método deverá definir um tempo de execução. Expirado esse tempo a aplicação poderá optar por alterar os parâmetros de filtragem ou abortar em definitivo a leitura de eventos. Caso não seja definido um período de execução, existe a possibilidade de o método ficar "eternamente" na espera de receber um evento com os parâmetros pretendidos. Caso nesse período receba um evento, o mesmo será filtrado e caso corresponda será encaminhado para a entidade que irá processar a sua informação. O processo de espera por mais eventos, não é interrompido e continua a espera de novas ocorrências. Tal é importante dado que o instante de chegada dos eventos é imprevisível assim como a quantidade dos mesmos. Poderão chegar um conjunto de eventos consecutivos, da mesma forma que poderão estar longos períodos sem a ocorrência do evento subscrito.

5.4. Linguagem adotada e Plataforma de desenvolvimento

A linguagem adotada para o desenvolvimento deste projeto foi PHP numa primeira fase. Posteriormente de forma a otimizar a API, foram introduzidos alguns módulos em

JavaScript. PHP (Personal Home Page) é uma linguagem de script open source, especialmente pensada para aplicações presentes do lado do servidor em ambientes web. Ao contrario de outras linguais como C e Perl veio permitir a introdução de código HTML juntamente em páginas PHP, o seu código é delimitado pelas tags iniciais e finais <?php e ?>. O PHP irá executar a script e gerar o código HTML que será enviado para o cliente, ocultando o conteúdo da página que se encontra no servidor. A escolha desta linguagem deve-se ao facto de a API de sincronização ser orientada a aplicações web, como tal a escolha da linguagem teria que ir de encontro com tal característica. PHP ganha em relação a linguagens como: perl, java, python, entre outras devido a sua simplicidade mas ao mesmo tempo é uma linguagem com grandes potencialidades. Por outro lado PHP foi desenvolvido para o desenvolvimento de aplicações web, enquanto outros exemplos de linguagens foram pensados para muito mais do que isso. Tal fator poderia tornar o desenvolvimento bastante mais complexo quer para a API, quer posteriormente para a sua utilização.

Por sua vez a foram introduzidos alguns métodos em JavaScript que ao contrário do PHP executa no lado do cliente. Não necessita de qualquer instalação, sendo que qualquer *browser* da mesma forma que suporta HTML, também suportará JavaScript, no entanto é necessário verificar se está ativo. Contém enormes vantagens, não tratando o código HTML como simples Strings, mas sim cada tag como um objeto individual. O JavasScript é indicado para otimizar aplicações, através da execução de métodos não que necessitem de conteúdo do servidor. Permite assim aliviar a carga do servidor, tornando a aplicação do lado do cliente mais autónoma.

No que diz respeito à plataforma de desenvolvimento foi utilizado o Oxygen. A sua escolha advém da sua versatilidade de ofertas. Dado que o projeto contém várias linguagens, tornou-se importante na mesma plataforma reunir todos os componentes. Teria que ser dado suporte a linguagens como: PHP, JavaScript, HTML e CSS. Ao mesmo tempo teria também que dar suporte a ficheiros XML de forma a auxiliar no debug. Trata-se de um editor simples, intuitivo capacitado de todos os mecanismos necessários para o projeto. Por outro lado contém também uma enorme vantagem, visto que permite validar a constituição de vários tipos de ficheiros como XML ou HTML, verificando se estão conforme a norma.

6. Sistema de demonstração

Como prova de conceito do nosso sistema, desenvolvemos uma aplicação Web direcionada para a execução em ecrãs públicos. Esta aplicação é uma aplicação que apresenta uma funcionalidade muito simplificada, mas ilustra os principais requisitos de sincronização discutidos neste documento. Cada instância da aplicação, a executar provavelmente em diferentes servidores públicos e associada a uma instalação de ecrã público, exibe uma animação de imagens acompanhada por comentários submetidos pelos utilizadores. A submissão de uma nova imagem e de um novo comentário implica uma reação síncrona entre todas as instâncias.

A aplicação é composta por três componentes. O componente central apresenta os conteúdos (comentários, imagens e respetivas descrições). Existem dois componentes auxiliares que permitem a interação dos utilizadores com a aplicação, dando a possibilidade ao utilizador de submeter novas imagens e comentários no sistema.

6.1. Descrição da aplicação de demonstração

A aplicação é uma aplicação Web e foi desenvolvida em HTML, PHP e *JavaScript*. As estruturas de dados necessárias ao funcionamento da aplicação, foram modeladas em XML. A aplicação é constituída pelos três componentes seguintes:

- Apresentação de conteúdos
- Submissão de comentários
- Submissão de novos conteúdos

A aplicação apresenta um conjunto de conteúdos, mais concretamente imagens e os respetivos comentários. Cada imagem tem associado a si um conjunto de comentários, sendo que cada comentário apenas tem indicação da data e hora em que foi feito. A aplicação não tem qualquer gestão de utilizadores, daí qualquer participação, seja com novas imagens ou comentários, é totalmente anónima. A apresentação de conteúdos é feita através de *slideshow*. A página de apresentação de conteúdos é dividida em três componentes: apresentação de comentários, apresentação da imagem e apresentação da descrição e do respetivo ID da imagem. Durante o tempo em que a imagem é apresentada, os vários comentários são apresentados através do deslizamento dos mesmos, para que todos possam ser visualizados.

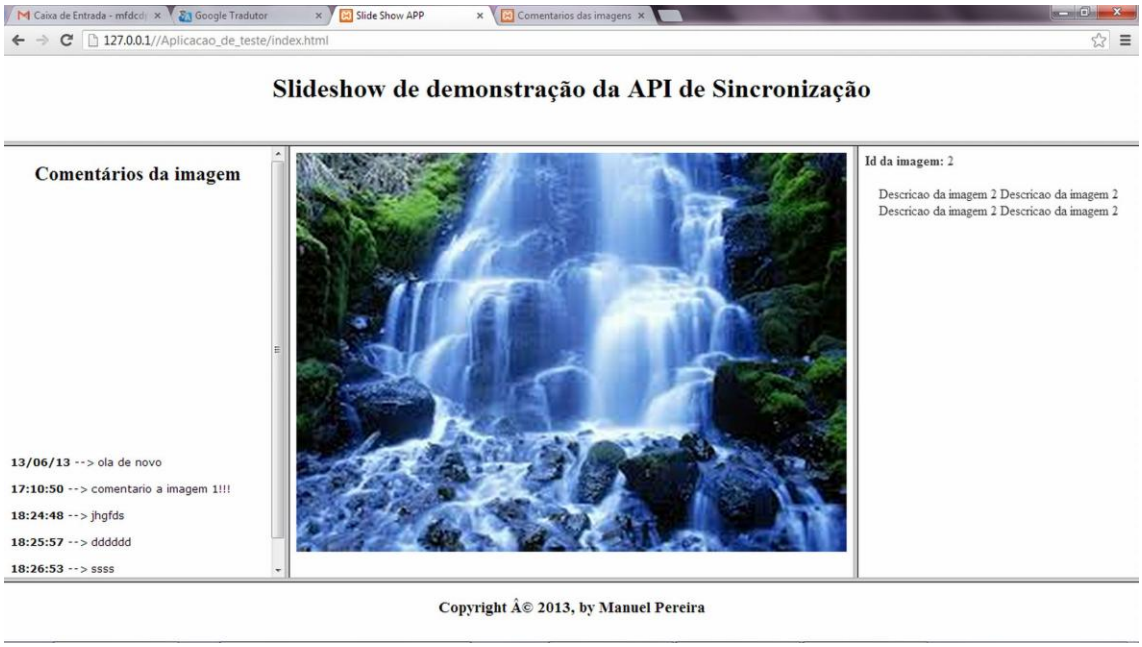


Fig. 6.1 – Ilustração da página de apresentação de conteúdos

Na Figura 6.1 é apresentada a página de apresentação de conteúdos. Como já foi referido anteriormente a página está dividida em três componentes. A primeira é composta pelos diversos comentários relativos à imagem, a segunda é composta pela imagem e a terceira é composta pela descrição da imagem, juntamente com o seu ID.

Fig. 6.2 – Página de submissão de novos comentários

Na Figura 6.2 é apresentada a página que permite a um utilizador submeter um comentário sobre uma imagem. As imagens são identificadas através de um ID. Este ID é utilizado para referir a imagem à qual é submetido o comentário. De forma a garantir que não existe

nenhum lapso na escolha da imagem, os utilizadores podem "pré-visualizar" a imagem associada ao código selecionado. Caso não corresponda, podem alterar o ID de forma a obterem a imagem pretendida. De salientar que não será possível efetuar qualquer comentário, sem efetuar a pré-visualização de forma a salvaguardar eventuais lapsos. Caso o utilizador tente efetuar um comentário sem identificar a imagem, ocorrerá um erro como o apresentado na Figura 6.3.

Fig. 6.3 – Ilustração de um erro em caso de não efetuar pré-visualização

Após a mensagem de erro, a aplicação voltará à página principal de introdução de novos comentários.

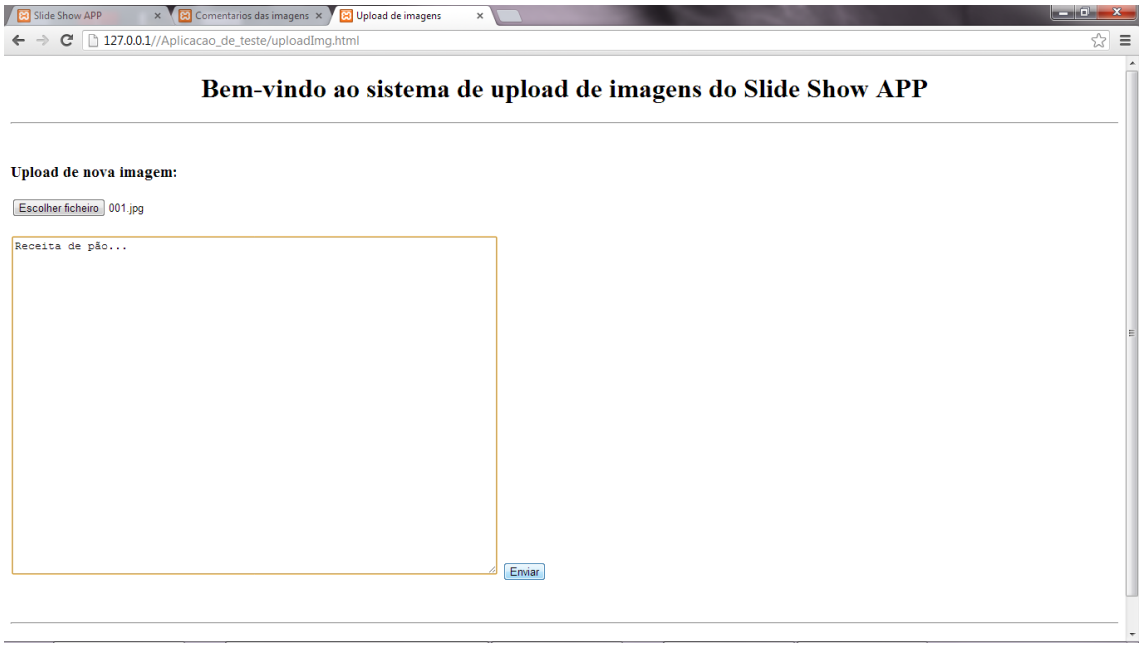


Fig. 6.4 – Ilustração da página de submissão de imagens

Na Figura 6.4 é possível visualizar a página de *submissão* de novas imagens. De uma forma simples, para efetuar *a submissão* de uma imagem deve-se indicar o caminho no disco local onde se encontra a imagem. Opcionalmente poder-se-á inserir uma pequena descrição sobre a imagem. De forma a efetuar uma boa gestão do sistema, e a prevenir alguns erros, o sistema limitou a dimensão dos ficheiros a 2 Mb. Adicionalmente, limitou também os formatos de ficheiros, permitindo apenas os seguintes formatos: *gif*, *jpeg*, *jpg*, *pjpeg*, *x-png* e *png*. Qualquer imagem que não respeite uma ou ambas limitações, não será adicionado ao sistema e será apresentada uma mensagem de erro descrita na 6.5.

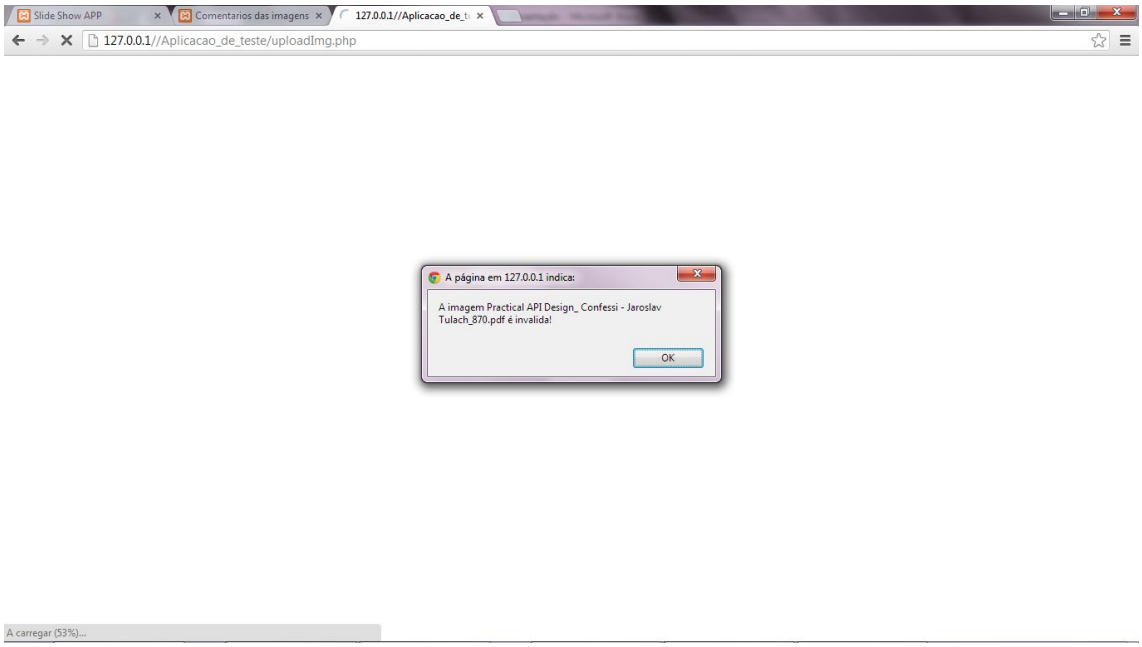


Fig. 6.5 – Mensagem de erro de imagem invalida

6.2. Descrição dos requisitos de sincronização

A aplicação desenvolvida será disponibilizada num servidor público e provavelmente replicada em diferentes servidores. Cada ecrã público será associado a uma instância da aplicação e disponibilizará a animação das imagens/comentários aos utilizadores.

A conceção da aplicação tem como requisito a execução sincronizada de cada instância. Isto é, independentemente da localização de cada instância, sempre que é submetida uma imagem ou comentário a uma instância todas as outras serão notificadas e o novo conteúdo exibido sincronamente. As notificações de novo conteúdo dão origem à visualização de uma nova página com o novo conteúdo.

De forma a dar resposta aos requisitos impostos pelo sistema, foram definidos dois tipos de eventos no contexto do nosso *Framework* de sincronização. Será gerado um evento do tipo "Novo comentário" sempre que for submetido um novo comentário. Adicionalmente, caso seja submetida uma nova imagem será gerado um evento do tipo "Nova imagem".

```
<item>
    <tipo>Novo comentario</tipo>
    <idImg>7</idImg>
    <comentario>Linda montanha!!!</comentario>
    <data>2013/09/24-16:32:18</data>
    <origem>127.0.0.1</origem>
</item>
```

Fig. 6.6 – Exemplo de evento "Novo comentário"

Na Figura 6.6 é possível visualizar a descrição de um evento do tipo "Novo comentário", que será gerado sempre que for efetuado um novo comentário por alguma instancia da aplicação. O evento "Novo comentário" é definido por cinco atributos: tipo, id da imagem, comentário, data e a origem. Tal como indicado no subcapítulo 5.3.1 o tipo deve-se à necessidade de definir que tipo de evento se trata, dado que o *feed* associado aos comentários poderá conter outro tipo de eventos. Será também necessário indicar a que imagem se refere o comentário. Por outro lado além do texto do comentário é útil conter a data e a origem do mesmo, dado que para determinadas aplicações poderão aplicar algum tipo de filtro através de tais características.

```
<item>
   <tipo>Nova imagem</tipo>
   <nome>imagem.jpg</nome>
   <id>26</id>
   <data>2013/07/05-15:31:40</data>
   <origem>87.103.47.240</origem>
</item>
```

Fig. 6.7 – Exemplo de evento "Nova imagem"

De uma forma similar ao evento "Novo comentário", o evento "Nova imagem", apresentado na Figura 6.7, é definido por cinco atributos. Pelas mesmas razões já anunciadas anteriormente, necessita de um tipo de evento. Por outro lado necessita do nome e do identificador que foi gerado pela aplicação no momento da inserção da imagem no sistema. Será também registado a data e a origem de tal inserção para que as aplicações possam fazer filtragem de eventos por data e/ou origem.

6.3. Implementação

A implementação da aplicação é dividida em três componentes já apresentados:

- Apresentação de conteúdos
- Submissão de comentários
- Submissão de novos conteúdos

Para a implementação de cada componente, além da utilização da API de Sincronização, foram desenvolvidas um diversos documentos HTML, PHP e scripts em JavaScript.

6.3.1. Pressupostos/Inicialização

Tal como indicado no capítulo 4.2.1, o *Framework* desenvolvido baseia-se no sistema PubSubHubbub, sendo necessário indicar o componente *Hub*. Na realização desta prova de conceito, utilizamos o *Hub* desenvolvido pela Google[3] e disponibilizado pelo projeto PubSubHubbub. Numa primeira fase, a aplicação foi apenas disponibilizada num servidor público, sendo instanciada o número de vezes correspondente ao número de ecrãs associados. Numa segunda fase, a aplicação foi adicionalmente disponibilizada num segundo servidor público (foi utilizado para o efeito o servidor público externo http://www.*000webhost*.com).

As configurações iniciais da aplicação no contexto do Framework são o endereço do http://pubsubhubbub.appspot.com/ e os endereços dos *feeds* que contêm as alterações de conteúdo, ou seja os eventos produzidos e subscritos apresentados na secção 4.2.1. Como já foi referido, os eventos correspondem à submissão de novas imagens e submissão de novos comentários. Na segunda fase de testes, os documentos *feed* foram alojados no segundo servidor público. Os endereços dos documentos *feed* são: *http://slideshowapp.host22.com/Aplicacao_de_teste/eventos_image.xml* e *http://slideshowapp.host22.com/Aplicacao_de_teste/eventos_coment.xml*.

6.3.2. Apresentação de conteúdos

Nome da página	Descrição
titulo.html	Título da página
comentarios.php	Apresentação dos comentários da imagem atual
playerImg.php	Apresentação do conjunto de imagens em ciclo
playerDesc.php	Apresentação da descrição da imagem atual
rodape.html	Rodapé da página

Tabela 6.1 – Componentes que compõem o componente de apresentação de conteúdos

[3] Empresa multinacional de serviços online, fundada por Lerry Page e Sergey Brin em 1996. Disponível através da página www.google.pt

Na Tabela 6.1 é possível analisar o conjunto de subcomponentes que compõem o componente de apresentação de conteúdos (index.html). A página *titulo.html* não é mais do que uma String que define o título da página. Da mesma forma o *rodape.html*, apenas apresenta os direitos de autor relativos à página. A subpágina principal é a *playerImg.php* que como é indicado na Tabela 6.1 corresponde à apresentação de um conjunto de imagens em ciclo. A Página contém um *slide show* implementado sendo que apresenta uma imagem durante um determinado período.

```
(…)
<script type="text/javascript" language="Javascript"
src="https://ajax.googleapis.com/ajax/libs/jquery/1.4.4/jquery.min.js">
            </script>
            <link rel="stylesheet" type="text/css" href="estilos.css">
                <div id="slideshow">
                <?php
                    if ($imagens = simplexml_load_file ('imagens.xml')) {
                    foreach ($imagens as $imagem){
                        $path = $imagem->path;
                        $id = $imagem->id;
                        echo"<div>
                       <img src='".$path."' height='480px' width='662px'>
                       </div>";
                        }
                   }//if
                   else echo"<script>alert('ERRO!!! Ficheiro de imagens não
disponivel!!!')</script>";
                    ?>
                </div>
            <script>
            $("#slideshow > div:gt(0)").hide();
            setInterval(function() {
              $("#slideshow > div:first")
                .fadeOut(1000)
                .next()
                .fadeIn(1000)
                .end()
                .appendTo("#slideshow");
            },  10000);
            </script>
            (…)
```

Fig. 6.8 – Código que implementa slide show de imagens

Na Figura 6.8 é apresentado um extrato do código do documento *playerImg.php*, que

permite apresentar as várias imagens periodicamente, neste caso de 10 em 10 segundos. A função em *javascript setInterval* permite a execução periódica de uma função.

A aplicação armazena a descrição das imagens em formato XML [26] num ficheiro no disco local – *imagens.xml*.

```
<imagem>
     <id>1</id>
     <nome>imagem 1</nome>
     <path>imagens/imagem1.jpg</path>
     <desc>Descrição da imagem 1!</desc>
</imagem>
```

Fig. 6.9 – Exemplo de um elemento do ficheiro imagens.xml

Na Figura 6.9 é apresentado um exemplo de um elemento imagem do ficheiro *imagens.xml*, contendo como atributos os seguintes elementos: *id*, *nome*, *path* e *desc*. O atributo *id* é um identificador numérico, gerado automaticamente pelo sistema de forma a identificar cada imagem univocamente. O atributo *nome* corresponde ao nome da imagem. O atributo *path* corresponde ao caminho do local físico em disco, onde se encontra a imagem respetiva. O atributo *desc* corresponde a uma breve descrição da imagem.

A submissão de novas imagens numa instância da aplicação implica, como já foi mencionada na secção 6.2, notificar todas as outras instâncias da aplicação. Deste modo, sempre que é submetida uma nova imagem, a instância correspondente produz uma notificação, notificação que será recebida por todas as instâncias subscritoras. Para isso, numa fase anterior, as instâncias subscritoras devem subscrever o evento correspondente.

```
(…)
$ip = getExternalIP();
$lease_seconds = 86400;
$hub_url = "http://pubsubhubbub.appspot.com/";
$topic_url = "http://".$ip."/Aplicacao_de_teste/eventos_image.xml";
$callback_url = "http://".$ip."/Aplicacao_de_teste/endpoint.php";
$s = new Subscriber();
$s->subscribe($hub_url, $callback_url, $topic_url, $lease_seconds);
```

Fig. 6.10 – Subscrição do feed eventos_image.xml

Na Figura 6.10 é apresentado o fragmento de código que representa a subscrição feita pela aplicação ao tópico correspondente à ocorrência da submissão de novas imagens. Tal como indicado anteriormente, foi utilizado o *Hub* de referência do projeto PubSubHubbub.. O período da subscrição é de um dia (86400 segundos). Concluído este processo, a aplicação

encontra-se em condições de receber todos os eventos associados ao tópico/feed subscrito. O *feed* poderá incluir vários tipos de eventos diferentes, pelo que a aplicação deverá criar um filtro adequado.

```
var myInfo = new Array();//info que se pretende obter
myInfo[0] = "id";
myInfo[1] = "nome";
setInitInfo("endpoint.php", "Nova imagem", "playerImg.php", myInfo);
```

Fig. 6.11 – Extrato de código corresponde ao filtro de conteúdos

Na Figura 6.11 está apresentado o código correspondente à definição de um filtro. O filtro define o tipo de evento e os atributos que se pretende filtrar. Neste exemplo, é construído um *array* com o nome dos atributos que se pretende filtrar, em concreto o *id* e *nome* da imagem. A invocação do método *setInitInfo()* serve para definir um conjunto de parâmetros necessários à execução do método *waitForEvent()*. Os argumentos não são passados pelo método *waitForEvent()*, por otimização do visto que o método *waitForEvent()* executa periodicamente. O argumento *endpoint.php* referencia o endereço do componente que vai processar as notificações de eventos enviadas pelo *hub*, tal como indicado na secção 5.3.5. O segundo argumento corresponde ao tipo de evento que se pretende subscrever, neste caso notificações de novas imagens. O terceiro argumento corresponde ao componente que vai processar o novo conteúdo, após filtragem subscrito pela aplicação.

Após a subscrição do evento e criação do filtro (descritos nas Figuras 6.10 e 6.11), o componente *endpoint.php* será responsável por fazer o tratamento da notificação de novo conteúdo. A componente *playerImg,php* através da invocação do método *waitForEvent()*, obtém a informação da imagem, filtrada segundo as condições impostas. A componente da aplicação *playerImg,php* será responsável por tratar a nova imagem, o que neste caso concreto corresponde a inserir a imagem no *slideshow*. O componente *playerImg*,php produz duas novas páginas Web (atualização do *slideshow* e uma de notificação). No entanto, por razões inerentes ao modelo Web, a página de atualização só é disponibilizada pelo browser quando este faz o refresh da página. Esta questão foi resolvida recorrendo á linguagem JavaScript.

O processo descrito para o componente *playerImage,php* é repetido para o componente *comentarios.php*. O componente *comentarios.php* reutiliza o processo de subscrição de

imagens, neste caso, considerando o evento "Novo comentário".

6.3.3. Submissão de imagens

O componente de submissão de imagens oferece aos utilizadores a funcionalidade de pra submeter uma imagem a uma determinada instância da aplicação. Este componente será responsável por produzir no sistema um evento de "Nova imagem". Após a receção da imagem, este componente cria e publica o evento correspondente.

```
$evt = new Eventos("Notificacao","www.slideshowplayer.pt","Notificacao");
$parametros = array("tipo" => "Nova
imagem", "nome" => $nome_ficheiro, "id" => $id);
$res = $evt->creat_event($parametros, "eventos_image.xml");
```

Fig. 6.12 – Código PHP de criação de evento.

Na 6.12 é possível verificar o código PHP que permite criar um evento quando é efetuada a submissão de uma nova imagem. Inicialmente é necessário criar um objeto do tipo evento que recebe como argumentos o *título*, *link* e descrição da aplicação em causa, tal como indicado na secção 5.3.2. De seguida é criado um *array* com os parâmetros que definem o evento em causa. A invocação do método "creat_event" cria o evento, o que corresponde à criação de uma nova entrada no ficheiro *feed* correspondente, neste caso "eventos_image.xml". A Figura 6.13 é uma ilustração do resultado de uma nova entrada do *feed* "eventos_image.xml".

```
<item>
      <tipo>Nova imagem</tipo>
      <nome>imagem.png</nome>
      <id>35</id>
      <data>2013/10/17-10:15:38</data>
      <origem>188.37.233.183</origem>
</item>
```

Fig. 6.13 – Exemplo de entrada no ficheiro eventos_image.xml

Após a criação do evento, é necessário publicar o mesmo.

```
if($res){
        $p = new Publisher("");
        $hub_url = "http://pubsubhubbub.appspot.com/";
        $topic_url = "http://".$ip."/Aplicacao_de_teste/eventos_image.xml";
        $p1 = $p->publish_event($topic_url, $hub_url);
            echo "<script>alert('Upload  da  imagem  ". $nome_ficheiro. "  com
sucesso!!!');
             window.open('uploadImg.html', '_self');</script>";
              }
              else echo "<script>alert('Erro ao efetuar upload da imagem!');
                  window.open('uploadImg.html', '_self');</script>";

                  window.open('uploadImg.html', '_self');</script>";
```

Fig. 6.14 – Código PHP de publicação de um evento

Na Figura 6.14 é apresentado o código PHP que permite publicar um evento, neste caso do tipo "Nova imagem".

O processo de submissão de comentários é idêntico ao processo descrito na secção anterior.

6.4. Resultados obtidos

A aplicação desenvolvida teve como objetivo efetuar uma prova de conceito sobre o *Framework* desenvolvido. A aplicação pretende comprovar a eficiência do *Framework* em relação aos seguintes requisitos:

- Execução em simultâneo;
- Apresentação do mesmo conteúdo no mesmo instante;
- Reação a eventos;

A aplicação terá que responder a estes requisitos em vários cenários de teste, com diferentes recursos. Os cenários apresentados ao longo do presente capítulo têm como objetivo a maior aproximação possível da realidade de um sistema distribuído e das particularidades que o caracterizam.

6.4.1. Cenário de demonstração inicial

Como já foi indicado anteriormente, numa primeira fase a aplicação foi disponibilizada num único servidor público. Durante esta primeira fase de testes foram criadas três instâncias da aplicação associadas a três ecrãs públicos. Os três ecrãs públicos foram emulados por três *browsers*, dois a executar na mesma máquina onde foi instalada a

aplicação e um browser adicional a executar numa máquina remota.

Entidade	Endereço
Aplicação	http://188.251.75.40//Aplicacao_de_teste/index.html
Submissão comentários	http://188.251.75.40//Aplicacao_de_teste/index_comentarios.html
Submissão de imagens	http://188.251.75.40//Aplicacao_de_teste/uploadImg.html
Feed de comentários	http://188.251.75.40//Aplicacao_de_teste/eventos_coment.xml
Feed de imagens	http://188.251.75.40//Aplicacao_de_teste/eventos_image.xml

Tabela 6.2 – Tabela com endereços do primeiro cenário de testes

Na Tabela 6.3 são apresentadas as entidades e os respetivos endereços utilizados durante a primeira fase de testes. Tal como indicado no capítulo 2.2, os vários requisitos podem ser definidos como sendo um evento. Como tal a aplicação de demonstração demonstra a eficiência da API de Sincronização na modelação de eventos, publicação e subscrição, dotando assim as aplicações de mecanismos capazes de darem resposta a qualquer tipo de requisito de sincronização. Neste primeiro cenário de testes foi adicionado um novo comentário no sistema e de imediato todas as instâncias em execução são notificadas. Na Figura 6.15 são apresentadas as páginas de notificação, geradas na máquina que contém duas instâncias clientes em execução em *browsers* diferentes.

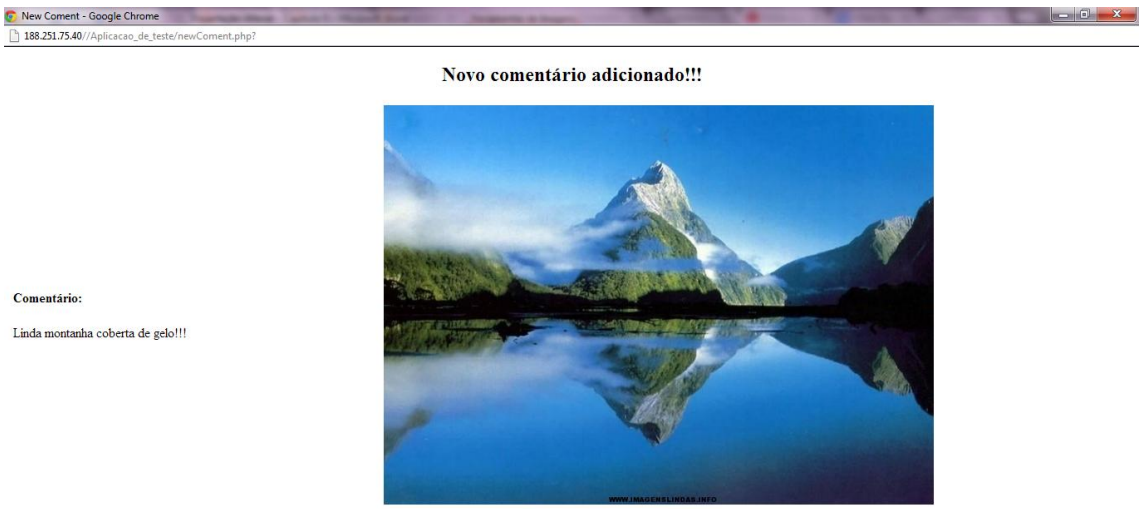


Fig. 6.15 – Notificação de novo comentário

Fig. 6.16 – Notificação de nova imagem.

Já no caso da submissão de uma nova imagem, é apresentada uma notificação como a apresentada na Figura 6.16, em todas as instâncias que se encontrem em execução. Além de visualizar a nova imagem, o utilizador é convidado a adicionar de imediato um comentário. Por outro lado é também indicado o ID da imagem, para o caso de o utilizador pretender efetuar comentários.

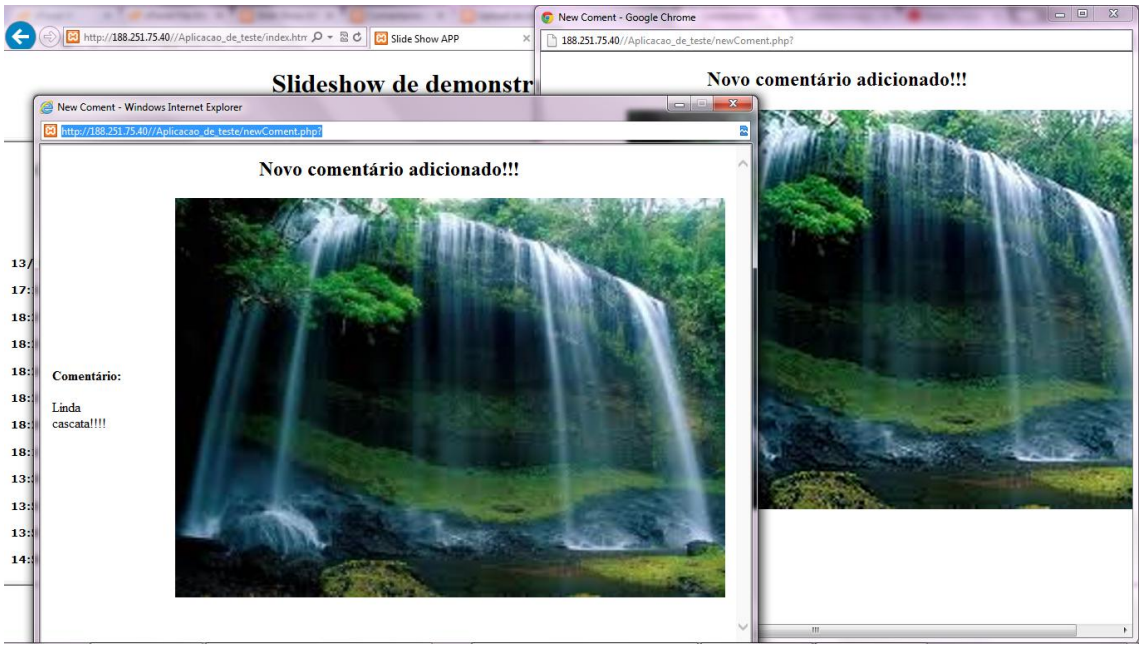


Fig. 6.17 – Notificação de duas instâncias distintas

Na Figura 6.17 é possível visualizar a notificação enviada às duas instâncias que se

encontram a executar na mesma máquina.

Os requisitos de sincronização sãos parcialmente dependentes da rede em que o sistema se encontra inserido e dos recursos computacionais das máquinas. Atrasos na rede podem implicar uma menor eficiência na sincronização, assim como o acesso a recursos computacionais reduzidos. Para avaliar o desempenho deste Framework, foi desenvolvida uma script php com o objetivo de obter tempos médios de sincronização. A script desenvolvida é apresentada na Figura 6.18.

```php
<?php
$file_eventos_date = date ("d/m/Y H:i:s", filemtime('eventos_coment.xml'));
$file_data_date = date ("d/m/Y H:i:s", filemtime('data.xml'));
echo "Eventos: ".$file_eventos_date."<br>";
echo "Data: ".$file_data_date."<br>";
$dif  = filemtime('eventos_coment.xml') - filemtime('data.xml');
echo "Tempo de sincronização: ".$dif;
?>
```

Fig. 6.18 – Script PHP de cálculo do tempo de sincronização

Com base em valores obtidos ao longo de vários testes diários efetuados durante vários dias, em horários diferentes foi possível determinar que com os recursos básicos utilizados é possível obter um intervalo de sincronização entre um a três segundos. Com este primeiro cenário foi possível provar que com diferentes instâncias a correr no mesmo ou em diferentes máquinas, a aplicação reage às ocorrências surgidas ao longo da execução e notifica todas as instâncias.

6.4.2. Segundo cenário de testes

A segunda fase de testes surgiu na necessidade de introduzir algumas características dos sistemas distribuídos, como é o caso de sistemas replicados e fragmentados. Foi necessário introduzir um novo servidor no sistema de forma a criar um cenário em que a aplicação é instanciada em servidores públicos independentes. Neste cenário, a aplicação está disponível em dois servidores públicos: o servidor já existente no primeiro cenário de testes e um segundo servidor externo (www.x10hosting.com). Com esta segunda fase de testes é pretendido avaliar o desempenho do *Framework* num contexto mais próximo do real em que instalações independentes de ecrãs podem associar instâncias de aplicações a executar em domínios da Internet diferentes. Na 6.4 encontram-se as entidades e respetivos endereços utilizados na segunda fase de testes. De salientar que as entidades denominadas

de “1” entendem-se pelas que se encontram alojados no servidor da primeira fase de testes. As restantes representam as que se encontram no servidor externo.

Entidade	Endereço
Aplicação 1	http://188.251.75.40//Aplicacao_de_teste/index.html
Submissão comentários 1	http://188.251.75.40//Aplicacao_de_teste/index_comentarios.html
Submissão de imagens 1	http://188.251.75.40//Aplicacao_de_teste/uploadImg.html
Feed de comentários	http://slideshowapp.x10.mx/Aplicacao_de_teste/eventos_coment.xml
Feed de imagens	http://slideshowapp.x10.mx/Aplicacao_de_teste/eventos_image.xml
Aplicação 2	http://slideshowapp.x10.mx/Aplicacao_de_teste/index.html
Submissão comentários 2	http://slideshowapp.x10.mx/Aplicacao_de_teste/index_comentarios.html
Submissão de imagens 2	http://slideshowapp.x10.mx/Aplicacao_de_teste/uploadImg.html

Tabela 6.3 – Endereços e entidades presentes no segundo cenário de testes

Neste caso temos informação replicada, nomeadamente as estruturas de dados relativas às imagens e comentários submetidos ao sistema. Ambas as instâncias servidoras dispõem de estruturas de dados locais, o que adiciona alguma complexidade ao sistema visto haver a necessidade de manter as estruturas de dados coerentes. Por exemplo, se for adicionado um comentário através da instância servidora A, não constará na estrutura de dados da instância B, até que a mesma seja notificada. Neste cenário além da sincronização de todas as instâncias clientes é necessário também sincronizar as instâncias servidoras, de forma a manter as estruturas de dados coerentes.

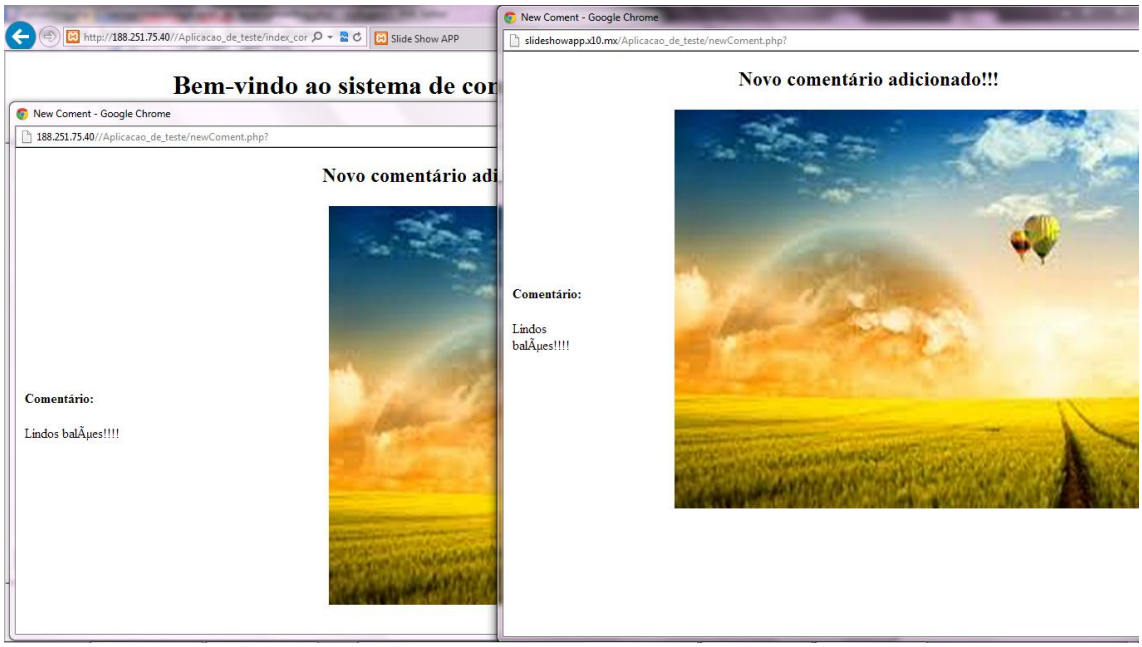


Fig. 6.19 – Notificação clientes ligados a instâncias distintas

Na Figura 6.19 é possível verificar que um novo comentário foi submetido ao sistema e ambos os clientes receberam a notificação. Um dos clientes encontrava-se a executar a aplicação alojada no servidor externo e o segundo cliente encontra-se a executar a aplicação da segunda instância servidora (utilizada já na primeira fase de testes). Como anteriormente foi descrito nesta segunda fase de testes, além da preocupação com a notificação de todas as instâncias em execução é necessário manter as estruturas de dados coerentes. Nesse sentido, sempre que uma instância servidora é notificada para a ocorrência de um determinado evento, irá atualizar as suas estruturas de dados locais.

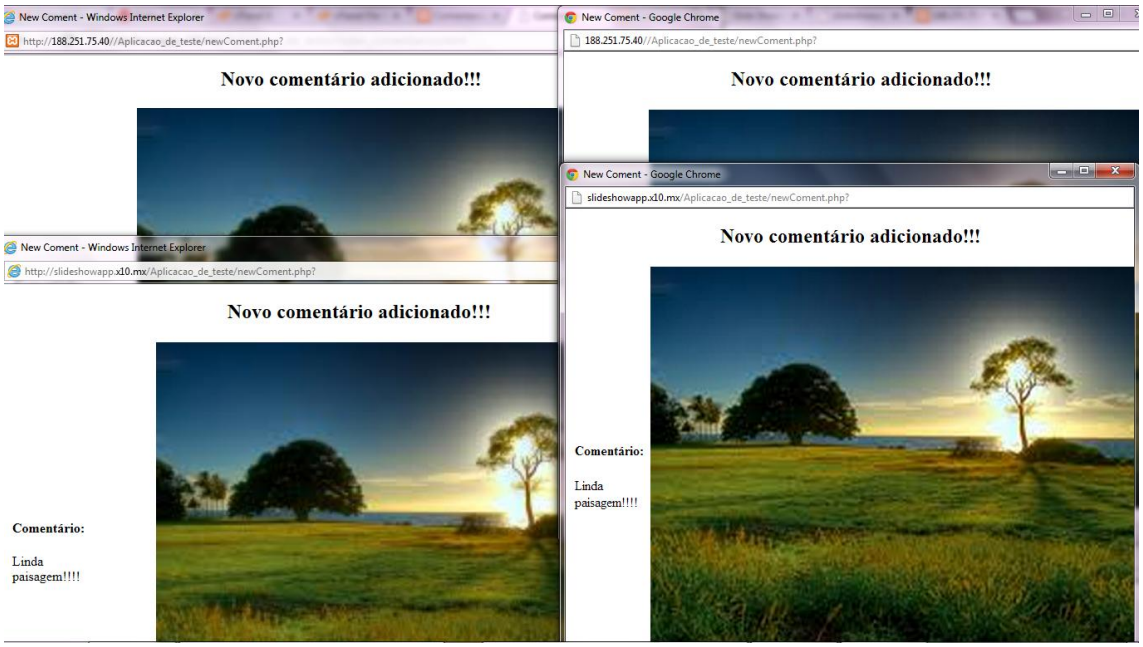


Fig. 6.20 – Notificação duas instancia clientes com duas instâncias servidoras

Num segundo teste é introduzido um pouco mais de complexidade e utiliza-se duas instâncias servidoras e dois clientes ligados em simultâneo a cada instância servidora. Na Figura 6.20 é apresentada a notificação de um novo comentário a todas as instâncias clientes gerada pelo *browser* do terminal em causa. As notificações enviadas pelo *Hub* são apenas entregues às aplicações que as subscreveram, posteriormente é que as aplicações processam a informação recebida e produzem as novas páginas como as que são apresentadas na Figura 6.20. No caso apresentado são apresentados quatro terminais, sendo que correspondem duas páginas a cada servidor. Dessa forma é possível verificar, que independentemente do servidor que foi utilizado para submeter o comentário, o outro servidor foi igualmente notificado. De uma forma similar à primeira fase de testes, foram feitas medições de desempenho do *Framework*. Nesta segunda fases foram registadas melhorias. Os tempos de sincronização não ultrapassaram um segundo, possivelmente em alguns caso até foi bastante menor. De salientar mais uma vez que o desempenho global do sistema é bastante dependente do estado da rede. Na medida que em caso de picos na rede, fraca largura de banda, problemas de acesso irão comprometer o desempenho da *Framework*. Por outro lado a instância servidora externa utilizada é uma versão gratuita o que não confere grande eficiência ao serviço, tornando o sistema ainda mais vulnerável ao desempenho do mesmo. Idealmente seria desejável efetuar os testes e o desenvolvimento em servidores dedicados, com grande capacidade de processamento e ao mesmo tempo boa largura de banda.

7. Conclusões

As redes de ecrãs públicos estão a surgir nos mais diversos espaços públicos. Cada vez mais se sente a sua presença nos centros das cidades, estádios, museus, edifícios públicos, centros comerciais, etc. Dada a grande expansão da área e a crescente descida dos custos de *hardware*, novas oportunidades de desenvolvimento do mais variado número de aplicações não param de surgir, sendo estas cada vez em maior escala. No futuro, a interligação entre instalações de ecrãs públicos, mesmo que independentes e remotos, irá promover não só novos tipos de interação situada, mas também interações sincronizadas entre as aplicações a executar em cada instalação. Cada instalação e os seus componentes irão provavelmente ser geridos por gestores independentes, partilhando apenas o modelo de publicação e distribuição de aplicações e os servidores públicos que disponibilizam a aplicação.

Neste trabalho de mestrado foi feito uma análise dos requisitos de sincronização em vários cenários de redes de ecrãs. Foi também analisado o estado da arte em infraestruturas para ecrãs públicos, nomeadamente os projetos *Instante Places*, *E-Campus* e o *UBI-Hotspot*. De entre os projetos analisados, apenas o projeto *E-Campus* aborda o problema de sincronização entre aplicações em ecrãs públicos. No entanto, aborda esta questão no contexto de instalações geridas por uma mesma entidade, tendo-se analisado apenas problemas relativos à partilha de recursos, (ecrãs, projetores, etc). Foram também analisadas outras abordagens à sincronização em sistemas distribuídos. Tendo em conta a natureza Web das aplicações, soluções baseadas em mecanismo adequados para redes locais tenderiam a não ser escaláveis.

Neste contexto, este trabalho de mestrado propôs um *Framework* – um modelo de sincronização e uma API (*Aplication Programming Interface*) – para programadores de aplicações para ecrãs públicos. Este *Framework* é baseado no PubSubHubbub, um protocolo de comunicação distribuída na Internet, escalável, baseado no modelo Produtor/Subscritor. Cada instalação de uma aplicação será um produtor e/ou um subscritor, sendo que sincronizarão as suas ações segundo o modelo de eventos. A API disponibiliza mecanismos que permitem às aplicações, criar, publicar, subscrever e filtrar e ficando a definição do evento e o processamento do mesmo ao cargo das aplicações. Com o modelo de eventos desenvolvido neste projeto, é possível modelar qualquer tipo de informação. Dessa forma, torna-se possível sincronizar, em qualquer situação, sem necessidade de sincronização de relógios, as diferentes atividades das aplicações. Por outro

lado, a API permite obter um sistema descentralizado e escalável, uma vez que as diferentes entidades poderão estar espalhadas por diferentes servidores na Internet.

Como prova de conceito foi desenvolvida uma aplicação constituída por três componentes: apresentação de conteúdos, submissão de comentários e submissão de imagens; e foi criado um sistema de demonstração formado por diferentes instâncias da aplicação e três instalações de ecrãs (os ecrãs foram emulados por *browser Web*). A Aplicação foi desenvolvida de forma a gerar um conjunto de cenários que permitam comprovar a funcionalidade e eficácia da API. De forma a tornar os cenários mais próximos de contextos reais foram utilizados dois servidores durante os testes, permitindo a replicação da aplicação.

Efetuados os testes, e numa perspetiva de análise de desempenho, foi possível verificar que dada a natureza Web da aplicação e da API, os resultados de desempenho dependem dos recursos de processamento e da largura de banda dos servidores. No entanto, durante a utilização de dois servidores, com várias instâncias clientes ligadas a ambos os servidores, foi possível notificar todas as instâncias das ocorrências com um atraso na ordem de um segundo. Devido a limitações temporais e de recursos, não foi possível efetuar outro tipo de medições ou testes.

Um *Framework* baseado nos protocolos de comunicação Produtor/Subscritor no domínio da Internet, oferece uma solução para sincronização de aplicações em ecrãs públicos, com um desempenho adequado para cenários que toleram atrasos decorrentes do modelo inerente à Internet. Adicionalmente este *Framework* dá suporte aos cenários de referência em sincronização para ecrãs públicos e oferece um modelo e API que sustentam a natureza aberta e escalável das redes de ecrãs. Este trabalho não está claramente completo. Será necessário analisar mais cenários de sincronização e terá de ser estudado o modelo de integração deste *Framework* na arquitetura genérica de redes de ecrãs públicos e a sua relação com modelos de publicação e distribuição de aplicações.

Adicionalmente, e ao nível da conceção do *Framework*, deverá ser analisado a implementação de *Hubs* e redes de *Hubs* com objetivo de melhorar o desempenho dos sistemas, pois seria permitido às aplicações publicarem e subscreverem eventos em diferentes *Hubs*.

Finalmente, ao nível da implementação da API, e mais concretamente do lado do cliente Web da aplicação – método *waitForEvent* – o trabalho futuro passará por estudar

alternativas ao mecanismo de *polling*, nomeadamente através da utilização de mecanismos oferecidos pela especificação do HTML5.

8. Bibliografia

[1] R. Jose, H. Pinto, B. Silva, and A. Melro, “Pins and posters: Paradigms for content publication on situated displays,” *Computer Graphics and Applications, IEEE*, vol. 33, no. 2. pp. 64–72, 2013.

[2] R. José, H. Pinto, B. Silva, A. Melro, and H. Rodrigues, “Beyond interaction: tools and practices for situated publication in display networks,” in *Proceedings of the 2012 International Symposium on Pervasive Displays*, 2012, pp. 8:1–8:6.

[3] O. Storz, A. Friday, N. Davies, J. Finney, C. Sas, and J. G. Sheridan, “Public Ubiquitous Computing Systems: Lessons from the e-Campus Display Deployments,” *Pervasive Computing, IEEE*, vol. 5, no. 3. pp. 40–47, 2006.

[4] O. Storz, A. Friday, and N. Davies, “Supporting content scheduling on situated public displays,” Comput. Graph., vol. 30, no. 5, pp. 681–691, Oct. 2006.

[5] T. Ojala, H. Kukka, T. Lindén, T. Heikkinen, M. Jurmu, S. Hosio, and F. Kruger, “UBI-Hotspot 1.0: Large-Scale Long-Term Deployment of Interactive Public Displays in a City Center,” *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*. pp. 285–294, 2010.

[6] T. Ojala, V. Valkama, H. Kukka, T. Heikkinen, T. Lindén, M. Jurmu, F. Kruger, and S. Hosio, “UBI-hotspots: sustainable ecosystem infrastructure for real world urban computing research and business,” in *Proceedings of the International Conference on Management of Emergent Digital EcoSystems*, 2010, pp. 196–202.

[7] PD-NET Consortium. Deliverable D2.1 - scientific evaluation of pervasive display network prototype, 2012.

[8] C. Taivan and R. José. An application framework for open application development and distribution in pervasive display networks. In *On the Move to Meaningful Internet Systems*: *OTM 2011 Workshops*, pages 21-25. Springer, 2011.

[9] H. R. Constantin Taivan, Rui José and B. Silva. Situatedness for global display web apps, 2013.

[10] S. Clinch, N. Davies, T. Kubitza, and A. Schmidt, “Designing application stores for public display networks,” in *Proceedings of the 2012 International Symposium on Pervasive Displays*, 2012, pp. 10:1–10:6.

[11] S. Clinch, J. Harkes, A. Friday, N. Davies, and M. Satyanarayanan. How close is close enough? understanding the role of cloudlets in supporting display appropriation by mobile users. In *Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference* on, pages 122-127. IEEE, 2012.

[12] S. Tarkoma, J. Kangasharju, T. Lindholm, and K. Raatikainen, "Fuego: Experiences with Mobile Data Communication and Synchronization," *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*. pp. 1–5, 2006.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[14] G. B. G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems - Concepts and design*, 5th ed. 2011.

[15] RSS 2.0 Specification (version 2.0)". Disponível em: *http://www.rssboard.org/rss-specification*, visualizado pela última vez a 28 de Outubro de 2013.

[16] M. Nottingham and R. Sayre, "The Atom Syndication Format". Disponível em *http://tools.ietf.org/html/rfc4287*, consultado em 28 de Outubro de 2013.

[17] "ComparingProtocols - pubsubhubbub - Comparison of PubSubHubbub to light-pinging protocols - A simple, open, webhook based pubsub protocol & open source reference implementation. - Google Project Hosting.". Disponível em: *http://code.google.com/p/pubsubhubbub/wiki/ComparingProtocols*, consultado em 28 de Outubro de 2013.

[18] "pubsubhubbub - A simple, open, webhook based pubsub protocol & open source reference implementation.". Disponível em: http://code.google.com/p/pubsubhubbub/, consultado em 28 de Outubro de 2013.

[19] P. Millard, P. Saint-Andre, and R. Meijer, "Publish-Subscribe." XMPP Standards Foundation, 12-Jul-2010. Disponível em: *http://xmpp.org/extensions/xep-0060.html*, consultado em 28 de Outubro de 2013.

[20] "simpleupdateprotocol - Simple Update Protocol (SUP) - a feed update protocol - Google Project Hosting.". Disponível em : http*://code.google.com/p/simpleupdateprotocol/*, consultado em 28 de Outubro de 2013.

[21] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)." Disponível em*: http://tools.ietf.org/html/rfc4627*, consultado em 28 de Outubro de 2013.

[22] J. Cascio, “SLAP – Simple Lightweight Announcement Protocol.” Disponível em: *http://joecascio.net/joecblog/2009/05/18/announcing-slap/*, consultado em 28 de Outubro de 2013.

[23] “Weblogs.com : Weblogs.”. Disponível em: *http://weblogs.com/*, consultado em 28 de Outubro de 2013.

[24] “Hypertext Transfer Protocol - HTTP/1.1.”. Disponível em: *http://www.w3.org/Protocols/rfc2616/rfc2616.html*, consultado em 28 de Outubro de 2013.

[25] “Google Pubsubhubbub Hub.”. Disponível em: http://pubsubhubbub.appspot.com/, consultado em 28 de Outubro de 2013.

[26] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible Markup Language (XML) 1.0 (Fifth Edition).” Disponível em: *http://www.w3.org/TR/REC-xml/* , consultado em 28 de Outubro de 2013.

Anexo A

XML

XML (**E**xtensible **M**arkup **L**anguage) [26] é um formato de dados de texto muito simples e flexível, capaz de modelar qualquer tipo de dados. Todo o documento se encontra organizado por hierarquias, sendo que contém um conjunto de tags em que cada tag irá conter um determinado conteúdo.

```
<?xml version="1.0"?>
<nota>
    <para>Mario</para>
    <de>Manuel</de>
    <titulo>Lembrete</titulo>
    <mensagem>Não te esqueças do que te pedi!!</mensagem>
</nota>
```

Fig. A.1 – Exemplo de ficheiro XML

Na Figura A.1 é possível analisar um pequeno exemplo de um ficheiro XML, sendo que o elemento raiz é denominado por "nota". Esse mesmo elemento é constituído por quatro outros elementos: para, de, titulo e mensagem. Todas as tags têm que respeitar a forma "<nome> conteudo </nome>", de salientar que o XML é case *sensitive* ou seja distingue maiúsculas de minúsculas. Um elemento poderá ser constituído por um conjunto de outros elementos, como também poderá conter um conjunto de parâmetros como o exemplo da Figura A.2.

```
<?xml version="1.0"?>
<nota hora=”15:00”>
    <para>Mario</para>
    <de>Manuel</de>
    <titulo>Lembrete</titulo>
    <mensagem>Não te esqueças do que te pedi!!</mensagem>
</nota>
```

Fig. A.2 – Exemplo de ficheiro XML com parâmetros

Na Figura A.2 é possível verificar o mesmo exemplo da Figura A.1, com a diferença de que o elemento “nota” contém um parâmetro denominado “hora”, com o valor “15h00”. Os parâmetros permitem fornecer informação adicional sobre os elementos. Por exemplo da descrição de uma receita de um bolo, cada elemento indicará os diferentes passos da elaboração, por sua vez os parâmetros de cada elemento indicarão por exemplo tempo de amassadura, tempo de cozedura, etc. Com tal flexibilidade e relativa simplicidade na modelação de qualquer tipo de dados, o XML veio ganhar grande importância no mundo web. Tal advém de em ambientes web ser trocada imensa informação nos mais diversos formatos, entre sistemas com características e recursos muito diferentes. Tal característica torna complexo a troca de dados entre elementos, em alguns casos seria mesmo impossível. O XML veio criar um formato “standard” capaz de modelar qualquer tipo de informação em qualquer ambiente.

RSS

A tecnologia RSS [15] permite a agregação de conteúdo web, normalmente gerado por blogs ou páginas web de notícias. Todos os ficheiros RSS são modelados em XML e devem respeitar a sintaxe imposta pela versão 1.0., descrita no início do presente anexo A. Cada página terá associado a si uma *feed*, que não é mais do que um formato de dados. O *feed* é disponibilizado através de um link, através do qual as entidades interessadas podem manifestar interesse nas atualizações do mesmo. Um feed por sua vez não é mais do que um ficheiro contendo uma lista com as atualizações da página ou do blog a que está associado.

Fig. A.3 – Ilustração do icon de um feed

Na Figura A.3 é possível verificar o icon que simboliza que a página ou blog que estamos a visitar dispõe de um endereço *feed*. Clicando nesse icon iremos obter o ficheiro *feed* associado à página e o respetivo endereço do f*eed*. Através de uma terceira entidade denominada de agregador de *feeds*, um utilizador pode subscrever todas as alterações registadas nas suas páginas e blogs preferidos. Além de facilitar a vida aos utilizadores, permite também em alguns casos manter em segundo plano de uma página um resumo das notícias ou do estado do trânsito, através da subscrição de uma página que disponha de tal conteúdo. A subscrição de um *feed* ou anulação do mesmo é um processo simples, consiste apenas adicionar ou remover o link do *feed* ao agregador, sem necessitar de qualquer confirmação de nenhuma entidade. Para que todo este processo funcionar, é necessário que o *feed* rss respeite um conjunto de regras.

Elemento	**Descrição**
Title	Título do canal. Caso o *feed* contenha a mesma informação que a página web a que está associado, o titulo deverá ser o link da página
Link	URL da página ou blog a que está associado
Description	Descrição do canal

Tabela A.1 – Elementos obrigatórios de um canal

Dado que se trata de um ficheiro XML, terá que respeitar as regras de um ficheiro XML e ao mesmo tempo respeitar a estrutura de um feed, que contém um conjunto de campos obrigatórios. Dessa forma o primeiro elemento de um *feed* será um uma tag indicando que se trata de um ficheiro rss e a sua respetiva versão. O elemento seguinte será denominado de *channel* que contém a si associado um conjunto de outros elementos que representam

informação sobre o mesmo. Assim sendo existe um conjunto de elementos obrigatórios que serão apresentados na Tabela A.1.

Elemento	Descrição
language	Linguagem do canal
copyrigth	Notificação de direitos de autor
managingEditor	Endereço de e-mail do responsável editor
webMaster	Endereço de e-mail do responsável por problemas tecnicos
pubDate	Data de publicação do conteúdo
lastBuildDate	Ultima data de modificação
category	Especifica as categorias do canal
generator	String indicativa do programa utilizado para gerar o canal
docs	Link indicando documentação sobre o formato RSS utilizado
cloud	Permite a associação a outra entidade que notifica das alterações do canal
ttl	Número de minutos que poderá estar em cache antes de ser atualizado pela fonte
image	Imagem que pode ser apresentada pelo canal
rating	Classificação das PICS[4] do canal
textInput	Especifica o conteúdo de texto que pode ser apresentado com o canal
skipHours	Indicação de que intervalos de horas os agregadores podem utilizar
skipDays	Indicação de que intervalos de dias os agregadores podem utilizar

Tabela A.2 – Elementos opcionais do elemento canal

[4] Platform for Internet Content Selection (PICS)

Por outro lado além dos elementos obrigatórios existe um conjunto de elementos opcionais que estão apresentados na Tabela A.2.

Por sua vez um canal poderá conter um conjunto de elementos denominados de *item*. Ao contrário do canal, não tem qualquer item obrigatório, no entanto deverá sempre ter uma descrição ou um título. Na Tabela três é possível verificar o conjunto de elementos opcionais.

Elemento	**Descrição**
Title	Título do item
Link	Link do item
Description	Descrição do item
author	Endereço de e-mail do autor
category	Indica a categoria do item
comments	Endereço da página onde podem adicionar comentários do item
enclosure	Descreve os objetos mídia associados ao item
guid	String que identifica univocamente o item
pubDate	Indica a data de publicação do item
source	Canal rss a que o item está associado

Tabela A.3 – Elementos opcionais de um item

```
<rss version="2.0">
<channel>
<title>Liftoff News</title>
<link>http://liftoff.msfc.nasa.gov/</link>
<description>Liftoff to Space Exploration.</description>
<language>en-us</language>
<pubDate>Tue, 10 Jun 2003 04:00:00 GMT</pubDate>
<lastBuildDate>Tue, 10 Jun 2003 09:41:01 GMT</lastBuildDate>
<docs>http://blogs.law.harvard.edu/tech/rss</docs>
<generator>Weblog Editor 2.0</generator>
<managingEditor>editor@example.com</managingEditor>
```

```
<webMaster>webmaster@example.com</webMaster>
<item>
<title>Star City</title>
<link>
http://liftoff.msfc.nasa.gov/news/2003/news-starcity.asp
</link>
<description>
How do Americans get ready to work with Russians aboard the International
Space Station? They take a crash course in culture, language and protocol at
Russia's <a href="http://howe.iki.rssi.ru/GCTC/gctc_e.htm">Star City</a>.
</description>
<pubDate>Tue, 03 Jun 2003 09:39:21 GMT</pubDate>
<guid>
http://liftoff.msfc.nasa.gov/2003/06/03.html#item573
</guid>
</item>
<item>
<description>
Sky watchers in Europe, Asia, and parts of Alaska and Canada will experience a
<a
href="http://science.nasa.gov/headlines/y2003/30may_solareclipse.htm">partial
eclipse of the Sun</a> on Saturday, May 31st.
</description>
<pubDate>Fri, 30 May 2003 11:06:42 GMT</pubDate>
<guid>
http://liftoff.msfc.nasa.gov/2003/05/30.html#item572
</guid>
</item>
<item>
<title>The Engine That Does More</title>
<link>
http://liftoff.msfc.nasa.gov/news/2003/news-VASIMR.asp
</link>
<description>
Before man travels to Mars, NASA hopes to design new engines that will let us
fly through the Solar System more quickly. The proposed VASIMR engine would do
that.
</description>
```

```
<pubDate>Tue, 27 May 2003 08:37:32 GMT</pubDate>
<guid>
http://liftoff.msfc.nasa.gov/2003/05/27.html#item571
</guid>
</item>
<item>
<title>Astronauts' Dirty Laundry</title>
<link>
http://liftoff.msfc.nasa.gov/news/2003/news-laundry.asp
</link>
<description>
Compared to earlier spacecraft, the International Space Station has many
luxuries, but laundry facilities are not one of them. Instead, astronauts have
other options.
</description>
<pubDate>Tue, 20 May 2003 08:56:02 GMT</pubDate>
<guid>
http://liftoff.msfc.nasa.gov/2003/05/20.html#item570
</guid>
</item>
</channel>
</rss>
```

Fig. A.4 – Exemplo de um ficheiro feed RSS [15]

Na Figura A.4 é possível visualizar um ficheiro *feed* em que contém um conjunto de items com os respetivos elementos opcionais. RSS poderá não ser perfeito, no entanto contém imensas vantagens é utilizado em larga escala.

Atom

Atom ao contrário do RSS não é uma sigla, no entanto é também um formato de dados gerados por páginas web e blogs. De uma forma similar ao RSS, é também baseado em XML e deve respeitar as duas regras. Cada página web ou blog terá assim associado um *feed* que por sua vez contém um endereço. Um *feed* Atom da mesma forma que um *feed* RSS não é mais do que um ficheiro XML.

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
   <title>Example Feed</title>
  <link href="http://example.org/"/>
  <updated>2003-12-13T18:30:02Z</updated>
  <author>
    <name>John Doe</name>
  </author>
  <id>urn:uuid:60a76c80-d399-11d9-b93C-0003939e0af6</id>

        <entry>
          <title>Atom-Powered Robots Run Amok</title>
          <link href="http://example.org/2003/12/13/atom03"/>
          <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-
80da344efa6a</id>
          <updated>2003-12-13T18:30:02Z</updated>
          <summary>Some text.</summary>
        </entry>
      </feed>
```

Fig. A.5 – Exemplo de feed Atom [16]

Na Figura A.5 é possível analisar um *feed* Atom básico. Sendo que qualquer *feed* Atom é constituído por um conjunto de elementos que constituem a meta data obrigatória e posteriormente N entradas.

Elemento	Descrição
id	Identifica o feed com uma URI única e permanente
title	Contém o título atribuído ao feed
updated	Indica o data-hora da ultima alteração

Tabela A.4 – Elementos obrigatórios de um feed Atom

Elemento	Descrição
author	Nome(s) do(s) autor(es)
link	Link da página web ou blog a que o feed está associado

Tabela A.5 – Elementos aconselhados de um feed Atom

Elemento	Descrição
category	Descreve a categoria do feed
contributor	Indica os nomes de quem contribui para o feed
generator	Indica o software que gera o feed
icon	Representa um icon que permite identificar o feed visualmente
logo	Representa a imagem que permite identificar o feed visualmente
rigths	Informação sobre direitos de autor
subtitle	Representa o subtítulo do feed

Tabela A.6 – Elementos opcionais de um feed Atom

Ao contrário de um *feed* RSS, Atom contém três conjuntos de elementos, sendo eles: obrigatórios, recomendados e opcionais. Os três grupos e os respetivos elementos estão descritos respetivamente pelas Tabelas A.4, A.5 e A.6.

Elemento	Descrição
author	Nome do(s) autor(es)
content	Link das páginas com o conteúdo na íntegra
link	Identifica o link da página a que está associada a entrada
summary	Apresenta um breve resumo da entrada

Tabela A.7 – Elementos recomendados de uma entrada de um feed Atom

No entanto cada *feed* poderá ter um ou mais entradas, sendo que cada entrada também contém um conjunto de elementos. Ao contrário dos *items* de um *feed* RSS, contém elementos obrigatórios, tornando o *feed* mais robusto mas ao mesmo tempo mais complexo. Os elementos obrigatórios de uma entrada são os mesmos apresentados na Tabela A.4, no

entanto referindo-se à entrada. Já no que diz respeito aos elementos recomendados, estão apresentados na Tabela A.7.

Elemento	Descrição
category	Especifica a categoria da entreda
contributor	Especifica o(s) autor(es) da entrada
published	Contém o momento que foi publicado a entrada
source	Se a entrada for cópia de outro feed, a meta data do feed original deverá ser indicada
rigths	Apresenta direitos de autor dos dados da entrada

Tabela A.8 – Elementos opcionais de uma entrada de um feed Atom

Na Tabela A.8 são apresentados os elementos opcionais de uma entrada de um *feed* Atom.

A estrutura de um *feed* Atom é bastante mais elaborada do que um feed RSS o que em alguns casos poderá complicar a modelação da informação com a obrigatoriedade de preenchimento de demasiados parâmetros. Por outro lado contém uma grande vantagem visto que permite adicionar elementos de *feeds* RSS.

Durante o desenvolvimento do projeto foram apenas utilizados *feeds* RSS devido a sua maior flexibilidade no que diz respeito a construção dos mesmos. Tal característica advém de não conterem quaisquer campos obrigatórios nos *items* presentes num *feed*. Dada a natureza do projeto e a impossibilidade de prever que tipos de dados podem ser modelados, torna-se importante usar a menos informação possível. Só assim se poderá construir uma API simples que qualquer aplicação possa utilizar.