INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

LightStream – Sistema de comunicação *Publish/Subscribe* com *WebSockets*

MANUEL VARGAS FELÍCIO

(Licenciado)

Trabalho Final de Mestrado para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientador:

Professor Jorge Manuel Rodrigues Martins Pião

Júri:

Presidente: Professor Walter Vieira

Vogais:

Professor Pedro Félix

Professor Jorge Manuel Rodrigues Martins Pião

Setembro de 2012

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

LightStream – Sistema de comunicação *Publish/Subscribe* com *WebSockets*

MANUEL VARGAS FELÍCIO

(Licenciado)

Trabalho Final de Mestrado para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientador:

Professor Jorge Manuel Rodrigues Martins Pião

Júri:

Presidente: Professor Walter Vieira

Vogais:

Professor Pedro Félix

Professor Jorge Manuel Rodrigues Martins Pião

Setembro de 2012

Índice

Índice de figuras

Índice de listagens

1 INTRODUÇÃO

No presente capítulo, pretende-se dar a conhecer a motivação para o desenvolvimento deste trabalho, bem como os objetivos propostos para a realização do mesmo.

1.1 MOTIVAÇÃO

Este projeto consiste na implementação de uma plataforma que permita a várias aplicações *Web*, ou outro tipo de clientes que a plataforma venha a suportar, a publicação e subscrição de conjuntos de informação que sejam do seu interesse.

Atualmente, grande parte de aplicações *Web* que pretendem notificar o utilizador de alterações que tenham ocorrido sobre informação relevante fazem-no através de *polling*, isto é, a aplicação cliente questiona periodicamente o servidor sobre alterações aos dados a visualizar, sem ter a certeza de que tenha havido de facto alterações nos dados. Isto resulta em carga desnecessária sobre o servidor, bem como tráfego desnecessário para o cliente. A plataforma proposta neste projeto pretende colmatar este problema, através do protocolo *WebSocket* (1), tirando partido da existência de uma ligação bidirecional entre servidor e cliente com a capacidade de fazer *push* de informação relevante para o cliente, no momento em que a informação é publicada.

O protocolo *WebSocket* é suportado pela maioria dos *Web browsers* mais recentes, fazendo parte da especificação do *HTML 5* (2).

O nome LightStream foi escolhido para esta plataforma devido à sua preocupação com a utilização de poucos recursos do lado do cliente (*light*) e à existência de um canal bidirecional de comunicação entre o cliente e o servidor (*stream*).

1.2 OBJETIVOS GERAIS

Este projeto tem como principal objetivo a disponibilização de uma plataforma extensível e escalável, que permite a aplicações publicarem/subscreverem conjuntos de informação que sejam do seu interesse, através do protocolo *WebSocket*.

Com esta plataforma é disponibilizada um conjunto de APIs para que aplicações cliente desenvolvidas em *HTML 5/Javascript* e *.NET* possam tirar partido das funcionalidades da mesma.

Sendo a redução de tráfego na rede um dos objetivos da plataforma, é também disponibilizada uma API para a especificação de filtros na subscrição de informação que são avaliados no servidor, o que implica que quem subscreve informação tenha a garantia de que a informação que recebe é sempre útil.

Para utilizar esta plataforma deverá ser necessário possuir uma conta de utilizador e utilizar a API disponível para a tecnologia em que se pretende tirar partido da mesma.

A plataforma permite ainda que uma aplicação possa partilhar informação com outras aplicações, facilitando cenários de integração de sistemas. Esta funcionalidade é também interessante para aplicações *Web* que recorram extensivamente a *Web Services* públicos, na medida em que é possível criar aplicações que chamem, periodicamente, operações destes serviços e quando houver novos dados possam publicá-los na plataforma. Assim sendo, será a plataforma a informar os clientes interessados em receber novos dados vindos destes *Web Services*, sem que haja a necessidade de cada cliente da aplicação *Web* realizar pedidos aos serviços. Desta forma, consegue-se reduzir drasticamente o impacto nos servidores onde estão publicados estes serviços, bem como a utilização de recursos por parte dos clientes. Quanto mais serviços deste género estiverem disponíveis na plataforma, maior será o interesse por quem desenvolve aplicações em tirar partido da mesma.

1.3 ORGANIZAÇÃO DO DOCUMENTO

Este documento está dividido em capítulos que refletem a seguinte estrutura:

1. Introdução – Constitui o presente capítulo. Contém a motivação para o desenvolvimento deste trabalho, os objetivos gerais, a organização do documento e a lista de acrónimos utilizados
2. Enquadramento – Trata-se do capítulo onde é feito um estudo em volta do tema do trabalho, englobando conceitos essenciais para a compreensão da solução final.
3. Solução – Contém a arquitetura geral da solução proposta, fazendo uma primeira aproximação na descrição de tecnologias utilizadas.
4. Implementação – Contém a descrição dos principais pontos de desenvolvimento. Será acompanhada por diagramas e listagens de código sempre que necessário.
5. Trabalho futuro – Apresenta um conjunto de sugestões que poderiam ser realizadas de forma a melhorar o trabalho.
6. Conclusão – Neste capítulo são apresentadas as conclusões finais.
7. Referências – Contém uma listagem das principais referências consultadas ao longo da realização deste trabalho.

8. Anexos – Contém diagramas e listagens de código que ajudam na compreensão de determinados conceitos ou técnicas de implementação.

1.4 LISTA DE ACRÓNIMOS

- JSON – Javascript Object Notation
- AJAX – Asynchronous Javascript and XML
- HTTP – Hypertext Transfer Protocol
- HTML – Hypertext Markup Language
- SSL – Secure Socket Layer
- API – Application Programming Interface
- TPL – Task Parallel Library
- WCF – Windows Communication Foundation
- *0MQ* – *ZeroMQ* ou *ZMQ*

2 ENQUADRAMENTO

Durante as duas últimas décadas, a maioria dos *Web sites* foi desenvolvida em torno do paradigma pedido/resposta através do protocolo HTTP. Um utilizador abre uma página *Web* e nada acontece até o utilizador navegar para a próxima página. Por volta de 2005, a utilização da técnica AJAX ajudou a tornar os *Web sites* um pouco mais dinâmicos. Ainda assim, toda a comunicação HTTP é orquestrada do lado do cliente, o que requer interação do utilizador para receber novos conteúdos ou a realização de *polling* periódico, o que pode resultar em grande desperdício de tráfego e carga desnecessária sobre o servidor, quando o servidor não tem nova informação para devolver (ver Figura 24).

Atualmente existem soluções que permitem que seja o servidor a enviar informação para o cliente por iniciativa do servidor. No entanto, estas soluções são baseadas em técnicas de *long-polling* (3), o que pressupõe pedidos HTTP de longa duração, realizados periodicamente, em que o servidor pode ir enviando informação para o cliente (ver Figura 25).

No entanto, todas estas técnicas têm os mesmos problemas: a ligação cliente-servidor não é bidirecional e é baseada em HTTP, o que significa que se o cliente quiser enviar informação para o servidor terá que realizar outros pedidos HTTP em simultâneo e todas as mensagens trocadas entre cliente/servidor têm o peso dos cabeçalhos do protocolo HTTP, resultando em tráfego desnecessário.

Para além disto, existem problemas de segurança como *cross-domain*, o que faz com que alguns *Web browsers* não permitam realizar pedidos HTTP a servidores que não pertençam ao domínio de onde a página *Web* foi carregada, mesmo que o servidor esteja configurado para aceitar pedidos HTTP vindos de outros domínios.

Com o surgimento do HTML5, foi introduzido o protocolo *WebSocket*[1], que permite resolver os problemas aqui descritos. O protocolo *WebSocket* permite criar uma ligação persistente entre um *Web browser* e um servidor. Desta forma, em qualquer altura o servidor poderá enviar mensagens para o cliente e vice-versa.

Para estabelecer uma ligação com *WebSockets* o cliente envia um pedido de *handshake*, ao qual o servidor responde (ver Listagem 17 e Listagem 18). A partir deste momento, a ligação é estabelecida e pode-se enviar mensagens em ambas as direções (ver Figura 26).

Note-se que, enquanto uma mensagem transmitida por HTTP pode levar no pedido imensos cabeçalhos e *cookies*, uma mensagem transmitida por *WebSocket* tem apenas 2 *bytes* de *overhead*, que delimitam o início e fim de mensagem.

[1] Embora esteja bastante estável e funcional, o protocolo *WebSocket* ainda não está concluído e pode sofrer alterações ao longo do tempo

Por estes motivos, algumas das tecnologias sobre as quais assenta a plataforma desenvolvida neste trabalho foram escolhidas não só por questões de performance e versatilidade mas também pelo suporte que trazem na utilização do protocolo *WebSocket*.

2.1 REQUISITOS

Com base nos objetivos gerais que foram propostos para este trabalho, segue um conjunto de requisitos funcionais e não funcionais que a plataforma deverá suportar:

2.1.1 REQUISITOS FUNCIONAIS

- O sistema deve permitir a criação de contas de utilizador e aceitar apenas ligações de clientes fidedignos. Tipicamente, cada aplicação *Web* terá uma conta de utilizador no sistema, à qual deverá estar associada uma chave de acesso ao mesmo e com a qual os seus clientes podem realizar ligações ao sistema.
- O sistema deverá permitir ligações através do protocolo *WebSocket* através dos portos 80 e 443, sendo este último usado para ligações SSL.
- O sistema deverá ter um mecanismo de autenticação e autorização de forma a garantir que os clientes de uma aplicação apenas podem aceder à informação que está acessível a essa aplicação e/ou acessível às credenciais do cliente em particular. Deverá ser possível criar perfis de autorização para diferentes tipos de clientes dentro de uma aplicação. Por exemplo, um *Web site* tipicamente tem utilizadores anónimos ou autenticados e restringe o acesso a determinada informação a certos grupos de utilizadores. Deverá ser possível atribuir permissões de escrita e/ou leitura a determinada informação aos perfis definidos numa conta de utilizador.
- O sistema deverá permitir que uma aplicação possa partilhar conjuntos de informação a clientes de outras aplicações, mediante configuração, de forma a suportar cenários de integração de sistemas.
- Sempre que um cliente estabelece uma ligação ao sistema, deverá poder realizar operações de leitura e/ou escrita (publicar e/ou subscrever) sobre os vários tipos de informação acessíveis a esse cliente.
- Deverá ser possível tipificar a informação que é disponibilizada no sistema, de forma a poder realizar operações *publish/subscribe*, isto é, os clientes subscrevem determinados tipos de informação que podem ser publicados por outros clientes.
- Deverá ser possível especificar filtros nas operações de subscrição sobre informação que seja publicada no formato *JSON*. Estes filtros deverão ser avaliados no servidor e caso as condições do filtro não sejam satisfeitas, então essa informação não deverá ser enviada para o cliente que especificou os filtros. Por exemplo, um cliente de uma aplicação (ou serviço) que publica informação de meteorologia de todas as cidades de

Portugal pode querer receber apenas informação quando a cidade é Lisboa e a temperatura é inferior a 15ºC. Outro cliente pode querer receber toda a informação que seja publicada. Desta forma, garante-se que um cliente apenas recebe a informação que realmente lhe interessa, evitando tráfego desnecessário.

- Sempre que determinada informação é publicada, esta deverá ser enviada, assim que possível, para todos os subscritores que tenham interesse nesse tipo de informação.

2.1.2 REQUISITOS NÃO FUNCIONAIS

- A arquitetura do sistema deverá permitir escalabilidade horizontal, para que não haja limite de ligações em simultâneo ao sistema, e para que a performance do sistema não seja degradada pelo número de ligações existentes num dado momento.
- O sistema deverá ser extensível, modular, e desenvolvido seguindo boas práticas de desacoplamento entre componentes para que os mesmos possam ser testados separadamente.

2.2 TECNOLOGIAS PROPOSTAS

As tecnologias propostas para a concretização deste trabalho foram escolhidas de modo a que o sistema seja *cross-platform* e a API cliente para *HTML5/Javascript* fosse *cross-browser*.

O sistema deverá poder ser alojado em ambientes Windows e/ou Linux, e todas as tecnologias utilizadas deverão ter um tipo de licenciamento que permita a redistribuição do seu código fonte juntamente com o código fonte da plataforma proposta.

3 SOLUÇÃO

No presente capítulo é descrita a solução proposta para atingir os objetivos do trabalho.

3.1 DESCRIÇÃO GERAL

De um modo geral, o sistema funciona como sendo um *endpoint* ao qual um grupo de clientes se pode ligar e aceder a conjuntos de informação que sejam do seu interesse, quer seja para realizar operações de leitura (*subscribe*) ou escrita (*publish*). Entende-se por cliente uma conexão ao sistema, em nome de uma conta de utilizador (*account*). Entende-se por *subscriber* um cliente que faz *subscribe* de informação e *publisher* um cliente que faz *publish* de informação.

A ligação ao *endpoint* do sistema é feita através de *WebSockets*, caso se trate de clientes em *web browsers* com suporte para *WebSockets*, ou, caso o *browser* não suporte *WebSockets*, outro mecanismo de *fallback* que o sistema possa vir a disponibilizar, não fazendo parte dos objetivos deste trabalho a criação destes mecanismos.

Os *subscribers* recebem informação por iniciativa do sistema, e não a pedido, isto é, o sistema faz *push* de informação para os interessados sem que estes tenham de estar constantemente a fazer *polling*. Este requisito apenas é garantido quando a comunicação é baseada em *WebSockets*. Outros mecanismos de *fallback* podem envolver operações de *polling*.

A informação enviada de e para o sistema é, de agora em diante, denominada de mensagem, e feita através de canais lógicos, aos quais chamamos de *streams*.

Uma *stream* é um canal lógico por onde passam mensagens num determinado formato. Se um *subscriber* especificar filtros ao subscrever numa *stream*, apenas recebe as mensagens que cumpram as regras especificadas nesses filtros. Do ponto de vista do sistema, uma *stream* não é mais do que um nome semântico associado ao envio/receção de mensagens e que serve para agrupar *subscribers* e multiplexar mensagens.

Um *subscriber* não conhece os *publishers*, nem um *publisher* conhece os *subscribers*. A ligação *publisher -> subscriber* é realizada através de *streams*.

Para além das *accounts* que podem ser registadas no sistema, o sistema permite que sejam registados serviços, cujo objetivo é a disponibilização de *streams* com informação que possa ser útil para quem está a desenvolver aplicações dinâmicas e que tipicamente recorre a *Web Services*.

Um serviço não é mais do que uma *account* cujas *streams* são públicas e que pode ser associado a contas de utilizador. Por omissão, apenas os clientes que efetuam ligações em nome do serviço podem escrever nas *streams* do mesmo, salvo configuração em contrário.

Na Figura 1 está representada, de forma geral, a interação entre clientes do sistema, entidades do sistema e respetivas *streams*.

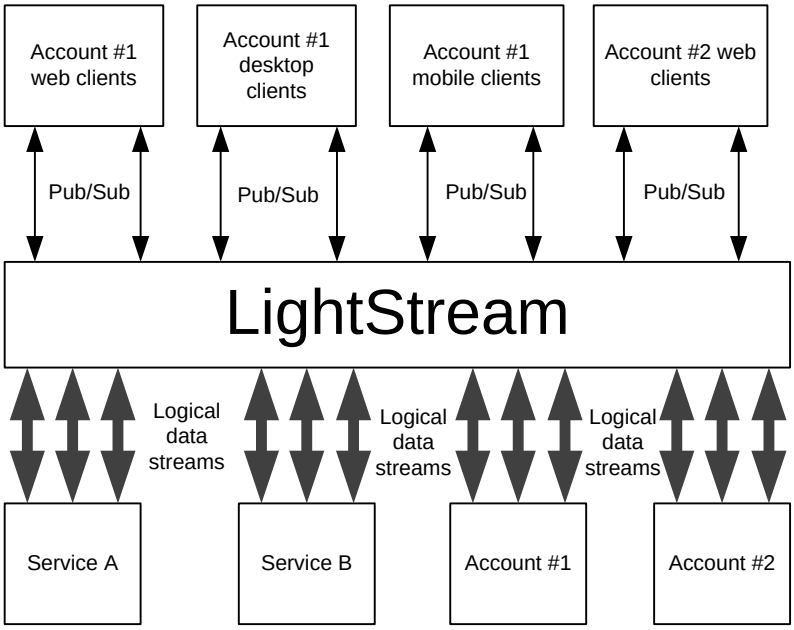

FIGURA 1 – VISÃO GERAL DO SISTEMA

O sistema conta com um modelo de segurança para garantir autenticação dos clientes e autorização no acesso às *streams*. Este modelo é descrito no tópico 3.3.

Para que qualquer cliente se possa ligar ao sistema, e aceder a uma ou mais *streams*, deverá autenticar-se e ter permissões

3.2 ARQUITETURA

A arquitetura proposta para o sistema foi pensada tendo em conta a escalabilidade e performance do sistema, para evitar possíveis *bottlenecks*. A escolha das tecnologias e a separação das várias camadas do sistema permite que os seus componentes possam ser testados isoladamente, bem como implementados noutras tecnologias caso seja necessário.

Na Figura 2 está representada a arquitetura geral do sistema, onde é possível visualizar as várias camadas que o compõem.

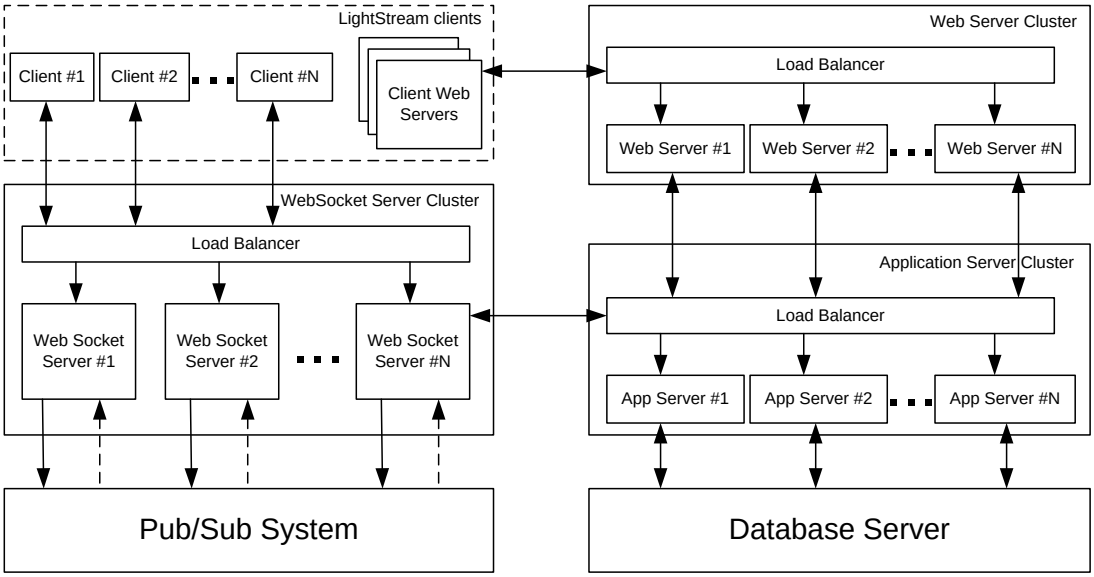

FIGURA 2 - ARQUITETURA GERAL DO SISTEMA

O sistema está dividido em duas grandes camadas: *frontend* e *backend*. A camada *frontend* é composta pelos servidores *WebSocket Server* e *Web Server* e pode ser acedida por qualquer cliente externo ao sistema, enquanto a camada *backend* é composta pelos servidores aplicacionais e só pode ser acedida por componentes internos do sistema.

A camada *WebSocket Server Cluster* representa o conjunto de servidores *Web* ao qual é possível estabelecer ligações ao sistema utilizando a API cliente, quer seja através do protocolo *WebSocket* ou outro mecanismo de *fallback*, como *long-polling*. Esta camada tem como responsabilidade a persistência das ligações dos clientes e realizar o envio/receção de mensagens. Uma vez que cada ligação ao sistema resulta numa conexão TCP aberta e ocupa recursos do sistema operativo, será necessário escalar horizontalmente, distribuindo as várias ligações entre os diversos servidores disponíveis nesta camada.

Surge assim a necessidade de ter um balanceador de carga nesta camada. Este balanceador deve suportar o conceito de sessões e afinidades a um servidor, para que todos os pedidos oriundos de ligações que não sejam efetuadas através do protocolo *WebSocket* sejam sempre dirigidos ao servidor onde foi criada a sessão.

A camada *Pub/Sub System* tem como responsabilidade a receção de todas as mensagens publicadas nas *streams* do sistema e a distribuição (*multicasting*) dessas mesmas mensagens para todos os *WebSocket Servers* onde existam ligações de clientes interessados em receber mensagens dessas mesmas *streams*.

Na camada *Application Server Cluster* encontra-se toda a lógica aplicacional do sistema exposta através de serviços. Estes serviços estão acessíveis às camadas *WebSocket Server*

Cluster e *Web Server Cluster*. Esta camada conta com um balanceador de carga, de modo a que seja possível escalar horizontalmente os servidores desta camada, sem comprometer a performance na execução dos serviços.

Na camada *Web Server Cluster* encontram-se expostos alguns serviços da camada aplicacional através do protocolo *HTTP*, recorrendo a interfaces *REST* (4). Embora o processamento dos serviços expostos nesta camada seja realizado pela camada aplicacional, esta camada conta também com um balanceador de carga, para evitar um possível *bottleneck*, num cenário em que exista um grande número de clientes a utilizar a plataforma e a chamar serviços.

Na camada *Database Server* encontra-se a base de dados do sistema, onde estão configuradas as *accounts*, bem como outros tipos.

Toda a comunicação entre os componentes do sistema, é realizada através de *sockets TCP* e deve ser realizada de forma assíncrona, ou seja, quando um componente chama um serviço não deve ficar bloqueado à espera da resposta, mas sim continuar disponível para realizar trabalho. Quando a resposta do serviço chegar, o componente poderá retomar o fluxo de execução que estava à espera da resposta do serviço.

3.3 MODELO DE SEGURANÇA

O modelo de segurança do sistema é baseado em *tokens*, *roles* e *passwords*.

O sistema apenas aceita ligações de clientes fidedignos. Entende-se por cliente fidedigno um cliente que se liga ao sistema em nome de uma determinada *account* e que esteja autorizado para o fazer.

As ligações ao sistema são sempre efetuadas passando um *token* válido.

Cada *account* possui uma chave de acesso, uma *password* e uma chave para assinaturas digitais. A chave de acesso é pública, a *password* e a chave para assinaturas deverão ser privadas.

Em cada *account* pode ser configurado um conjunto de *roles*, que são grupos de permissões sobre as *streams que* os clientes dessa *account* irão ter. Uma *account* pode definir permissões de leitura e/ou escrita sobre as suas *streams*, bem como permissões de leitura sobre *streams* de serviços.

O par chave de acesso/*password* apenas é utilizado para pedir ao sistema para gerar um *token* de acesso com um determinado conjunto de *roles* associado. Este *token* será válido durante um período de tempo limitado. Findo esse tempo, o *token* é automaticamente invalidado.

O pedido de criação de *tokens* apenas deve ser realizado onde não haja perigo em expor a *password* da *account*, tipicamente em componentes de *backend* e nunca em aplicações cliente. O sistema deverá disponibilizar serviços com *SSL* para o pedido de geração de *tokens*.

O sistema conta ainda com dois sistemas de protecção adicionais, com base no IP do cliente para o qual o *token* se destina, e com base na utilização de um outro tipo de *token* específico para aplicações *web*, o qual chamamos de *webtoken*.

Na Figura 28 encontra-se representado o modelo de dados que dá suporte ao modelo de segurança.

No sistema de autenticação com base em IP, este é passado ao serviço de geração de *tokens* e o sistema apenas aceitará uma ligação com aquele *token* caso o IP do cliente coincida com o IP para o qual o *token* foi criado.

No sistema de autenticação com base em *webtokens*, o *token* criado é usado para estabelecer uma ligação ao sistema mas não está autorizado a realizar quaisquer operações sobre *streams*. Neste caso, a API cliente disponibiliza um mecanismo para que o cliente possa confirmar que de facto o utilizador da aplicação *Web* que está a tentar estabelecer uma ligação ao sistema é o utilizador para o qual o *token* foi gerado. Para realizar esta confirmação o sistema pede ao cliente que realize uma assinatura digital com a sua chave sobre um conjunto de dados que inclui a chave de acesso da *account*, o *token* de segurança e um identificador da ligação no servidor de *WebSockets*. Esta assinatura é enviada para o servidor de *WebSockets* e posteriormente validada na camada de lógica aplicacional, uma vez que o sistema também consegue produzir a mesma assinatura, pois tem acesso à chave de assinaturas digitais da *account*.

Note-se que a confirmação de *webtokens* é da responsabilidade da aplicação *Web*, e esta deverá recorrer a quaisquer mecanismos de autenticação que esteja a utilizar no seu *site* para poder validar a associação entre um utilizador do seu *site* e o *token* que pediu ao sistema para esse utilizador. Este mecanismo de segurança serve para que uma aplicação *Web* possa restringir de forma segura o acesso a determinadas *streams* apenas a determinados utilizadores, uma vez que se alguém conseguir aceder de forma indevida a um *token* que tenha sido gerado com permissões sobre essas *streams,* não conseguirá realizar uma ligação válida ao sistema pois o mecanismo de confirmação com *webtokens* deverá falhar quando o *Web site* detetar que o utilizador não está autenticado no seu site ou não está associado a esse *token*.

O sistema permite que sejam utilizados vários sistemas de autenticação ao mesmo tempo para uma ligação, a fim de tornar as ligações e o acesso à informação que passa nas *streams* usadas pela *account* o mais segura possível.

Após estabelecida uma ligação autorizada, o cliente poderá realizar operações de escrita e leitura nas *streams*, de acordo com as permissões configuradas nos *roles* associados ao *token*.

No capítulo Implementação é descrito em pormenor como foi implementado todo este modelo de segurança.

3.4 TECNOLOGIAS

Nesta secção é realizada uma primeira abordagem às tecnologias escolhidas para a implementação das diversas camadas que compõem o sistema. No capítulo Implementação, encontram-se os principais detalhes de implementação, problemas e desafios encontrados na utilização destas tecnologias.

As tecnologias escolhidas para a concretização deste trabalho foram escolhidas de modo a cumprir os requisitos propostos. Algumas destas tecnologias são *open source* e bastante populares, existindo imensa documentação e grupos de discussão acerca das suas funcionalidades, o que facilitou a aprendizagem das mesmas.

3.4.1.1 WebSocket Server

Como tecnologia de servidor web é proposta a utilização de *NodeJS* (5). Esta tecnologia permite criar servidores web tirando partido do modelo de programação assíncrono, não bloqueante e *event-driven* de *NodeJS*.

NodeJS tem suporte para *WebSockets*, mas já existem bibliotecas que fornecem uma abstração sobre uma ligação persistente ao servidor, quer seja através do protocolo *WebSocket* ou outros mecanismos de *fallback*, como *long-polling*. Tipicamente estas bibliotecas especificam um protocolo próprio e fornecem uma versão para servidor e outra para cliente onde implementam este protocolo. Foram estudadas as bibliotecas *SockJS* (6) e *Socket.IO* (7), sendo a última a mais usada e documentada. No entanto, *SockJS* tem uma API muito mais simples do que *Socket.IO*, uma vez que o seu principal objetivo é garantir um transporte eficiente de mensagens, de forma transparente para o programador, utilizando a estratégia de ligação que mais se adeque ao *web browser* do cliente, sendo o protocolo *WebSocket* a estratégia preferida. No caso de *Socket.IO*, para além da preocupação com a camada de transporte, esta biblioteca tem uma API muito orientada para o padrão *publish/subscribe*, o que poderia limitar a API cliente disponibilizada neste trabalho.

Desta forma, a melhor abordagem será a plataforma utilizar uma estratégia à base de *providers*, criando uma abstração sobre a biblioteca que trata da componente de transporte, podendo ser usada *SockJS*, *Socket.IO* ou outra qualquer. No âmbito deste trabalho, apenas será implementado o *provider* para *SockJS*.

Como tecnologia para balanceamento de carga será utilizado o *HAProxy* (8), uma vez que é o balanceador aconselhado na documentação do *SockJS* e para o qual existem exemplos e tutoriais. Este balanceador suporta o afinidade de sessões a servidores, através da análise da *query path* num *URL* e respetivo mapeamento para um servidor em particular.

3.4.1.2 Publish/Subscribe Server

A tecnologia escolhida para esta camada foi a base de dados *Redis* (9). *Redis* é uma base de dados *NoSQL* (10) de registos par chave/valor, em memória, com a capacidade de persistir os dados em disco. O que torna esta tecnologia apelativa para este trabalho é a sua capacidade de realizar operações de *publish*/*subscribe* (11) de mensagens categorizadas em tópicos, onde os *publishers* colocam mensagens com um tópico associado na base de dados, e o *Redis* encarrega-se de fazer o *multicast* para todos os *subscribers* interessados em receber mensagens desse tópico. Esta abordagem permite escalar o *multicasting* de mensagens, independentemente do número de *publishers* e *subscribers*.

Redis é *cross-platform* e pode ser utilizado por um vasto conjunto de linguagens (12), entre as quais *.NET* e *NodeJS*.

3.4.1.3 Application Server

A tecnologia escolhida para a implementação da lógica aplicacional é *.NET* 4.0, de forma a tirar partido da *Task Parallel Library* (13).

As funcionalidades desta camada são disponibilizadas através de serviços. Quando se pretende disponibilizar serviços em *.NET*, tipicamente recorre-se a *WCF*. No entanto, quem irá consumir estes serviços será não só a camada *Web Server*, mas também a camada *WebSocket Server*. Surge assim a necessidade de disponibilizar estes serviços com *bindings* que sejam compreendidos pelas várias tecnologias abrangidas nestas camadas. De forma a otimizar a performance do sistema, a comunicação deve ser feita através de *bindings TCP* e não *HTTP*. Uma vez que *WCF* apenas suporta, de raiz, comunicação por *TCP* entre clientes *.NET*, não cumpre os requisitos de interoperabilidade entre tecnologias que o sistema requer.

A tecnologia proposta para a implementação da comunicação entre as diversas camadas do sistema é *ZeroMQ* (14). *ZeroMQ* é uma camada de transporte baseada em *sockets*, *cross-platform*, *open source* e pode ser utilizada por um vasto conjunto de tecnologias (15), entre as quais *.NET* e *NodeJS*.

ZeroMQ permite tirar partido de diversos padrões de troca de mensagens entre sistemas, tais como *request/reply*, *task distribution*, *publish/subscribe*, entre outros, para criar soluções distribuídas e totalmente assíncronas.

3.4.1.4 Web Server

A tecnologia escolhida para esta camada foi também *NodeJS*, uma vez que reutiliza alguns componentes desenvolvidos na camada *Web Socket Server*, como por exemplo o componente de comunicação entre *NodeJS* e *.NET*.

Para o balanceamento de carga entre os vários servidores desta camada será utilizado *HAProxy*.

3.4.1.5 Database Server

Para a persistência de dados foi escolhida a base de dados *MongoDB* (16). *MongoDB* é uma base de dados *NoSQL*, *cross-platform*, *open source* e pode ser usada por um vasto conjunto de linguagens, entre as quais *.NET*.

Nesta base de dados é persistida toda a informação sobre *accounts* e *tokens*.

3.5 APIs e *Frameworks*

O sistema disponibiliza duas APIs cliente (*Javascript* e *C#*) para realizar ligações persistentes ao sistema e um endpoint HTTP/HTTPS para geração de *tokens* de segurança. A API de *C#* realiza ligações *TCP* ao servidor de *websockets* e foi desenvolvida e estruturada da mesma forma que a API para *Javascript*. No tópico 4.1 é detalhado o funcionamento desta API, bem como alguns detalhes de implementação.

Para realizar a comunicação entre as camadas de *frontend* e *backend* foi desenvolvida uma *Framework* de comunicação assente na tecnologia *ZeroMQ*, totalmente assíncrona, e que pode ser reutilizada noutros projetos. No capítulo 4.7 é detalhado o funcionamento desta *Framework*, bem como os principais detalhes de implementação e explicado de que forma pode ser reutilizada noutros projetos.

4 IMPLEMENTAÇÃO

Neste capítulo são apresentados os detalhes de implementação dos vários componentes do sistema.

O código fonte deste trabalho foi desenvolvido no *Visual Studio 2010*. Na Figura 3 é apresentada a estrutura da solução.

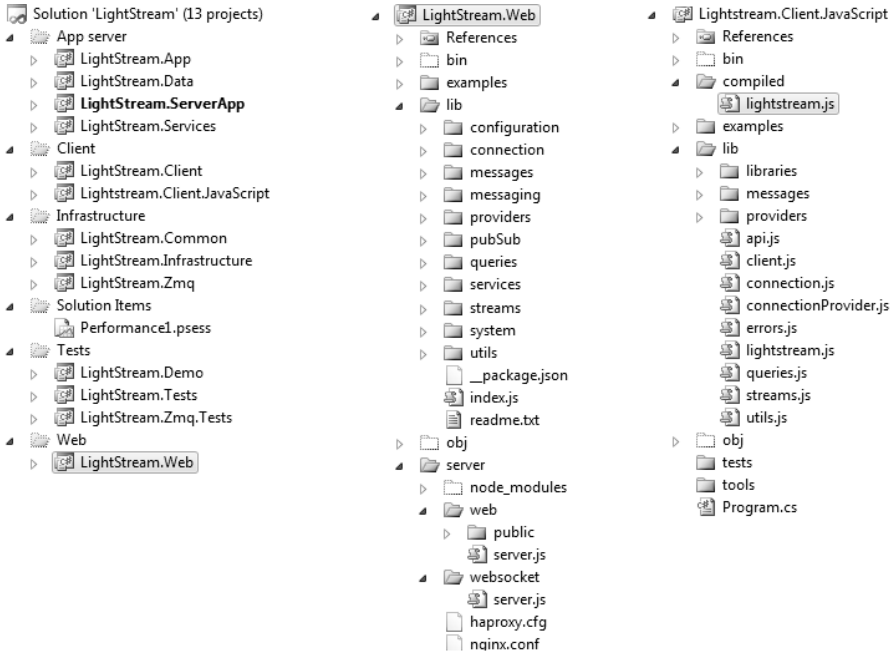

FIGURA 3 - ESTRUTURA DA SOLUÇÃO

A solução está dividida em pastas, que representam as várias camadas do sistema. A camada *backend* é composta pelos projetos das pastas *AppServer* e *Infrastructure*. A camada *frontend* é composta pelos projetos da pasta *Web* e *Client*. Na pasta *Tests* encontram-se testes efetuados aos componentes do sistema e um projeto de demonstrações (*Demo*) das funcionalidades do mesmo.

Os projetos *LightStream.Client.Javascript* e *LightStream.Web* têm *pre* e *post-build events*. No caso do projeto *LightStream.Client.Javascript*, o código fonte foi organizado em ficheiros *.js* separados e quando o projeto é compilado é gerado o ficheiro *compiled/lightstream.js* que resulta na agregação de todos os ficheiros *.js*. De seguida, este ficheiro é copiado para a pasta *server/web/public* do projeto *LightStream.Web*, onde são alojados os ficheiros de conteúdo estático pela camada *Web Server*. Na pasta *exemples* do projeto *LightStream.Client.Javascript* está também uma página *HTML* onde se podem ver as funcionalidades da *API* para *Javascript*.

No caso do projeto *LightStream.Web*, como os servidores *Web* estão em *NodeJS* e executam em *Linux*, o projeto copia o conteúdo das pastas *lib* e *server* para uma directoria partilhada entre a máquina de desenvolvimento (*Windows*) e a máquina onde correm os servidores *Web* (*Linux*), automatizando o processo de *deployment*. No caso da pasta *lib*, o conteúdo é copiado para a directoria *node_modules*, uma vez que o código de servidor *Web* da plataforma *LightStream* é exportado como um módulo de *NodeJS*.

4.1 API CLIENTE

No âmbito deste trabalho, foi desenvolvida uma API para *Javascript* e para *.NET*. Neste tópico é explicado como foram implementadas as funcionalidades da API cliente.

4.1.1 ARQUITETURA

Na Figura 4 é apresentada a arquitetura geral da API.

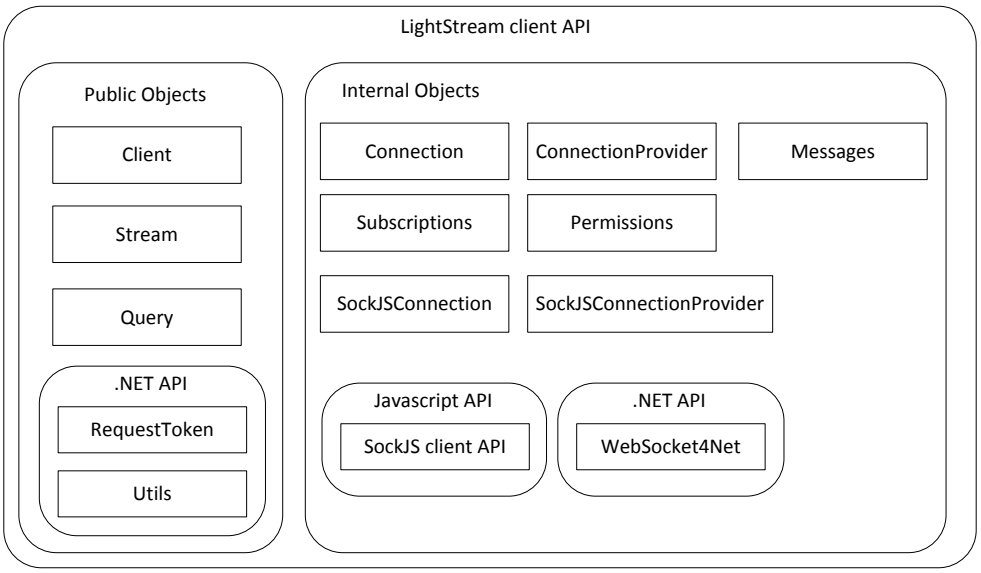

FIGURA 4 - ARQUITETURA LÓGICA DA API CLIENTE

As funcionalidades da API são implementadas pelos objetos *Client*, *Stream* e *Query*. Os restantes componentes são internos à API.

O diagrama UML de classes na Figura 5 serve de referência na explicação das funcionalidades ao longo deste tópico.

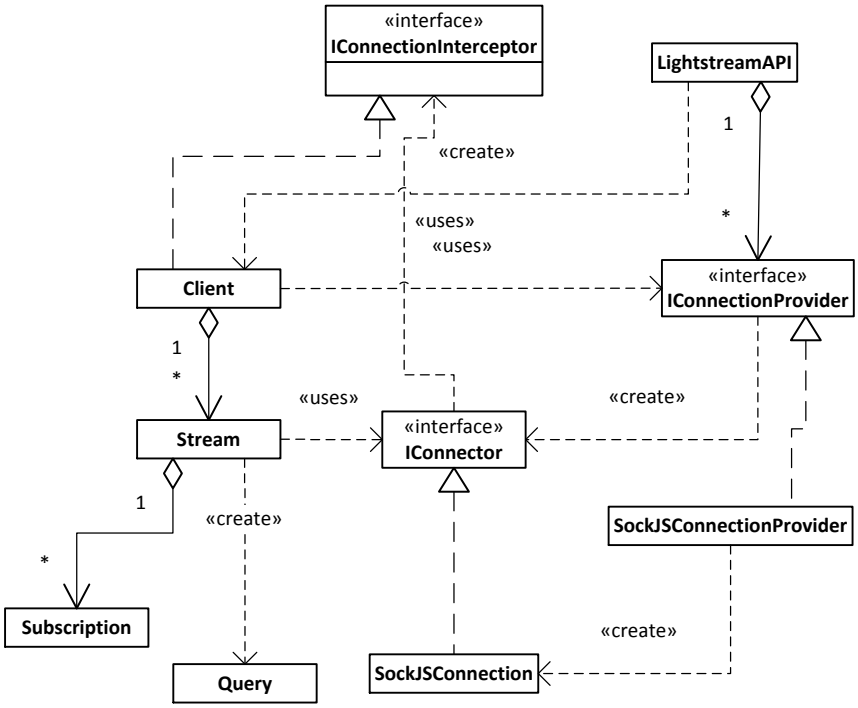

FIGURA 5 - DIAGRAMA UML DE CLASSES DA API CLIENTE

O código fonte das duas APIs encontra-se nos projetos *LightStream.Client* (*.NET*) e *LightStream.Client.Javascript*, sendo que no caso da API de *.NET* existe também uma referência para o projeto *LightStream.Common*.

4.1.2 CRIAÇÃO DE CLIENTES

Através de um objeto do tipo *Client* é possível criar uma ligação ao sistema, fechar a ligação caso esteja aberta, e obter uma referência para uma *Stream* caso o cliente tenha permissões para o fazer.

Para criar um objeto *Client* é necessário fornecer um objeto com a configuração que este irá utilizar para se ligar ao sistema. Na criação de um *Client*, é obtido um objeto que implemente a interface *IConnection* e que representa uma ligação persistente ao servidor com a qual se pode enviar dados para o sistema. A obtenção de um objeto I*Connection* é feita através do objeto *IConnectionProvider* que está associado ao *provider* que foi especificado na criação do *Client*.

O método *CreateConnection* da interface *IConnectionProvider* recebe uma implementação de *IConnectionInterceptor* que, neste caso, é implementada pelo objeto *Client*. Isto permite desacoplar as funcionalidades que a API fornece e o conceito de ligação ao servidor. A interface *IConnectionInterceptor* permite ao objeto *Client* saber quando uma ligação é aberta, fechada e quando recebe dados do servidor.

O uso de *providers* permite também que possam ser utilizados outros mecanismos de ligação ao sistema. Do lado do servidor existe também o conceito de *providers* de ligação, de forma a abstrair também os seus componentes do tipo de ligação que está a ser utilizado. Este tema é abordado capítulo 4.2.2.

Quem utiliza a API não tem acesso às implementações de *IConnection* e apenas consegue enviar e receber dados do sistema através de *Streams*.

No caso da API de *.NET*, foi utilizada a biblioteca *WebSocket4Net* para a criação de ligações ao servidor. Esta biblioteca implementa o protocolo *WebSocket*, o que permite estabelecer ligações ao servidor de *WebSockets* com o provider *SockJS*, pois ambas respeitam a mesma versão do protocolo. No entanto, *SockJS* tem um protocolo próprio que é usado no envio de mensagens entre cliente e servidor e, como tal, foi necessário implementar este protocolo na classe *SockJSConnection* da API de *.NET*, de forma a garantir que as mensagens entregues ao *IConnectionInterceptor* seguem a estrutura esperada.

Na chamada ao método *Open* do objeto *Client* deve ser passado um *callback* que é invocado quando a ligação é efetuada com sucesso ou quando falha, e um *callback* que é invocado para que o cliente possa validar um *token* de autenticação caso se trate de um *webtoken*. Uma ligação só é efetuada com sucesso caso o servidor consiga autenticar o cliente. O processo de autenticação é detalhado no tópico 4.2.3, no entanto do ponto de vista do cliente consiste no envio de uma mensagem com os dados de autenticação (*appkey* e *token*) recebidos no objeto de configuração do cliente.

A obtenção de um *token* é feito invocando um *endpoint* HTTP que é disponibilizado pela camada *Web Server* e que tem o seguinte formato:

https://**[server]**/services/auth/generatetoken?publicKey=**{publicKey}**&privateKey=**{privateKey}**
&roles=**{roles}**&ttl=**{ttl}**&ipAddress=**{ipAddress}**&webToken=**{webToken}**

A seguinte tabela explica os parâmetros que podem ser utilizados.

publicKey	Chave de acesso da *account*, também designada de *appKey*
privateKey	Chave privada da *account*, também designada de *password*
roles	Conjunto de roles, separados por '|'. Ex: role1|role2|role3
ipAddress	Endereço de IP do cliente para o qual o *token* é destinado
ttl	Tempo de vida, em segundos, para expiração do *token*
webToken	Valor booleano, que indica se o *token* é um *webToken*

A API cliente de *.NET* fornece um método assíncrono (ver *LightStreamAPI.RequestToken*) que faz a invocação deste *endpoint* e facilita a obtenção de *tokens*. Note-se que este *endpoint*, por receber a chave privada da *account*, não deverá ser invocado onde haja perigo em expor esta chave. Na Listagem 21 é apresentado o código fonte do método *RequestToken* e na Listagem 22 é mostrado um exemplo da criação de um cliente onde é pedido um *webtoken* pelo método *RequestToken*.

4.1.3 MENSAGENS DE SISTEMA

Todas as mensagens de sistema têm um identificador associado (ver *MessageTypes*), e quando são enviadas quer pelo cliente quer pelo servidor são encapsuladas por uma mensagem comum que é referida como *MessageEnvelope*. Esta mensagem contém o identificador da mensagem original e conteúdo da mensagem. Desta forma, é possível determinar qual o tipo de mensagem que está a ser enviado sem ser necessário inspecionar o seu conteúdo e entregá-la a um *handler* apropriado. Esta técnica é usada tanto na API de *Javascript* como na de *.NET*, tal como é ilustrado na Listagem 19 e na Listagem 20. No caso da API de *Javascript* a técnica é semelhante, em que é criado um objeto cujas propriedades têm como nome o identificador da mensagem e como valor o *handler* que a sabe tratar.

O formato das mensagens é uma *string JSON*. Na API de *Javascript* são utilizadas as funções *JSON.stringify* e *JSON.parse*, e na API de *.NET* é utilizado o *DataContractJsonSerializer*, pelo que foi necessário anotar as classes com atributos *DataContract* e *DataMember*.

Na Figura 27 está representado o diagrama UML de classes onde se pode ver quais as mensagens que existem e qual o seu formato.

De seguida, descreve-se em que situações são utilizadas as mensagens.

AuthenticationRequestMessage – Enviada pelo cliente para o servidor com os dados de autenticação.

AuthenticationResponseMessage – Enviada pelo servidor para o cliente como resposta ao seu pedido de autenticação.

WebTokenRequestMessage – Enviada pelo servidor para o cliente como resposta ao seu pedido de autenticação, quando o *token* é um *webtoken*.

WebTokenResponseMessage – Enviada pelo cliente para o servidor com o conteúdo dos dados necessários para a validação do *webtoken*, assinado com a chave secreta da *account*.

DataMessage – Enviada pelo cliente para o servidor quando está a publicar uma mensagem ou pelo servidor para o cliente quando este recebe uma mensagem.

SubscribeRequestMessage – Enviada pelo cliente para o servidor quando pretende subscrever uma *Stream*.

SubscribeResponseMessage – Enviada pelo servidor para o cliente com a resposta ao pedido de subscrição.

UnsubscribeRequestMessage – Enviada pelo cliente para o servidor quando pretende cancelar uma subscrição numa *Stream*.

UnsubscribeResponseMessage – Enviada pelo servidor para o cliente com a confirmação de que a sua subscrição foi cancelada.

4.1.4 PUBLISH / SUBSCRIBE

Para realizar operações de *Publish* e *Subscribe*, o cliente deve aceder a uma *Stream*. Para isto, é necessário possuir as devidas permissões. Estas permissões são obtidas durante o processo de autenticação e utilizadas, tanto no servidor, como cliente, para validar os acessos às *Streams*. Caso o cliente tente realizar uma destas operações sem ter as devidas permissões, é lançada uma exceção com essa indicação.

Na Listagem 23 e na Listagem 24 é apresentado o código da classe *Stream* da API de *.NET* como referência para a implementação das operações *Publish* e *Subscribe*. Note-se que do lado da API para *Javascript* foram utilizadas técnicas semelhantes, exceto nos cuidados com os acessos concorrentes, visto não existirem.

Na operação de *Publish*, é criada uma mensagem *DataMessage* com o conteúdo da mensagem e a identificação da *Stream*. De seguida esta mensagem é encapsulada num *MessageEnvelope* e enviada pelo objeto *IConnection*. O tratamento desta mensagem do lado do servidor é explicado no tópico 4.2.4.

Na operação de *Subscribe*, é criado um objeto *Subscription* com um identificador temporário e único por *Stream*, ao qual ficam associados o *callback onMessage* e o *callback onSubscribed*. O identificador temporário é obtido através da operação *Interlocked.Increment*, que garante o incremento atómico e thread-safe do campo *_nextRequestId*. O objeto *Subscription* é guardado no dicionário *_pendingSubscriptions* cuja chave é o identificador temporário. De seguida, é enviada a mensagem *SubscribeRequestMessage* para o servidor dando indicação de que este cliente pretende criar uma subscrição na *Stream*. Do lado do servidor, caso o cliente tenha permissões é também criado um objeto que representa uma subscrição, com um identificador único e final (ver tópico 4.2.4). O identificador da subscrição do lado do servidor é enviado para o cliente na mensagem *SubscribeResponseMessage*, juntamente com o identificador temporário, para que do lado do cliente se consiga relacionar o par pedido/resposta (ver o método *OnSubscribeResponse*).

Assim que a subscrição é confirmada, é chamado o *callback onSubscribed* e a partir deste momento, sempre que chegarem mensagens do servidor para esta subscrição o *callback onMessage* é chamado (ver o método *OnDataMessage*).

Note-se que o tipo de dicionário utilizado para gerir os objetos *Subscription* na classe *Stream* é o *ConcurrentDictionary*. Este tipo de colecção garante o acesso *thread-safe* e *atómico* ao dicionário, não sendo necessário recorrer a *locks* ou outros mecanismos de sincronismo por parte de quem o utiliza.

Na API de *.NET* os métodos *Publish* e *Subscribe* foram criados com suporte para tipos genéricos, possibilitando ao cliente a publicação de qualquer tipo de objeto e a subscrição de mensagens de forma tipificada, sem que este se tenha que preocupar com os detalhes de seriação, ficando a cargo da API a seriação dos objetos no formato *JSON*.

Um cliente pode criar várias subscrições numa *Stream*. Este cenário faz sentido se o cliente especificar *queries* no ato da subscrição.

4.1.5 QUERIES

A especificação de *queries* no ato da subscrição de uma *Stream* é útil em cenários em que o cliente apenas pretende receber mensagens quando o seu conteúdo seja do seu interesse.

A forma tradicional de fazer isto seria receber todas as mensagens e realizar a *query* do lado do cliente, o que poderia causar tráfego desnecessário. Na plataforma LightStream as *queries* são executadas no servidor, garantindo que o cliente apenas recebe os dados que lhe interessam.

Através de uma *Stream* é possível criar um objeto *Query* que pode ser usado durante uma subscrição para filtrar uma mensagem que seja apenas texto ou uma mensagem que seja um objeto no formato *JSON*. A classe *Query* expõe os seus operadores através de uma interface *fluent*, ou seja, é possível encadear vários operadores e no fim utilizar o objeto que resulta das várias chamadas encadeadas para criar uma subscrição. Ao subscrever uma *Stream* com uma *query* é um enviado para o servidor um objeto com informação dos operadores especificados na *query* (ver *StreamQuery* e *StreamQueryItem*), de forma que a *query* possa ser reconstruída e executada no servidor.

A API disponibilizada para a criação de *queries* segue as mesmas regras que a API *jLinq*, que é usada no servidor para a execução das *queries*. No entanto, não foram implementados todos os operadores disponíveis na API jLinq, mas apenas os mais relevantes para demonstrar o conceito. A classe *Query* foi implementada de forma que fosse possível, no futuro, acrescentar mais operadores.

Do lado do servidor, o objeto que representa a *query* foi implementado com o mesmo intuito, existindo um mapeamento entre o nome dos operadores e o nome de propriedades num objeto que contém as funções de cada operador a aplicar sobre a *query*. Embora *jLinq* seja uma API apenas utilizada em *Web Browsers*, foi relativamente fácil incluir o seu código fonte na solução e exportá-lo como sendo um módulo de *NodeJS*, podendo ser reutilizado pelos componentes do servidor de *WebSockets*.

Um objeto *query* pode ter dois tipos de avaliação: *Any* ou *Select*. Uma avaliação do tipo *Any* apenas valida se a mensagem obedece aos requisitos impostos pela *query*, e em caso afirmativo, a mensagem original é entregue ao cliente. Uma avaliação do tipo *Select* apenas pode ser realizada sobre uma propriedade do tipo *array* em objetos *JSON.* A *query* é executada contra os elementos do *array* e a mensagem que é entregue ao cliente será um objeto *JSON*, cujo *array* que foi usado para a *query* apenas contém os elementos que passaram nos filtros. Este tipo de avaliação permite ao cliente ter uma granulosidade ainda maior sobre os dados que recebe, uma vez que não só recebe mensagens que contenham dados do seu interesse, como também consegue alterar a mensagem original para receber os dados que de facto são úteis. No projeto *LightStream.Demo* encontram-se exemplos onde esta funcionalidade é utilizada.

4.2 WEB SOCKET SERVER

O servidor de *WebSockets* foi desenvolvido em *NodeJS*. No âmbito deste trabalho, e para efeitos de demonstração, este servidor é executado em ambiente *Linux*, mais concretamente *Ubuntu 12.04*.

4.2.1 ARQUITETURA

Na Figura 6 é apresentada a arquitetura lógica do servidor de *WebSockets*.

A Figura 7 contém o diagrama UML de classes onde se pode ver os vários componentes que compõem o servidor de *WebSockets*, e que serve de apoio ao longo deste tópico.

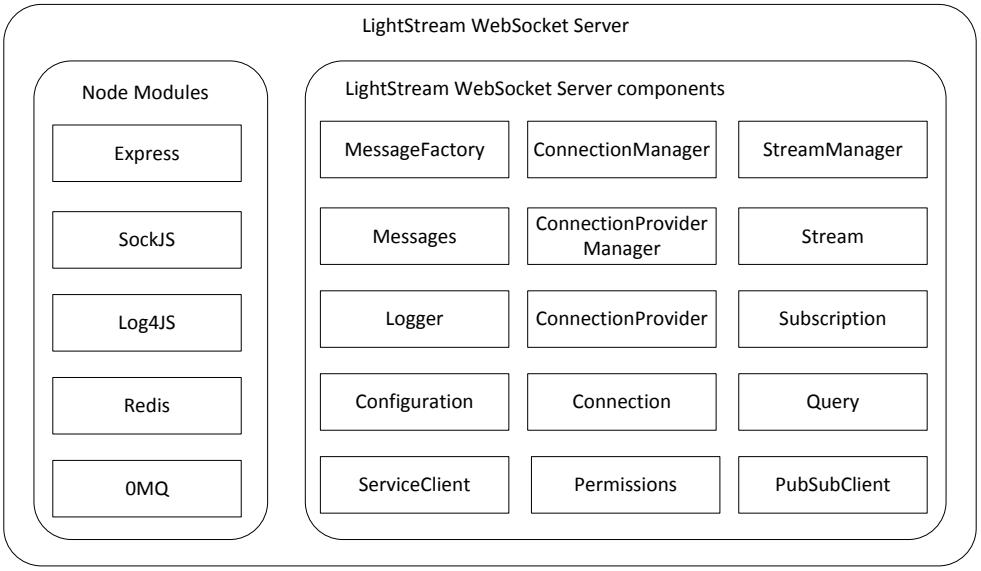

FIGURA 6 - ARQUITETURA LÓGICA DO SERVIDOR DE WEBSOCKETS

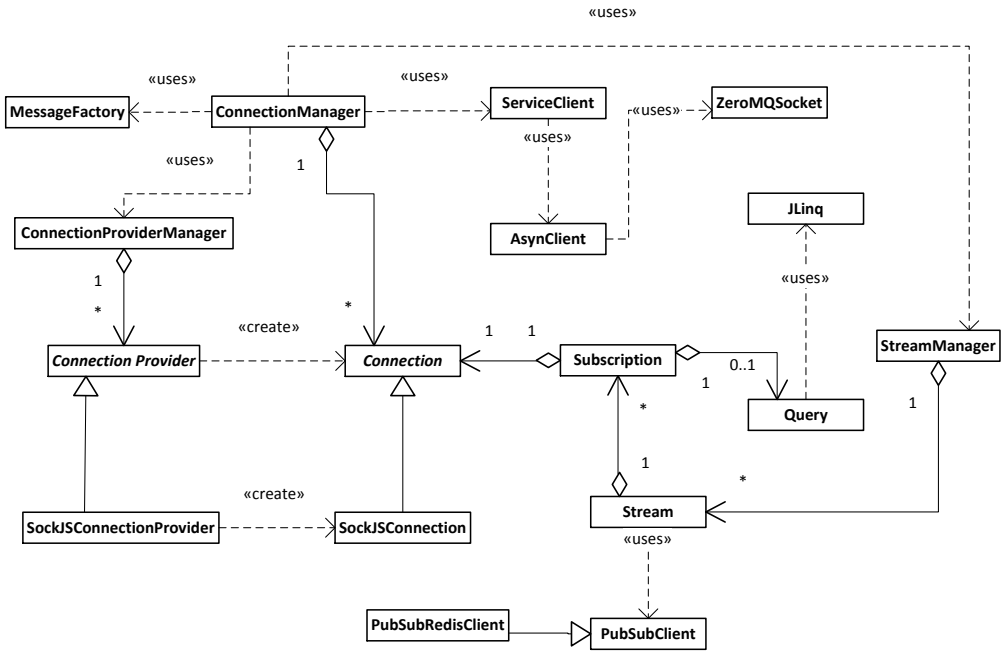

FIGURA 7 - DIAGRAMA UML DE CLASSES DO SERVIDOR DE WEBSOCKETS

Os principais componentes são o *ConnectionManager*, responsável por gerir as ligações, o *MessageFactory*, responsável por desseriar/seriar mensagens de sistema, o *StreamManager*, responsável por gerir as *Streams*, o *PubSubClient*, responsável por enviar e receber do sistema de *PubSub*, o *ServiceClient*, responsável pela comunicação com a camada de *backend*, e o *ConnectionProviderManager*, onde estão registados os vários *providers* de ligações que existem, e que são usados pelo *ConnectionManager* quando é criada uma nova ligação.

A plataforma *LightStream* funciona como um módulo do *NodeJS*, sendo o seu ficheiro principal o *index.js*, onde estão organizados, por namespaces, os vários objetos que compõem o sistema. O ficheiro executado pelo servidor de *Node* é o *websocket/server.js*, cujo código é apresentado na Listagem 25.

4.2.2 GESTÃO DE LIGAÇÕES

A gestão de ligações é feita pelo componente *ConnectionManager*. Este componente aceita ligações de qualquer *provider*, recorrendo ao *ConnectionProviderManager* para obter um *ConnectionProvider* que saiba criar objetos *Connection* do *provider* actual. A Figura 8 ilustra este processo.

A convenção utilizada foi que cada *provider* de ligação trata os pedidos HTTP realizados ao endereço /*lightstream*/websocket/**{providerName}**, e registar o respectivo *ConnectionProvider* com o nome {***providerName}*** no *ConnectionProviderManager*.

O componente *ConnectionManager* gere objetos *Connection*, pelo que os *providers* deverão instanciar objetos que derivem de *Connection* e implementar os métodos definidos no *prototype* deste. Desta forma a todos os restantes componentes do servidor de *WebSocktets* podem usar os objetos *Connection*.

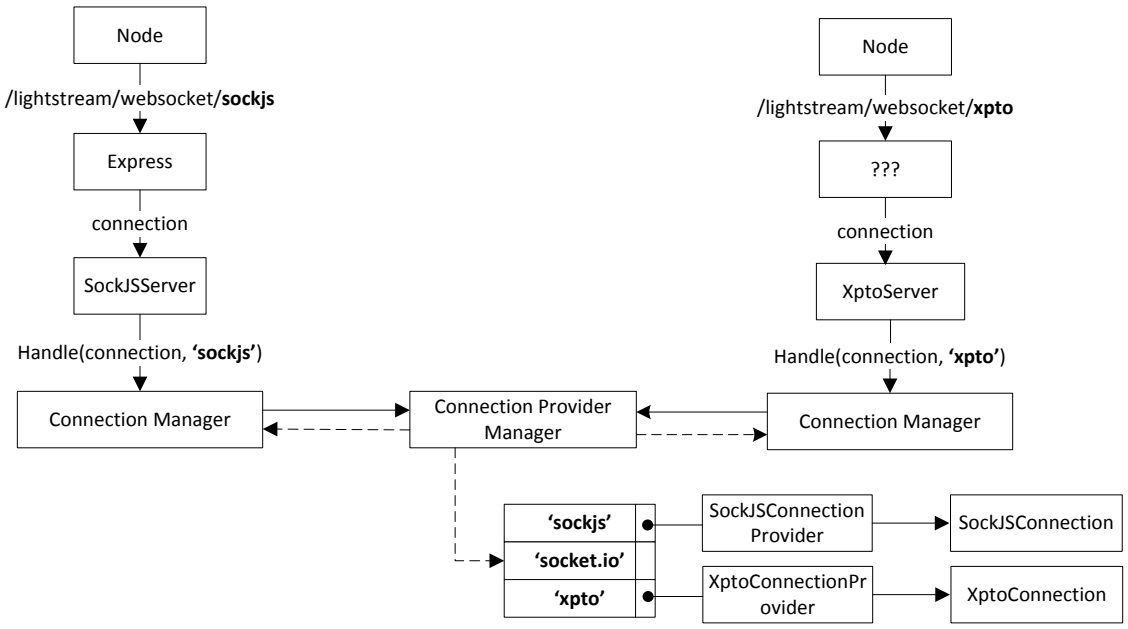

FIGURA 8 - CONNECTION PROVIDERS

Após receber uma ligação, o *ConnectionManager* coloca o objeto *Connection* num estado "pendente" até receber a mensagem *AuthenticationRequestMessage*, onde se inicia o processo de autenticação.

Enquanto o processo de autenticação não é concluído, a *Connection* não tem autorização para realizar qualquer tipo de operação de *publish/subscribe*, e caso o faça, será imediatamente fechada pelo servidor. O processo de autenticação é discutido no tópico 4.2.3.

Todas as mensagens enviadas pelos clientes são recebidas pelos objetos *Connection* e tratadas pelo *ConnectionManager*, através da API de eventos de *NodeJS*. É o *ConnectionManager* que, ao receber as mensagens *DataMessage* e *SubscribeRequest* verifica se o cliente tem permissões, e em caso afirmativo, delega o tratamento destas ações para o *StreamManager*. Caso o cliente não tenha permissões a ligação é imediatamente fechada, uma vez que está a ser realizado um acesso não autorizado.

4.2.3 SEGURANÇA

O processo de autenticação é desencadeado assim que uma ligação chega ao *ConnectionManager*.

O cliente envia a mensagem *AuthenticationRequestMessage*, que contém a *appKey* e o *token* de autenticação. O *ConnectionManager* chama a operação *ValidateToken* do serviço *IAuthService* do servidor aplicacional, através do objeto *ServiceClient*, a fim de obter um objeto com a informação do *token* (*TokenInfo*). Caso não obtenha resposta, assume-se que ou o *token* não existe ou já expirou, e então é enviada a mensagem *AuthenticationResponseMessage* com informação de erro. Caso o *token* tenha sido gerado para um endereço de *IP* específico, este é validado no *ConnectionManager*, uma vez que tem acesso ao endereço de *IP* do cliente. Caso o *token* seja válido, são guardadas as permissões obtidas pelo *TokenInfo* no objeto *Connection*, que serão usadas para verificar permissões nas ações de *Publish* e *Subscribe* efetuadas por essa *Connection*. De igual forma, as permissões são enviadas para o cliente na mensagem *AuthenticationResponseMessage*, para que este possa validar os acessos nas ações de *Publish* e *Subscribe*. A Figura 9 contém o diagrama de sequência onde está representada esta troca de mensagens.

Caso o *token* seja um *WebToken*, é enviada a mensagem *WebTokenRequestMessage*, juntamente com a *appkey*, o *token*, e o identificador da *Connection*, apenas conhecido no servidor. O cliente deverá produzir uma assinatura usando o algoritmo SHA-256 com a *secretKey* da sua *account* e enviá-la na mensagem *WebTokenResponseMessage*, cujo conteúdo será usado na chamada à operação *ValidateWebToken* do serviço *IAuthService*. A Figura 10 contém o diagrama de sequência onde está representada esta troca de mensagens. A Listagem 1 contém a implementação do método *Sign*, que pode ser utilizado por quem se está a ligar ao sistema usando a API cliente de *.NET*, ou em componentes de servidor do cliente final caso possam usar a *dll LightStream.Common*.

Em qualquer uma das situações, sempre que o processo de autenticação falha, a ligação é fechada pelo *ConnectionManager*, uma vez que está a ser realizado um acesso indevido ao sistema.

```
public static string Sign(string dataToSign, string secretKey)
{
    byte[] signedDataBytes;
    using (var hmac = new HMACSHA256(Encoding.ASCII.GetBytes(secretKey)))
    {
        signedDataBytes = hmac.ComputeHash(Encoding.ASCII.GetBytes(dataToSign));
    }

    var sb = new StringBuilder();
    for (var i = 0; i < signedDataBytes.Length; ++i)
    {
        sb.Append(signedDataBytes[i].ToString("x2"));
    }

    return sb.ToString();
}
```

LISTAGEM 1 - IMPLEMENTAÇÃO DO MÉTODO *SIGN*

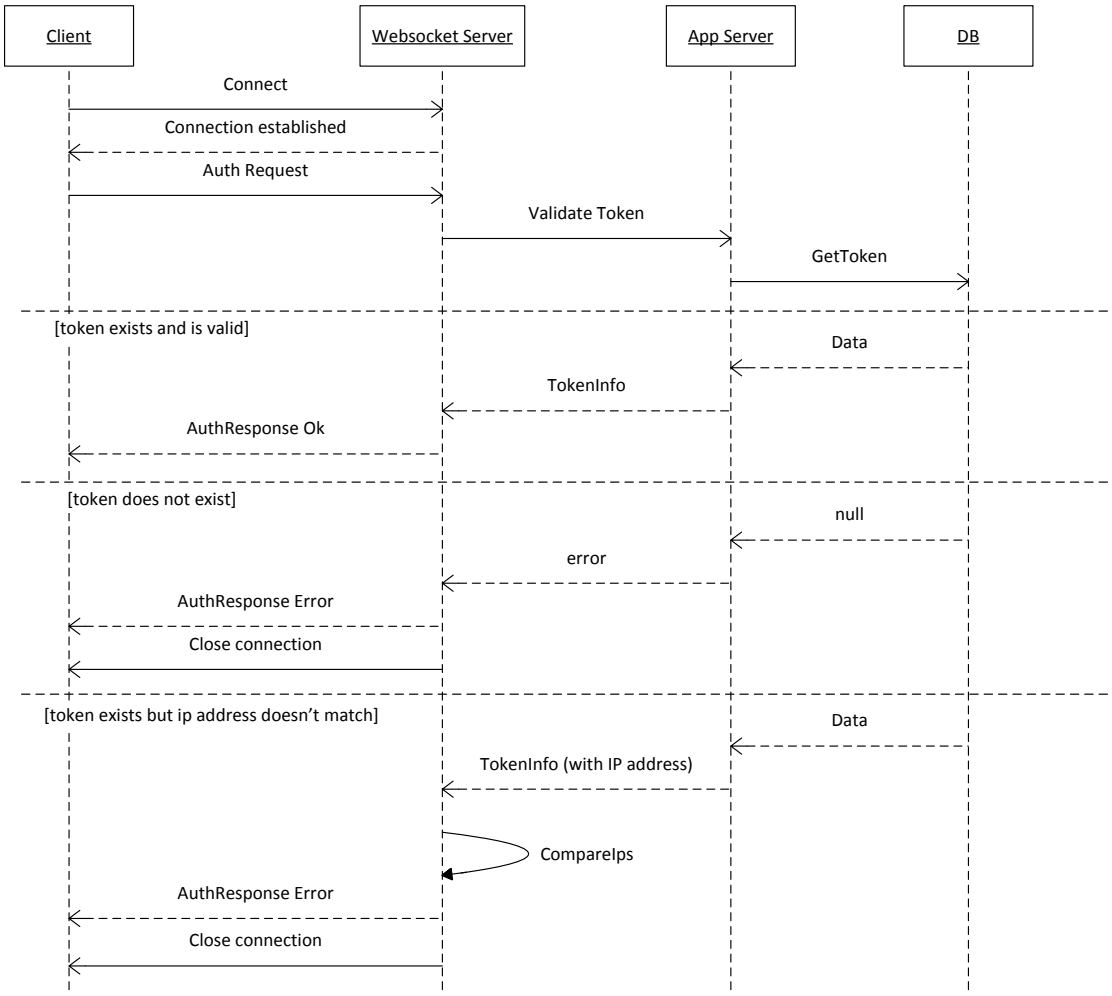

FIGURA 9 - DIAGRAMA DE SEQUÊNCIA DO PROCESSO DE AUTENTICAÇÃO

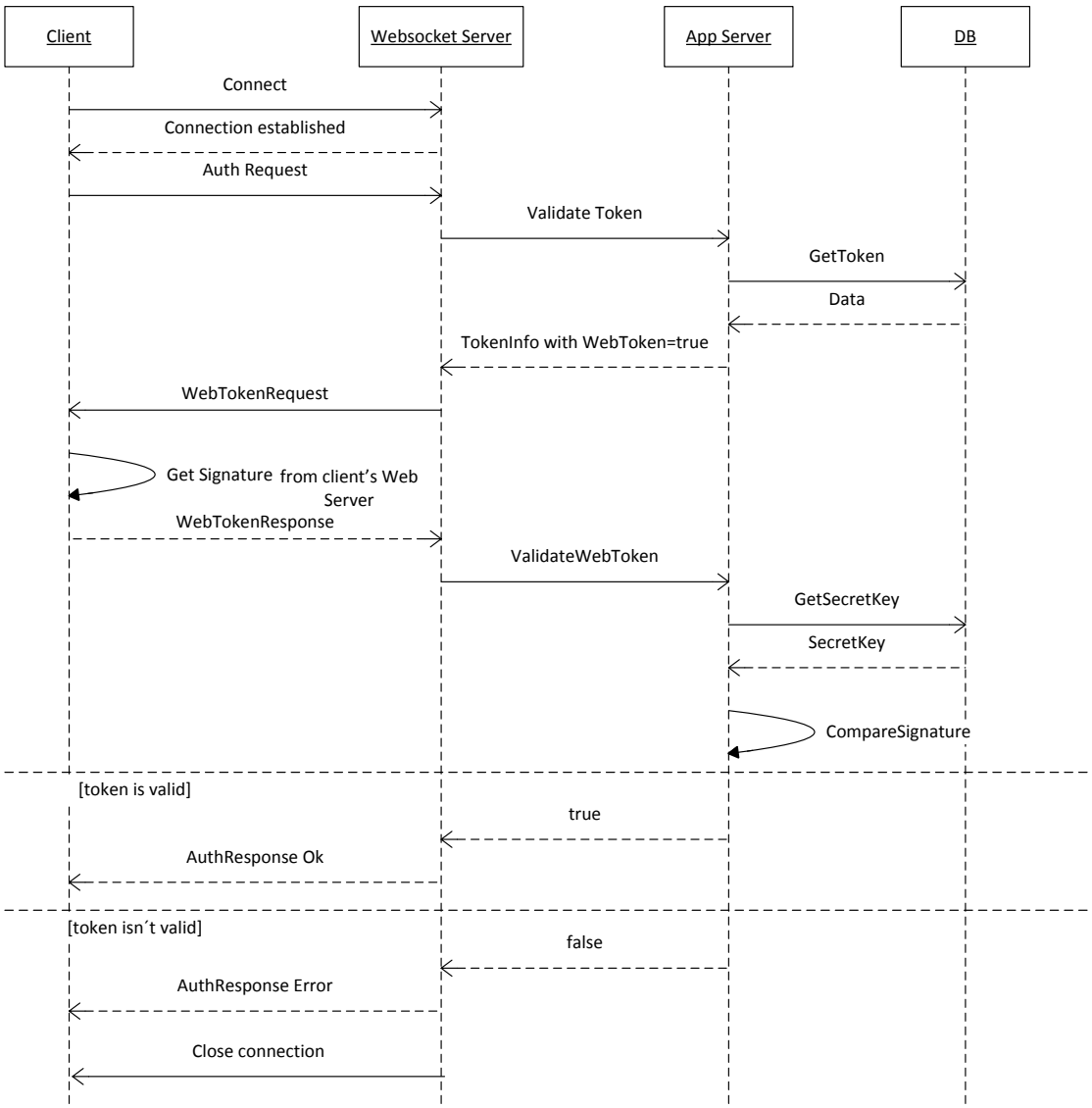

FIGURA 10 - DIAGRAMA DE SEQUÊNCIA DO PROCESSO DE AUTENTICAÇÃO COM *WEBTOKEN*

4.2.4 GESTÃO DE STREAMS E PUBLISH/SUBSCRIBE

A gestão de *Streams* é feita pelo *StreamManager*. Sempre que um cliente pretende subscrever uma *Stream*, o *StreamManager* cria um objeto *Subscription*, que contém uma referência para a *Connection* do cliente e é notificado sempre que a *Stream* recebe dados.

Uma *Stream* não é mais do que um agregador de subscrições, e que regista um *callback* no *PubSubClient*, e cancela o registo quando deixa de ter subscrições. Sempre que recebe dados pelo *PubSubClient*, notifica todas as suas subscrições.

A *Stream* depende do *PubSubClient*, mas apenas utiliza os métodos definidos no seu *prototype*, de forma a desacoplar o funcionamento das *Streams* e *Subscriptions* da implementação do *PubSubClient*. O objeto utilizado como *PubSubClient* é o *PubSubRedisClient*, que usa o módulo *redis* para *NodeJS*. Em *Redis*, as operações de *Publish* e *Subscribe* usam tópicos e, como tal, cada *Stream* tem associado um tópico único em toda a plataforma, com o seguinte formato: ***{appKey}:{streamName}***, em que *appKey* é a chave pública da *account* que detém a *Stream*.

As Figuras 12, 13, 14 e 15 contêm os diagramas de sequência das ações *Publish*, *Subscribe* e receção de mensagens nas *Streams* quando o *PubSubRedisClient* é notificado ao receber uma nova mensagem num determinado tópico do *Redis* e entrega-a à respetiva *Stream*, para que a faça chegar aos clientes. No caso das ações *Publish* e *Subscribe*, quando a validação das permissões falha do lado do cliente, é lançada uma excepção com essa indicação. Caso a validação das permissões falhe do lado do servidor, é fechada a ligação, uma vez que esta situação nunca deverá ocorrer porque as permissões são validadas primeiro do lado do cliente. A validação das permissões só poderá falhar do lado do servidor se não forem validadas do lado do cliente, o que pode acontecer se o cliente não se estiver a ligar ao sistema com uma das duas APIs cliente disponibilizadas neste trabalho.

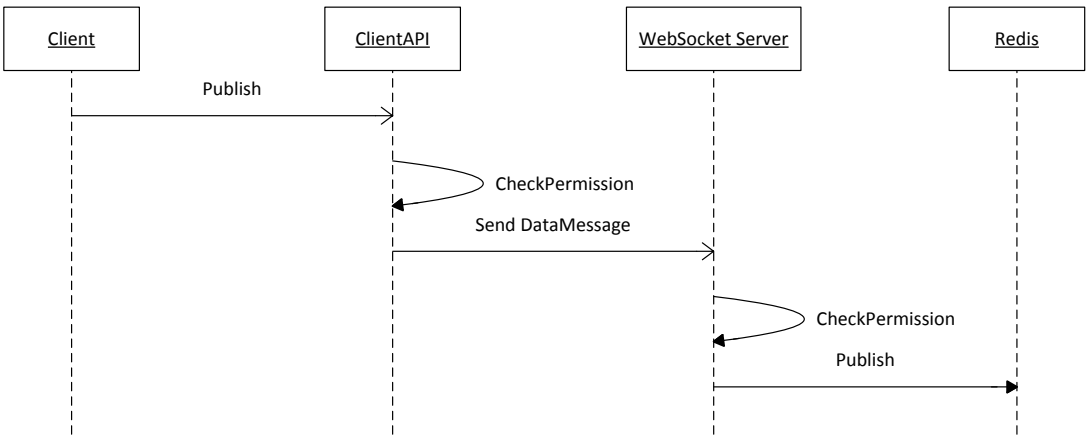

FIGURA 11 - DIAGRAMA DE SEQUÊNCIA DA AÇÃO PUBLISH

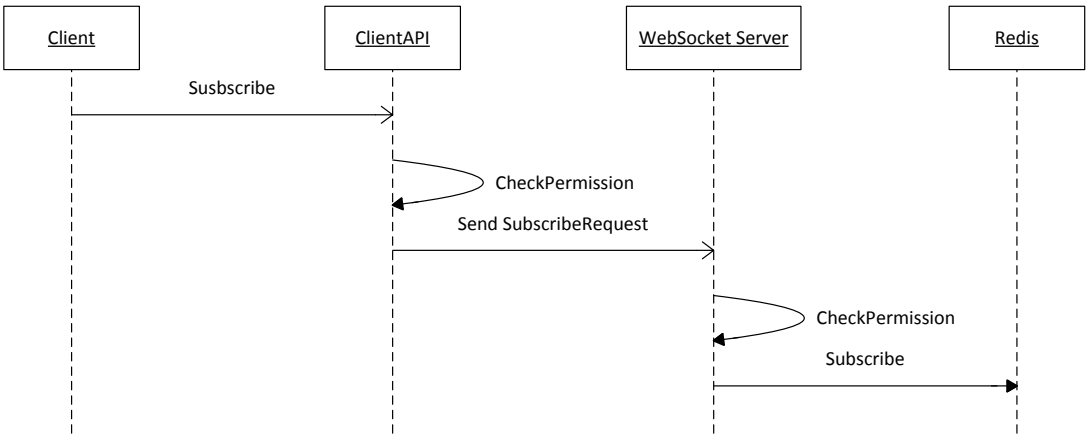

FIGURA 12 - DIAGRAMA DE SEQUÊNCIA DA AÇÃO SUBSCRIBE

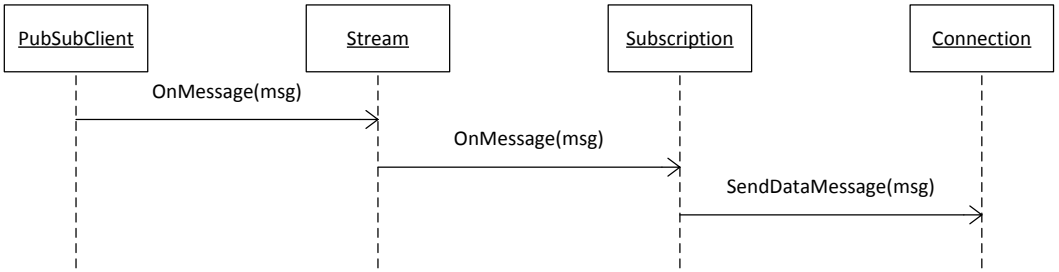

FIGURA 13 - RECEÇÃO DE MENSAGEM NA *STREAM* SEM *QUERY*

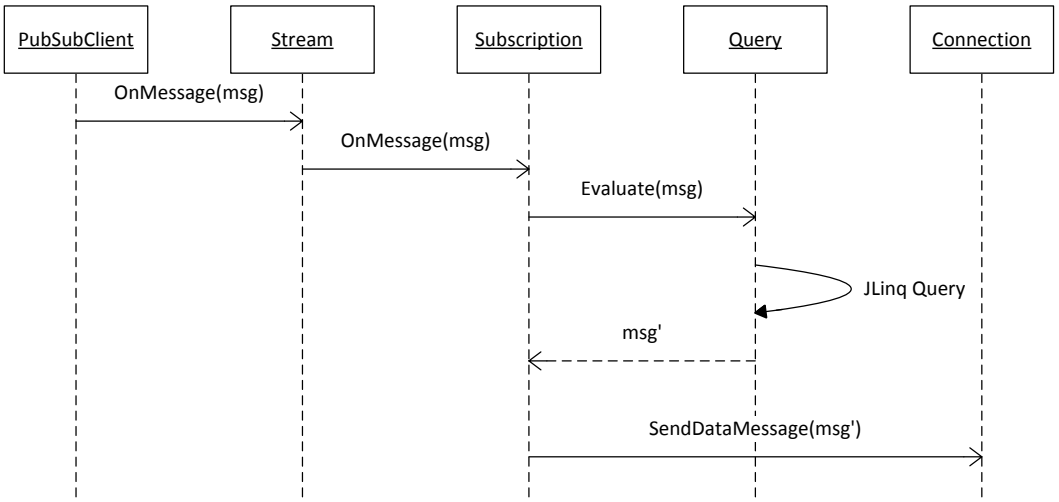

FIGURA 14 - RECEÇÃO DE MENSAGEM NA *STREAM*, COM *QUERY*

4.2.5 COMUNICAÇÃO COM OS SERVIÇOS DE BACKEND

A comunicação com os serviços de *backend* é realizada pelo objeto *ServiceClient*. Este objeto fornece uma API considerada *high-level*, no sentido em que quem o utiliza apenas se tem que conhecer o nome do serviço, nome da operação e os argumentos esperados. Internamente, este objeto usa o objeto *AsyncClient*, que é o cliente da *Framework* de comunicação desenvolvida com *ZeroMQ*, e que também é disponibilizada neste trabalho. A Listagem 2 contém o código do método *requestData* no objeto *ServiceClient* e que serve de *proxy* para invocar qualquer serviço do *backend*. O funcionamento do objeto *AsyncClient* será explicado no tópico 4.7 como sendo um cliente para *NodeJS* da *Framework* de comunicação desenvolvida.

```
//serviceName, operation, arg1, arg2, arg3, arg4, arg5, callback
//callback: function(result, err)
this.requestData = function () {


    var svc = arguments[0];
    var op = arguments[1];
    var callback = arguments[arguments.length - 1];

    var args = new Array();

    for (var i = 2, argsIdx = 0; i < arguments.length - 1; ++i, ++argsIdx) {
        args[argsIdx] = _prepareArgument(arguments[i]);
    }

    var request = {
        serviceName: svc,
        body: JSON.stringify({
            o: op,
            a: args
        })
    };

    _asyncClient.requestData(request, callback);
}
```

LISTAGEM 2 - IMPLEMENTAÇÃO DO MÉTODO REQUESTDATA

4.3 SISTEMA DE PUBLISH/SUBSCRIBE

O sistema de *Publish/Subscribe* utilizado neste trabalho é o *Redis*, o que permite escalar horizontalmente os servidores de *WebSocket*, pois caso haja mais do que um servidor de *WebSocket* com subscrições para a mesma *Stream*, estarão subscritos no mesmo tópico do *Redis* e recebem ambos a mesma mensagem sempre que haja uma publicação nesse tópico.

Atualmente este ponto da arquitetura pode ser considerado um ponto de falha e de estrangulamento. No entanto, é possível configurar vários nós de *Redis* em *cluster*. A implementação atual da funcionalidade *Publish/Subscribe* do *Redis* quando está configurado em *cluster*, consiste em escalar horizontalmente as subscrições mas não as publicações.

Um cliente pode subscrever um tópico em qualquer nó, mas quando publica uma mensagem, a mensagem é publicada para todos os nós, mesmo os que não têm subscrições nesse tópico. Segundo a documentação do *Redis*, este processo será otimizado em versões futuras.

No âmbito deste trabalho não se achou necessário configurar o *Redis* em *cluster*, uma vez que se trata de um trabalho académico, e não um cenário real onde são realizadas ações de *publish/subscribe* com carga suficiente que justifique uma configuração em *cluster*.

4.4 WEB SERVER

O servidor *Web* foi desenvolvido em *NodeJS* e reutiliza alguns componentes desenvolvidos para o servidor de *WebSockets*. Neste servidor são disponibilizados o serviço *Web* que permite gerar *tokens* de autenticação, bem como ficheiros de conteúdo estático. Embora esteja a ser disponibilizado apenas um serviço, este servidor poderia disponibilizar também serviços para a gestão de *accounts* de forma a serem consumidos por uma aplicação *Web*.

4.4.1 ARQUITETURA

Na Figura 15 é apresentada a arquitetura lógica do servidor *WebServer*.

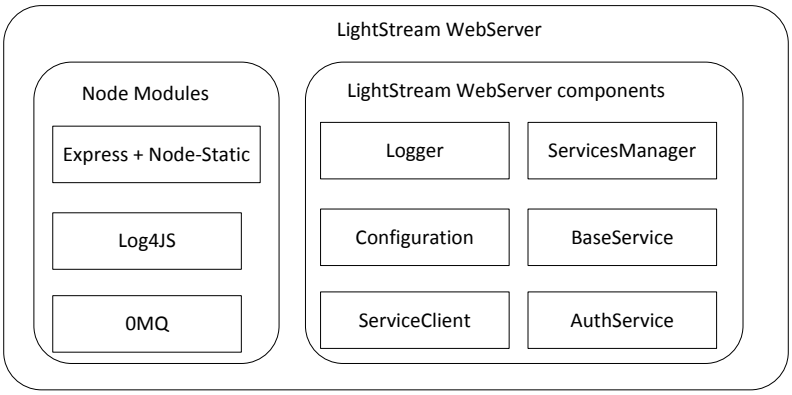

FIGURA 15 - ARQUITETURA LÓGICA DO SERVIDOR *WEBSERVER*

Esta camada tira partido dos módulos *Express* e *Node-Static*, sendo este último para servir ficheiros de conteúdo estático, como por exemplo o ficheiro da API cliente para *Javascript*. Através do módulo *Express*, o componente *ServicesManager* regista os serviços *Web* que estejam disponíveis em *endpoints* HTTP, seguindo uma convenção de nomes. A comunicação com o *backend* é feita da mesma forma que no servidor de *WebSockets*, através do *ServiceClient*.

A Listagem 26 contém o código do arranque do servidor, onde é possível ver a criação do servidor HTTP e o servidor de conteúdo estático, bem como a inicialização do componente *ServicesManager*.

4.4.2 WEB SERVICES

O componente responsável pela gestão dos serviços *Web* é o *ServicesManager*. Quando o servidor é inicializado, o *ServicesManager* trata de registar os vários serviços em *endpoints* HTTP.

Cada serviço deve derivar de *BaseService* e deve especificar a propriedade *name*, bem como implementar o método *install*.

A Listagem 3 contém o código do componente *ServicesManager*.

```
var express = require('express');
var ls = require('lightstream');

var ServiceManager = function () {

    var _registeredServices = {};

    this.registerService = function (name, service) {

        _registeredServices[name] = service;
    };

    this.installServices = function (httpServer, config) {

        var root = config.prefix || '/';

        ls.Logger.info('installing lightstream services');

        httpServer.use(express.bodyParser());
        httpServer.set("jsonp callback", true)

        for (var name in _registeredServices) {

            var serviceRoot = root + '/' + name + '/';
            ls.Logger.info('installing service ' + name + ' on ' + serviceRoot);
            _registeredServices[name].install(httpServer, serviceRoot);
        }
    };

}

module.exports = ServiceManager;
```

LISTAGEM 3 - COMPONENTE *SERVICESMANAGER*

O serviço disponibilizado neste servidor é o serviço *AuthService,* que serve de ponte entre a camada de *frontend* e o *backend*, onde está implementado de facto o serviço de geração de *tokens*. Embora apenas esteja a ser disponibilizado um serviço, a flexibilidade da arquitetura permite que sejam expostos novos serviços com o mínimo de esforço, bastando registá-los no *ServicesManager*.

O método *GenerateToken* está registado como sendo o *handler* que trata os pedidos quando o endereço termina em */auth/GenerateToken*. Note-se que *auth* é o nome do serviço. A Figura 16 ilustra este processo.

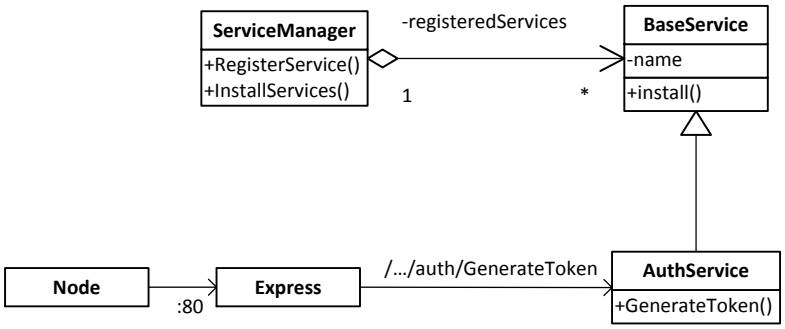

FIGURA 16 - EXEMPLO DE CHAMADA AO SERVIÇO AUTHSERVICE

4.4.3 CONTEÚDO ESTÁTICO

O conteúdo estático é colocado na directoria *public* do servidor e servido através do módulo *node-static* ao invés de usar directamente o módulo *Express*, uma vez que este requer o registo de um *handler* para o *endpoint* de cada ficheiro. No caso do módulo *node-static* apenas é registado um *handler* que consegue servir qualquer ficheiro, desde que esteja na directoria *public*. A Listagem 4 contém o código onde é feito este registo.

```
//Static file server
var fileHandler = new static.Server(__dirname + '/public');

var fileServer = express.createServer(function (req, res) {
        fileHandler.serve(req, res);
});

fileServer.listen(8020, '0.0.0.0');
```

LISTAGEM 4 - REGISTO DE HANDLER PARA CONTEÚDO ESTÁTICO

4.5 BALANCEAMENTO DE CARGA E SSL

O balanceamento de carga nos servidores *WebServer* e *WebServerSocket* é realizado por *software*, através do balanceador *HAProxy*, que executa em máquinas *Linux*. Este balanceador, para além da sua popularidade e maturidade, é recomendado pelo *SockJS* por funcionar com *WebSockets* e com as restantes técnicas de *fallback* que *SockJS* implementa. Note-se que quando a ligação não é feita por *WebSocket*, o *SockJS* utiliza um identificador de sessão com o cliente através da *query path* do *URL* utilizado.

Em cenários onde é feito balanceamento de servidores de *WebSocket*, é necessário que todos os pedidos realizados pelo mesmo cliente sejam encaminhados para o servidor onde foi criada a sessão. O balanceador *HAProxy* consegue garantir este tipo de afinidade especificando regras sobre o *URL*, via configuração. No âmbito deste trabalho, está-se a usar o *HAProxy* para balancear os pedidos realizados a toda a camada de *frontend*, ou seja, aos servidores de *WebSocket* e aos servidores *Web*, através da definição de regras sobre os *URLs* e atribuindo grupos de servidores a essas regras. Cada grupo de servidor pode ter configurado vários nós com endereços diferentes, que serão usados pelo *HAProxy* para balancear os pedidos. A configuração utilizada é apresentada na Listagem 27.

O balanceador *HAProxy* não suporta *SSL*. No âmbito deste trabalho, a solução adotada para resolver este problema foi colocar um componente à escuta no porto 443, receber a informação por *SSL*, e encaminhá-la para o porto 80, onde o *HAProxy* está à escuta e já tem toda a lógica de balanceamento configurada. Este componente que escuta os pedidos por *SSL* é um *bottle-neck* no sistema.

No âmbito deste trabalho foram investigados os componentes *Stunnel* e *NginX*, capazes de receber tráfego por *SSL* e encaminhá-lo para um destinatário, neste caso o endereço onde o *HAProxy* está à escuta. Estes dois componentes são os componentes tipicamente recomendados nas comunidades *open-source*, onde mesmo sem a realização de testes de performance, a opinião geral refere que o *Stunnel* tem melhor performance que o *NginX*, e suporta *SSL* com *WebSockets*, enquanto o *NginX* apenas suporta *SSL* para pedidos *HTTP*.

Infelizmente, não foi possível realizar uma configuração do componente *Stunnel* com sucesso em tempo útil, ao contrário do componente *NginX* cuja instalação e configuração foi bastante simples, pelo que, por falta de tempo, optou-se por avançar com esta solução. O lado negativo desta solução é que não se consegue criar ligações de *WebSocket* com *SSL*. No entanto, visto que a *API* de *SockJS* (*Javascript*) usa mecanismos de *fallback* quando não consegue estabelecer a ligação por *WebSocket*, é possível realizar ligações por *SSL* com *long-polling*.

Em relação ao serviço de geração de *tokens*, este pode ser invocado por *SSL* sem qualquer problema, uma vez que o pedido é feito por *HTTPS*, recebido pelo *NginX* e encaminhado para o *HAProxy*, que por sua vez irá balancear pela lista de servidores configurados para a camada de servidores *Web*.

A Listagem 28 contém a configuração utilizada no *NginX*. O certificado utilizado nos testes realizados com *SSL* é um certificado *self-signed*, gerado com a ferramenta *openssl* para *Linux*.

4.6 APPLICATION SERVER

A camada *Application Server* representa a camada de *backend* da solução, onde se encontra toda a lógica de geração de *tokens* e operações de acesso à base de dados onde está a informação das *accounts*. Esta camada expõe algumas das suas operações à camada de *frontend* através de serviços. O diagrama com o modelo de dados utilizado nesta camada encontra-se na Figura 28.

A Figura 17 contém a arquitetura lógica desta camada. Na Figura 18 encontra-se o diagrama *UML* de classes dos componentes utilizados nesta camada.

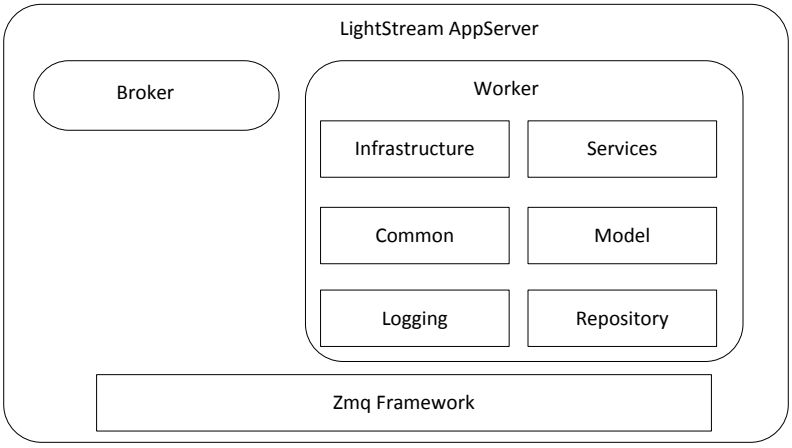

FIGURA 17 - ARQUITETURA LÓGICA DO SERVIDOR APLICACIONAL

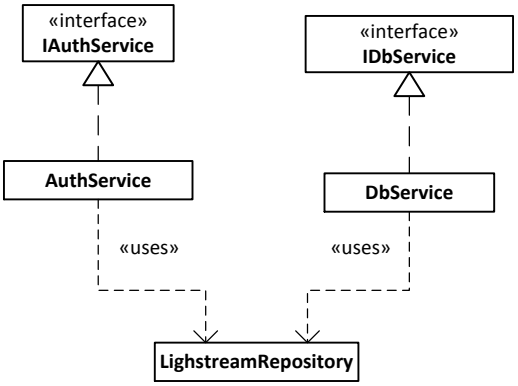

FIGURA 18 - DIAGRAMA UML DE CLASSES DOS SERVIÇOS

Esta camada divide-se em dois grandes componentes, *Workers* e *Broker*.

O componente *Broker* é o componente responsável pela escalabilidade horizontal, realizando o balanceamento de carga pelos vários *Workers*.

Um *Worker* é um processo que, sendo executado numa máquina com acesso ao endereço do *Broker*, liga-se a este e passa automaticamente a estar incluído na lógica de balanceamento. No âmbito deste trabalho, os *Workers* estão a expor lógica aplicacional do sistema através de serviços.

Os serviços implementados acedem ao repositório de dados através da classe *LightStreamRepository*.

As listagens 29 e 30 contêm o código de arranque de um *Worker* e do *Broker*, respetivamente.

4.6.1 CAMADA DE SERVIÇOS

As operações dos serviços foram implementadas de forma assíncrona, sendo utilizada a API assíncrona do *.NET* 4, recorrendo a *Tasks*. No entanto, para manter compatibilidade com o modelo de programação assíncrono do *.NET*, e que também é utilizado nas interfaces dos serviços *WCF*, optou-se por utilizar a convenção **Begin**Method / **End**Method, sendo retornado objectos *Task* nos métodos *Begin*, uma vez que uma *Task* implementa *IAsyncResult*.

Note-se que não está a ser utilizado *WCF*. Os serviços são expostos/consumidos pela *Framework* de comunicação desenvolvida neste trabalho.

Os contratos dos serviços expostos estão no projeto *LightStream.Infrastructure*. No âmbito deste trabalho foi criado o contrato *IAuthService*, com operações relacionadas com a gestão e validação de *tokens*, e o contrato *IDbService* com operações relacionadas com a gestão de *accounts*. Uma vez que não foi implementada a aplicação *Web* onde seria possível realizar toda a gestão de *accounts*, o contrato *IDbService* apenas expõe a operação *ResetDatabase*, cuja implementação elimina os registos da base de dados e insere *accounts* com base nos dados de um ficheiro XML. Este ficheiro encontra-se na Listagem 34.

Na Listagem 5 é possível ver a implementação da operação assíncrona *ValidateWebToken* do serviço *AuthService*, onde o método *BeginValidateWebToken* não é bloqueante e devolve um objeto TaskCompletionSource. Este objeto será utilizado para sinalizar o sucesso (*SetResult*) ou insucesso (*SetException*) da operação, podendo ser invocado o método *EndValidateWebToken* a fim de obter e retornar o resultado da operação. O método *SetResult* só é chamado sobre o objeto *TaskCompletionSource* após avaliar o resultado da chamada ao método *GetAccount* do *LightStreamRepository*, que também é assíncrono e devolve uma *Task<Account>*, de forma a poder registar um callback pelo método *ContinueWith*. A implementação das restantes operações é semelhante, quer a nível dos serviços quer a nível do *LightStreamRepository*.

Note-se que a classe *LightStreamRepository* utiliza a API oficial do *MongoDB* para *.NET* para realizar o acesso à base de dados. Embora a classe *LightStreamRepository* exponha métodos com assinatura assíncrona, a API do *MongoDB* para *.NET* é síncrona, ou seja, existe sempre uma *thread* bloqueada à espera da resposta. O cenário ideal seria a API do *MongoDB* para *.NET* ser completamente assíncrona, utilizando os métodos assíncronos para ler e escrever do *socket* e assim tirar partido de *I/O Completion Ports* para realizar *Non-Blocking I/O*.

```
public IAsyncResult BeginValidateWebToken(WebTokenValidationRequest request, AsyncCallback
callback, object state)
{
    Log.DebugFormat("BeginValidateWebToken request");

    var result = new TaskCompletionSource<bool>();

    if (!this.IsValidRequest(request, result))
    {
        return result.Task;
    }

    this._repository.GetAccount(request.PublicKey)
                        .ContinueWith(t =>
                        {
                            if (t.Exception != null)
                            {
                                result.SetException(t.Exception.InnerException ?? t.Exception);
                                return;
                            }

                            var account = t.Result;
                            if (account == null)
                            {
                                result.SetException(new ApplicationException("Invalid public
key"));
                                return;
                            }

                            var dataToSign = string.Format("{0}:{1}:{2}", request.PublicKey,
request.Token, request.ConnectionId);

                            var valid = Security.SignAndCompare(request.ValidationData, dataToSign,
account.SecretKey);

                            result.SetResult(valid);
                        });

    return result.Task;
}

public Result<bool> EndValidateWebToken(IAsyncResult asyncResult)
{
    Log.DebugFormat("EndValidateWebToken response");

    var task = asyncResult as Task<bool>;

    var result = new Result<bool>();

    if (task.Exception != null)
    {
        result.Errors.Add(new Error { Message = (task.Exception.InnerException ??
task.Exception).Message });
    }
    else
    {
        result.Data = task.Result;
    }

    return result;
}
```

LISTAGEM 5 - IMPLEMENTAÇÃO DO MÉTODO *BEGINVALIDATEWEBTOKEN*

4.6.2 *REPOSITÓRIO DE DADOS*

O repositório de dados utilizado foi o *MongoDB*. *MongoDB* permite especificar um tempo *TTL* para expiração automática de registos numa coleção. Esta funcionalidade foi utilizada para expirar *tokens* de autenticação automaticamente, ou seja, quando se tentar validar um *token* que já tenha expirado, este não existirá na base de dados e a validação concluirá que o *token* não é válido.

4.7 FRAMEWORK DE COMUNICAÇÃO COM ZEROMQ

No âmbito deste trabalho foi desenvolvida uma *Framework* de comunicação orientada a serviços que permite realizar comunicação entre *NodeJS* e *.NET*, não estando limitada a estas duas tecnologias, pois assenta sobre *ZeroMQ* (*0MQ*), que é uma *Framework* de comunicação *cross-platform* e assíncrona.

Esta *Framework* está dividida em duas camadas: baixo nível e alto nível. A camada de baixo nível é composta pelos componentes que utilizam a *Framework* do *0MQ*. Esta camada expõe uma API assíncrona para disponibilizar e implementar serviços, bem como criar clientes que enviam pedidos a serviços e recebem as respetivas respostas, de forma muito básica e sem impor qualquer contrato ou formato de mensagens. A camada de alto nível expõe uma API que permite disponibilizar serviços com base em contratos tal como é feito com *WCF*, sem qualquer dependência para esta *Framework*. Do mesmo modo a camada de alto nível expõe uma API que permite chamar operações de qualquer serviço com base num contrato.

Note-se que, embora neste trabalho os clientes dos serviços sejam clientes *NodeJS*, optou-se por implementar também uma API cliente para esta *Framework* em *.NET*, de forma a torná-la 100% utilizável por aplicações *.NET*.

4.7.1 *INTRODUÇÃO AO ZEROMQ*

ZeroMQ (*0MQ*) é uma *Framework* de comunicação cuja *API* é baseada em *sockets* que podem ser usados para enviar mensagens com vários tipos de transporte, tais como *inter-process*, *in-process*, *TCP* e *multicast*. A *Framework* permite ligações *N para N* entre *sockets* para implementar padrões de comunicação como *PubSub*, *Task-Distribution* e *Request-Reply*. Internamente a *Framework* usa um modelo de *I/O* assíncrono e *message queues*, permitindo comunicação assíncrona entre componentes que podem ser desligados e ligados a qualquer momento. ZeroMQ é *cross-platform*, e é suportada por mais de 20 linguagens de programação, entre as quais *C*, *C++*, *C#*, *Java*, *NodeJS* e *Python*.

No âmbito deste trabalho, esta *Framework* foi utilizada para implementar o padrão de comunicação *RequestReply*, sobre o transporte *TCP*.

O principal objeto com o qual se interage na API do *0MQ* é o *ZMQSocket*, que representa uma abstração sobre um *socket*. Quando um *ZMQSocket* é criado, é especificado qual o tipo de utilização que o *socket* terá, o que influencia o tipo de operações e o comportamento que o *socket* terá quando envia e recebe mensagens. As mensagens são enviadas em tramas, que são interpretadas de maneira diferente conforme o tipo de *socket* que as está a ler. Por exemplo, a primeira trama de uma mensagem pode ser o tópico quando se faz *PubSub*. No caso de *RequestReply*, a primeira trama é o endereço do *socket* que realiza o pedido.

Os tipos de *socket* utilizados no âmbito deste trabalho são *Dealer* e *Router*.

ZMQSockets do tipo *Dealer* são *sockets* que podem ser usados para enviar e receber dados de forma assíncrona, a qualquer momento. Um *socket* do tipo *Dealer* pode estabelecer ligação com vários *sockets*, e quando envia dados para os *sockets* aos quais está ligado faz balanceamento dos envios pelos vários *sockets*, de forma *round-robin*.

ZMQSockets do tipo *Router* são usados para enviar dados para destinatários conhecidos, sendo úteis no envio de respostas a *sockets* clientes. Sempre que um *socket* se liga a um *socket* do tipo *Router*, este guarda o seu endereço. Quando um *socket* do tipo *Router* é usado para enviar dados para os *sockets* aos quais está ligado, assume que a primeira trama da mensagem a enviar contém o endereço do *socket* destinatário e utiliza-o para enviar os dados.

Na Figura 19 está representada a arquitetura e tipos de *sockets* utilizada na comunicação entre os vários componentes do sistema.

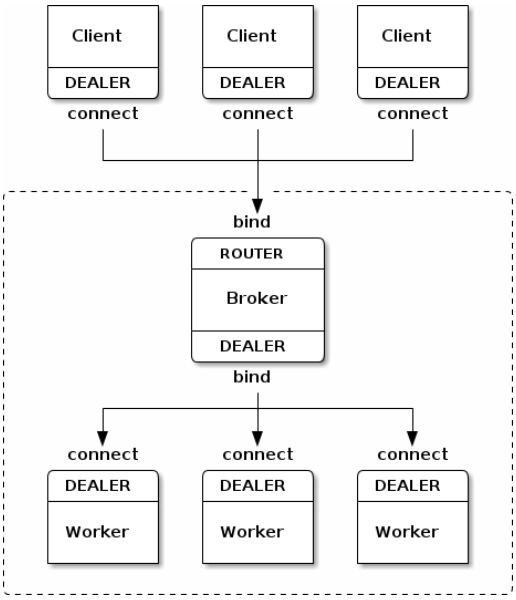

FIGURA 19 - ARQUITETURA COM *ZEROMQ*

O componente central é chamado de *Broker* e permite realizar o desacoplamento entre os clientes e os *workers*. O *Broker* irá balancear os pedidos pelos vários *workers*, de forma *round-robin*. Quando um *worker* tem uma resposta para dar ao cliente, envia a mensagem para o *Broker* e este saberá qual o cliente a quem deve entregar a mensagem. Note-se que os únicos endereços conhecidos na arquitetura são os endereços do *Broker*.

Nesta arquitetura o *Broker* pode ser considerado como um ponto de falha. Existem outros padrões de *RequestReply* mais complexos, mencionados na documentação do *0MQ*, que permitem escalar horizontalmente o número de *Brokers*, evitando assim pontos de falha. No âmbito deste trabalho apenas foi implementado o padrão apresentado na Figura 19.

4.7.2 CAMADA DE BAIXO NÍVEL

A Figura 20 contém os diagramas UML de classes dos componentes que fazem parte da camada de baixo nível, e serve de apoio ao longo deste tópico.

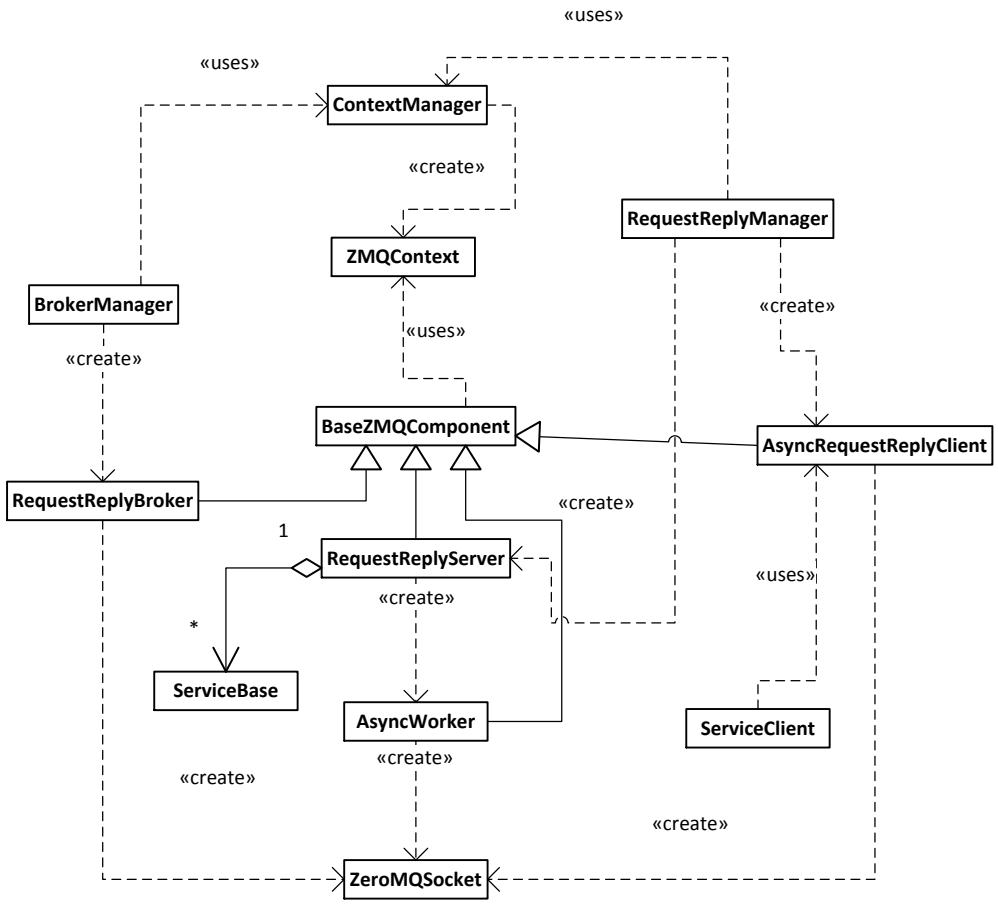

FIGURA 20 - DIAGRAMA UML DE CLASSSES DA CAMADA DE BAIXO NÍVEL

Todos os componentes que utilizam diretamente *0MQ* derivam de *BaseZmqComponent*, e partilham o mesmo objeto *ZMQContext*.

4.7.2.1 Broker

O *Broker* é um componente que utiliza dois *sockets* chamados de *frontend* e *backend*. No *socket* de *frontend* são ligados os clientes. No *socket* de *backend* são ligados os *workers*. Na Figura 21 é apresentado um diagrama com o funcionamento interno do *Broker*.

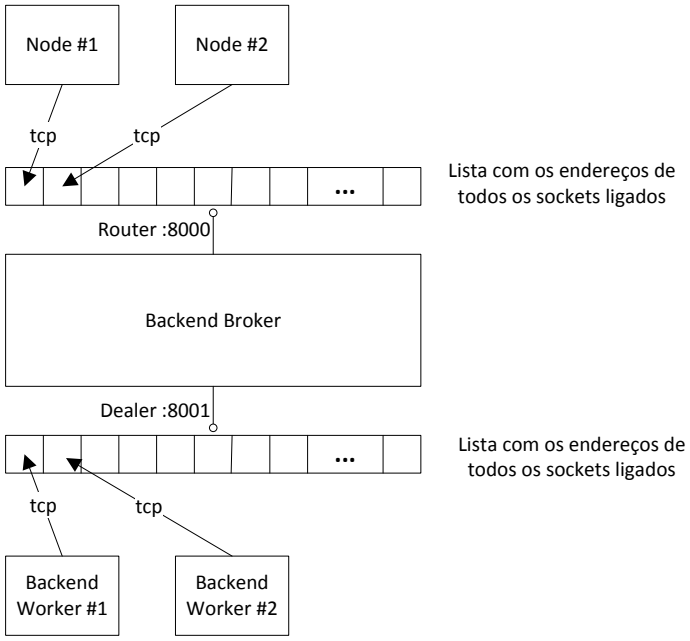

FIGURA 21 - ESTRUTURA INTERNA DO *BROKER*

A lista com os endereços dos *sockets* são meramente ilustrativas, uma vez que são geridas internamente pelo *ZMQSocket*, não estando acessível ao programador.

A Listagem 31 contém o código da implementação do *Broker*.

Todo o tráfego recebido no porto 8000 (*frontend*) vindo dos clientes, é enviado pelo *socket* de *backend* para os *workers*. Todo o tráfego recebido no porto 8001 (*backend*) é enviado pelo *socket* de *frontend* para o cliente com o endereço especificado na primeira trama da mensagem. Esta operação é realizada automaticamente, pelo método estático *ZMQ.Socket.Device.Queue*.

4.7.2.2 Mensagens enviadas

Na Figura 22 são apresentados os vários passos e os conteúdos das tramas das mensagens enviadas em cada passo.

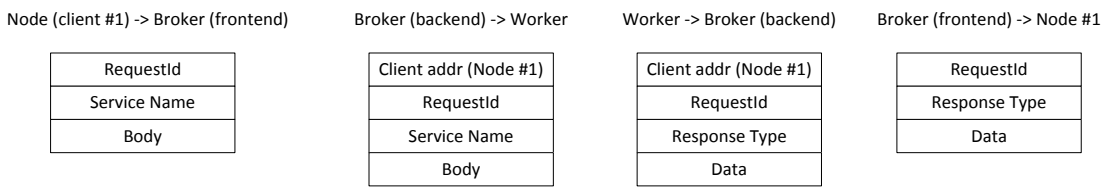

FIGURA 22 - MENSAGENS ENVIADAS ENTRE OS COMPONENTES

O formato das tramas que o servidor impõe é uma mensagem com pelo menos duas tramas, contendo o nome do serviço a invocar e o pedido em si (*body*). Todas as tramas que forem enviadas na mensagem antes destas duas, serão incluídas na resposta. A resposta contém um tipo de resposta e a resposta em si. O tipo de resposta pode ser *Success*, *Error* ou *Timeout*. Note-se que no diagrama com as mensagens enviadas a trama com o endereço do cliente é colocada automaticamente na mensagem pelo *0MQ*, pelo facto do *socket* de *frontend* do *Broker* ser do tipo *Router*, e retirada automaticamente quando a resposta é enviada ao cliente.

Como a comunicação é assíncrona, não é garantido que quando um cliente recebe uma resposta, esta seja relativa ao último pedido realizado. Para contornar este problema, sempre que o cliente envia um pedido é gerado um identificador de pedido (único naquele cliente) ao qual fica associado o *callback* da resposta, e é inserido na trama. Quando o cliente recebe uma resposta, sabe a que pedido se refere e poderá invocar o *callback* respetivo.

4.7.2.3 Cliente

No âmbito deste trabalho foram implementados dois clientes (*.NET* e *NodeJS*) capazes de enviar pedidos, respeitando o formato de mensagens definido no tópico 4.7.2.2. Neste tópico é descrita a implementação do cliente de *.NET*. A implementação do cliente de *NodeJS* é similar à implementação de *.NET*, estando o seu código na Listagem 32 e Listagem 33.

O componente responsável pelo envio de pedidos é o *AsyncRequestReplyClient*. Este componente é exposto pela classe *RequestReplyManager*, é thread-safe e não usa *locks*. Quando o *AsyncRequestReplyClient* é inicializado, é criado um *ZMQSocket* do tipo *Dealer* que se liga ao *socket* de *frontend* do *Broker*.

Este objeto expõe o método *SendRequestAsync* que permite enviar um pedido a um determinado serviço e especificar os *callbacks* para tratamento da resposta em caso de sucesso, erro e *timeout*. Por cada pedido é gerado um identificador único (incremento atómico de um contador) e é criado um objeto *AsyncRequest* onde são guardados os *callbacks*, ao qual fica associado o identificador. O objeto *AsyncRequest* é colocado numa *ConcurrentQueue* e assim que possível será enviado pelo *socket*.

Um *ZMQSocket* não é *thread-safe*, e apenas deve ser utilizado pela *thread* onde foi criado. A forma recomendada para lidar com *ZMQSockets* assíncronos é colocar a *thread* que o criou num *event loop* do *0MQ* à espera que o *socket* esteja pronto para enviar ou pronto para receber, intercetar esses eventos e realizar o trabalho necessário. Esta técnica é utilizada, exceto quando se tem a garantia que não há trabalho a realizar, ou seja, quando não há pedidos para enviar. Neste caso, a *thread* bloqueia-se num *ManualResetEventSlim*, que é sinalizado quando há um novo pedido para enviar.

A Listagem 6 contém o código do AsyncRequestReplyClient que é colocado no *event loop*, onde se interceta o evento que indica que o *socket* está pronto para receber dados e o evento que indica que o *socket* está pronto para enviar dados.

```
//send requests
var senderPollItem = zmqSocket.CreatePollItem(ZMQ.IOMultiPlex.POLLOUT);
senderPollItem.PollOutHandler += (socket, evt) =>
{
    AsyncRequest request;
    if (this._requestsQueue.TryDequeue(out request))
    {
        if (request.IsCanceled)
        {
            return;
        }

        request.SentExpiresAt = DateTime.Now.AddMilliseconds(sendTimeout);

        //prepare response data
        this._sentRequests[request.RequestId] = request;

        //send request id
        socket.SendMore(BitConverter.GetBytes(request.RequestId));

        //send service name
        socket.SendMore(request.Service, defaultEncoding);

        //send request
        socket.Send(request.RawRequestbody);

        Interlocked.Increment(ref Diagnostics.Performance.ReqRepClientRequestsSentCounter);
    }
};

//process responses
var receiverPollItem = zmqSocket.CreatePollItem(ZMQ.IOMultiPlex.POLLIN);
receiverPollItem.PollInHandler += (socket, evt) =>
{
    var responseData = socket.RecvAll();

    //read request id
    var requestId = BitConverter.ToInt64(responseData.Dequeue(), 0);

    //read response type
    var responseType = responseData.Dequeue()[0]; //response type is just a byte

    //read response body
    var responseBody = responseData.Dequeue();

    Interlocked.Increment(ref Diagnostics.Performance.ReqRepClientResponsesReceivedCounter);

    this.OnResponse(requestId, responseType, responseBody);
};
```

LISTAGEM 6 - *HANDLERS* DO *EVENT LOOP* DO *0MQ*

Sempre que o *ZMQSocket* recebe uma resposta é chamado o método *OnResponse* onde é obtido o objeto *AsyncRequest* associado com o *requestId* que vem na resposta e é invocado o seu *callback* numa nova *task*, de forma a não atrasar novas leituras ou escritas pelos *handlers* que estão no *event loop*.

A Listagem 7 contém o código do *event loop*, realizado através do método *Poll* do objeto *ZMQContext*.

```
var receive = new ZMQ.PollItem[] { receiverPollItem };
var send = new ZMQ.PollItem[] { senderPollItem };
var sendReceive = new ZMQ.PollItem[] { senderPollItem, receiverPollItem };

var pollTimeout = 1000; //microseconds

while (this._running)
{
    //nao tem requests para enviar, esperar no evento dos requests
    if (this._requestsQueue.Count == 0 && this._sentRequests.Count == 0)
    {
        this._requestsWaitHandle.Wait();
    }
    //nao tem respostas para receber e tem requests para enviar, poll com send
    if (this._sentRequests.Count == 0 && this._requestsQueue.Count > 0)
    {
        base._zmqContext.Poll(send, pollTimeout);
    }
    //tem respostas para receber e nao tem requests para enviar, poll com receive
    else if (this._sentRequests.Count > 0 && this._requestsQueue.Count == 0)
    {
        base._zmqContext.Poll(receive, pollTimeout);
    }
    //tem respostas para receber e requests para enviar, poll com sendReceive
    else
    {
        base._zmqContext.Poll(sendReceive, pollTimeout);
    }

    //no more work to do? reset the event
    if (this._requestsQueue.Count == 0)
    {
        this._requestsWaitHandle.Reset();
    }
}
```

LISTAGEM 7 – *EVENT LOOP*

Note-se que os *handlers* são registados no *event loop* de acordo com o estado atual. Isto é feito porque apenas se deve registar *handlers* para o evento *ZMQ.IOMultiPlex.POLLOUT* (*socket* pronto a enviar) quando há de facto pedidos para enviar, pois caso contrário o *handler* estará sempre a executar, causando processamento desnecessário. O parâmetro *pollTimeout* (1*ms*) é especialmente útil quando se está a fazer *poll* com *handlers* no evento *ZMQ.IOMultiPlex.POLLIN*, pois se ao fim de *1ms* não houver dados para receber, a operação de *poll* nesse evento é cancelada, para dar lugar à operação de *poll* no evento *POLLOUT* e desta forma não atrasar as escritas.

4.7.2.4 Servidor

Do lado do servidor o componente que trata da receção das mensagens por *0MQ* é o *AsyncWorker*.

Este componente utiliza uma técnica com o *event loop* semelhante ao *AsyncRequestReplyClient*, em que faz *poll* em determinados eventos de acordo com o estado atual. No *handler* do evento *POLLIN* é recebido um pedido e criado um objeto *AsyncWorkerRequest*, onde são guardados os dados do pedido (*serviceName*, *body* e restantes tramas) e onde será posteriormente colocada a resposta. Este objeto é passado ao *RequestReplyServer*, onde é obtida a referência para o serviço especificado no pedido e é chamado o método *ProcessRequest*, numa nova *task*, de forma a não atrasar as leituras de novos pedidos.

No *handler* do evento *POLLOUT* são retirados os objetos *AsyncWorkerRequest*, já com a resposta, e é construída a mensagem com a resposta para ser entregue ao *Broker*, mantendo as duas tramas da mensagem original (*clientAddress* e *requestId*).

Quando o *RequestReplyServer* recebe um pedido para processar, cria um objeto *AsyncServerRequest* que encapsula o *AsyncWorkerRequest* e disponibiliza os métodos *SetResponse* e *SetError*, para que a implementação do serviço possa dar uma resposta do tipo *Success* ou *Error*, respetivamente. Quando qualquer um destes métodos for invocado, é colocada a respetiva resposta no objeto *AsyncWorkerRequest* e este é colocado numa *queue* do *AsyncWorker*, estando pronto para ser entregue ao *Broker*, que por sua vez entregará a resposta ao cliente.

A Listagem 8 contém um exemplo de um serviço que pode ser exposto no *RequestReplyServer*, através do método *RegisterService*.

```
public class TestService : ServiceBase
{
    public TestService()
            : base("Test")
    {

    }
    public override void ProcessRequest(AsyncServerRequest request)
    {
        var ping = request.GetRequest();
        var response = Pong(ping);

        request.SetResponse(response);
    }
    public string Pong(string ping)
    {
        return string.Format("TestService reply to {0}", ping);
    }
}
```

LISTAGEM 8 - EXEMPLO DE UMA IMPLEMENTAÇÃO DE SERVIÇO

Como se pode ver na Listagem 8, a criação de um serviço não é *user friendly*, e torna-se difícil expor várias operações no mesmo serviço com um ou mais argumentos. Do mesmo modo, o consumo de um serviço com várias operações ou vários argumentos utilizando o *AsyncRequestReplyClient* também não é *user-friendly*. Para resolver este problema foi criada uma camada de alto nível onde se facilita o consumo e disponibilização de serviços.

4.7.3 CAMADA DE ALTO NÍVEL

Esta camada tem o objetivo de simplificar a criação de serviços e o consumo dos mesmos. A Listagem 9 contém um exemplo de um serviço que pode ser exposto nesta camada. A Listagem 10 contém um exemplo de como se pode consumir este serviço.

```
public interface ICalcService
{
    double Add(double arg1, double arg2);

    IAsyncResult BeginMultiply(double arg1, double arg2, AsyncCallback callback, object state);
    double EndMultiply(IAsyncResult result);
}

public class CalcService : ICalcService
{
    public double Add(double arg1, double arg2)
    {
        return arg1 + arg2;
    }

    public IAsyncResult BeginMultiply(double arg1, double arg2, AsyncCallback callback, object
state)
    {
        return Task<double>.Factory.StartNew(() => arg1 * arg2);
    }

    public double EndMultiply(IAsyncResult result)
    {
        return (result as Task<double>).Result;
    }
}

//....

ServiceBase svc = new ContractService<CalcService, ICalcService>();
RequestReplyManager.Instance.RegisterService(svc);
```

LISTAGEM 9 - EXEMPLO DE UTILIZAÇÃO DA CAMADA DE ALTO NÍVEL (SERVIDOR)

```
ICalcService client = ContractServiceClient.CreateClient<ICalcService>();

var addResult = client.Add(1, 5);

client.BeginMultiply(2, 5,
    asyncResult =>
    {
        var multiplyResult = client.EndMultiply(asyncResult);
    }, null);
```

LISTAGEM 10 - EXEMPLO DE UTILIZAÇÃO DA CAMADA DE ALTO NÍVEL (CLIENTE)

4.7.3.1 Formato das mensagens

Para representar uma chamada a uma operação de um serviço foi criada a classe *ServiceOperationInvocation*, apresentada na Listagem 11. Esta classe contém uma propriedade com o nome da operação, que mapeia para o nome de um método na interface do serviço, e a lista de argumentos, com a mesma ordem em que são declarados no método, sob a forma de *strings JSON*, garantindo interoperabilidade deste objeto entre *NodeJS, .NET*, e outras tecnologias. O seriador utilizado é o *DataContractJsonSerializer*. O resultado da chamada ao método *Serialize* desta classe representa a trama *body*, enviada pelo *AsyncRequestReplyClient*.

```
[DataContract]
internal class ServiceOperationInvocation
{
    [DataMember(Name = "o")]
    public string Operation { get; set; }
    [DataMember(Name = "a")]
    public string[] Arguments { get; set; }

    public static ServiceOperationInvocation Deserialize(string raw)
    {
        return raw.FromJson<ServiceOperationInvocation>();
    }

    public string Serialize()
    {
        return this.ToJson();
    }
}
```

LISTAGEM 11 - CLASSE SERVICEOPERATIONINVOCATION

4.7.3.2 Cliente

Para poder consumir um serviço do lado do cliente, utilizando uma referência para a interface do serviço, seria necessário, para cada serviço, criar uma classe que implemente a interface e em cada método construir um objeto *ServiceOperationInvocation* com o nome do método e lista de argumentos, devidamente preenchido, e realizar o pedido ao *AsyncRequestReplyClient*.

No âmbito deste trabalho foi utilizada uma solução que, através da geração de código dinâmico consegue criar uma classe em *runtime* que implementa uma determinada interface, e permite intercetar todas as chamadas aos métodos dessa interface. A solução encontrada para a geração dinâmica de código foi a utilização da classe *DynamicProxy.ProxyGenerator* da biblioteca *Castle.Core*, como se pode ver na Listagem 12.

No método *Intercept* são intercetadas todas as chamadas e é criado o objeto *ServiceOperationInvocation* para realizar os pedidos ao *AsyncRequestReplyClient*. Note-se que o *AsyncRequestReplyClient* apenas disponibiliza um método assíncrono para realizar os pedidos, pelo que foi necessário criar um *SyncRequestReplyClient* para as invocações de métodos síncronos, como é o caso do método *Add* na interface *ICalcService*.

O objeto *SyncRequestReplyClient* utiliza o *AsyncRequestReplyClient* e bloqueia-se num *Monitor* com um *timeout*, à espera que o *callback* do *AsyncRequestReplyClient* seja chamado. No caso dos métodos assíncronos, foi criada uma classe que implementa *IAsyncResult* e guarda o *AsyncCallback* que o cliente passa para obter o resultado da operação. Este *callback* é invocado quando o cliente recebe a resposta pelo *AsyncRequestReplyClient*.

```
public class ContractServiceClient : ServiceClient, IInterceptor
{
    private static ConcurrentDictionary<Type, object> _proxyCache = new ConcurrentDictionary<Type,
object>();

    public static TInterface CreateClient<TInterface>() where TInterface : class
    {
        var interfaceType = typeof(TInterface);

        var dynamicProxy = _proxyCache.GetOrAdd(interfaceType, k =>
        {
            var generator = new ProxyGenerator();
            var serviceName = ExtractServiceNameFromInterface(interfaceType);
            TInterface proxy = generator.CreateInterfaceProxyWithoutTarget<TInterface>(new
ContractServiceClient(interfaceType, serviceName));

            return proxy;
        });

        return dynamicProxy as TInterface;
    }

    private string _serviceName;
    private Type _interfaceType;

    internal ContractServiceClient(Type interfaceType, string serviceName)
    {
        this._interfaceType = interfaceType;
        this._serviceName = serviceName;
    }

    public void Intercept(IInvocation invocation)
    {
        if (invocation.Method.Name.StartsWith("Begin"))
        {
            this.BeginAsyncInvocation(invocation);
        }
        else if (invocation.Method.Name.StartsWith("End"))
        {
            this.EndAsyncInvocation(invocation);
        }
        else
        {
            this.SyncInvocation(invocation);
        }
    }
    //...
}
```

LISTAGEM 12 - CLASSE CONTRACTSERVICECLIENT

A implementação do método *Intercept* podia ser otimizada, utilizando técnicas semelhantes às que foram usadas na implementação da classe *ContractService* (ver tópico 4.7.3.3). No entanto, como no âmbito deste trabalho os clientes são em *NodeJS*, apenas se pretendeu disponibilizar uma implementação completa e funcional de um cliente em *.NET*.

4.7.3.3 Servidor

A solução implementada para poder disponibilizar um serviço utilizando apenas uma classe que não derive de *ServiceBase* e uma *interface* para o contrato, foi criar a classe *ContractService* que deriva de *ServiceBase* e é parametrizada com o tipo da classe que implementa o serviço (*TImpl*) e o tipo da interface (*TInterface*), via genéricos. A Figura 23 contém o diagrama UML de classes com as dependências da classe *ContractService*.

No método *ProcessRequest* são recebidos todos os pedidos (objetos *ServiceOperationInvocation*), e com base no nome da operação, invoca o respetivo método sobre a instância do serviço, passando os argumentos. Esta operação é trivial se for implementada com *reflection*. No entanto, isto causaria peso desnecessário do lado do servidor, uma vez que as invocações por *reflection* têm um custo de processamento associado.

Para poder realizar as chamadas aos respetivos métodos do serviço seria necessário que a classe *ContractService* conhecesse *TInterface*, em tempo de compilação, para poder chamar os métodos sem ser por *reflection*. No entanto, isto é possível através da geração de código dinâmico, tal como é feito do lado do cliente com o *DynamicProxy*.

A solução implementada para a geração de código dinâmico tira partido das *Expression Trees* do *.NET*, em que são compiladas *lambda expressions* onde são realizadas as chamadas aos métodos concretos com a referência para *TInterface*. Como este processo é demorado, esta operação é realizada no construtor da classe *ContractService*, onde é gerado um objeto que implemente *IServiceOperation*, por cada operação do serviço, ao qual estará associado o *delegate* que sabe fazer a invocação do método. No caso das operações assíncronas são gerados dois *delegates*, um para o método *Begin* e outro para o *End*. Estes objetos são guardados num dicionário cuja chave é o nome da operação, de forma a poder ser indexado pelo nome da operação que vem no objeto *ServiceOperationInvocation*.

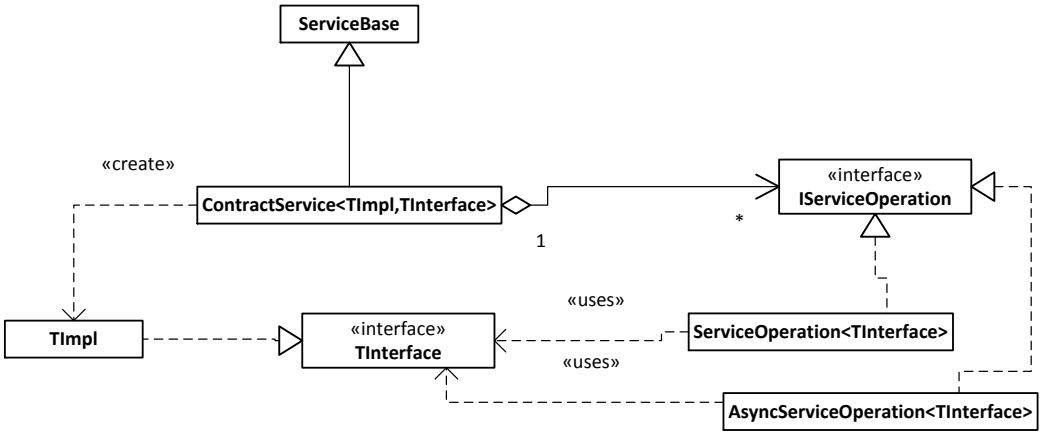

FIGURA 23 - DIAGRAMA UML DE CLASSES DA CAMADA DE ALTO NÍVEL (SERVIDOR)

As listagens 13 e 14 contêm o código da interface *IServiceOperation* e *ContractService*, respetivamente.

```
internal interface IServiceOperation
{
    Task<object> Invoke(ServiceOperationInvocation invocation);
    string Name { get; }
}
```

LISTAGEM 13 - INTERFACE *ISERVICEOPERATION*

```
public class ContractService<TImpl, TInterface> : ServiceBase
    where TImpl : class, TInterface, new()
{
    private readonly TInterface _service;

    private readonly Dictionary<string, IServiceOperation> _operations;

    public ContractService()
        : base(ExtractServiceNameFromInterface(typeof(TInterface)))
    {
        this._service = new TImpl();

        this._operations = typeof(TInterface).GetMethods()
                                             .Where(m => m.IsSyncOperation())
                                             .Select(m => new
ServiceOperation<TInterface>(_service, m) as IServiceOperation)
                                             .Union(
                                                typeof(TInterface).GetMethods()
                                                .Where(m => m.IsAsyncOperation())
                                                .Select(m => new
AsyncServiceOperation<TInterface>(_service, m) as IServiceOperation))
                                             .ToDictionary(op => op.Name, op => op);
    }

    public override void ProcessRequest(AsyncServerRequest request)
    {
        var invocation = ServiceOperationInvocation.Deserialize(request.GetRequest());

        var operation = _operations[invocation.Operation];

        operation.Invoke(invocation).ContinueWith(task =>
        {
            if (task.Exception == null)
            {
                try
                {
                    var result = task.Result;
                    request.SetResponse(result.ToJson());
                }
                catch (Exception exe)
                {
                    request.SetError(String.Format("Error serializing service response. Message:
{0}", exe.Message));
                }
            }
            else
            {
                request.SetError(String.Format("Error invoking service operation. Message: {0}",
task.Exception.InnerException.Message));
            }
        });
    }
    //..
}
```

LISTAGEM 14 - IMPLEMENTAÇÃO DA CLASSE *CONTRACTSERVICE*

De acordo com o tipo de operação, síncrono ou assíncrono, é criado um objeto, *ServiceOperation* ou *AsyncServiceOperation,* que implementa *IServiceOperation*. Estes objetos retornam o resultado da operação numa *Task<object>*. O resultado é seriado em *JSON* e é colocado no objeto *AsyncServerRequest* da camada de baixo nível, para que seja enviada a resposta para o cliente.

O delegate com a chamada ao método concreto do serviço é criado no construtor do *ServiceOperation* ou do *AsyncServiceOperation*, através da classe *ServiceOperationHelper*. As listagens 15 e 16 contêm o código das classes *ServiceOperation* e *AsyncServiceOperation*, respetivamente. No caso da classe *AsyncServiceOperation*, como a operação assíncrona segue o modelo de programação assíncrono do *.NET*, é utilizado o método *Task<object>.FromAsync*, que cria uma *Task* com base no objeto *IAsyncResult* devolvido pelo método *Begin*[*operation*] e respetivo método *End*[*operation*].

Os argumentos das operações são obtidos pela propriedade *Arguments* do objeto *ServiceOperationInvocation*. Uma vez que os argumentos estão representados em *strings*, é necessário convertê-los para os tipos originais dos parâmetros das operações. Para otimizar este processo é também gerado com *Expression Trees* um *delegate* que sabe fazer a conversão de cada argumento em *string* para o tipo original do parâmetro na operação. Uma vez que esta conversão tem que ser feita em todas as chamadas, o *delegate* é gerado de forma otimizada conforme o método.

Na classe *ServiceOperationHelper* encontra-se o código onde foram utilizadas as *Expression Trees*.

```
internal class ServiceOperation<TInterface> : IServiceOperation
{
    private readonly MethodInfo _methodInfo;
    private readonly TInterface _service;

    private readonly Func<object[], object> _operation;
    private readonly Func<string[], object[]> _convertArguments;

    public string Name { get; private set; }

    public ServiceOperation(TInterface serviceImpl, MethodInfo method)
    {
        this._service = serviceImpl;
        this._methodInfo = method;
        this.Name = this._methodInfo.Name;

        this._operation = ServiceOperationHelper.BuildOperation<TInterface>(serviceImpl, method);
        this._convertArguments = ServiceOperationHelper.BuildConvertArguments(method, false);
    }

    public Task<object> Invoke(ServiceOperationInvocation invocation)
    {
        var arguments = this._convertArguments(invocation.Arguments);

        return Task<object>.Factory.StartNew(() => this._operation(arguments));
    }
}
```

LISTAGEM 15 - IMPLEMENTAÇÃO DA CLASSE *SERVICEOPERATION*

```
internal class AsyncServiceOperation<TInterface> : IServiceOperation
{
    private readonly MethodInfo _methodInfo;
    private readonly TInterface _serviceImpl;

    private readonly Func<object[], AsyncCallback, object, IAsyncResult> _beginOperation;
    private readonly Func<IAsyncResult, object> _endOperation;
    private readonly Func<string[], object[]> _convertArguments;

    public string Name { get; private set; }

    public AsyncServiceOperation(TInterface serviceImpl, MethodInfo method)
    {
        this._serviceImpl = serviceImpl;
        this._methodInfo = method;

        this.Name = _methodInfo.Name.Substring(5, method.Name.Length - 5);

        this._beginOperation =
ServiceOperationHelper.BuildAsyncBeginOperation<TInterface>(serviceImpl, method);
        this._endOperation = ServiceOperationHelper.BuildAsyncEndOperation<TInterface>(serviceImpl,
typeof(TInterface).GetMethod(string.Format("End{0}", this.Name)));

        this._convertArguments = ServiceOperationHelper.BuildConvertArguments(method, true);
    }

    public Task<object> Invoke(ServiceOperationInvocation invocation)
    {
        var arguments = this._convertArguments(invocation.Arguments);

        var asyncResult = this._beginOperation(arguments, r => { }, null);
        return Task<object>.Factory.FromAsync(asyncResult, this._endOperation);
    }
}
```

LISTAGEM 16 - IMPLEMENTAÇÃO DA CLASSE *ASYNCSERVICEOPERATION*

5 TRABALHO FUTURO

No âmbito deste trabalho, ficou por implementar a aplicação *Web* que permitiria a criação de *accounts*, bem como a gestão de acessos às suas *Streams*.

No entanto, existem outros pontos que podem ser melhorados e que são apresentados de seguida.

Formato das mensagens de sistema

Atualmente o formato das mensagens de sistema utilizado é *JSON*. O processo de seriação/desseriação de mensagens em formato *JSON* requer mais processamento do que a utilização de um formato próprio, em que os vários campos da mensagem podem ser separados utilizando um carácter especial, uma vez que os campos das mensagens são conhecidos.

SSL com *WebSockets*

Ficou por realizar a configuração do componente *Stunnel* que permitia utilizar *SSL* com ligações por *WebSocket*.

API assíncrona para acesso à base de dados do *Backend*

Como a API oficial do *MongoDB* para *.NET* é síncrona, é necessário ter sempre uma *thread* bloqueada à espera da resposta em cada pedido. Como melhoria a este ponto sugere-se a investigação de novas APIs para *MongoDB*, ou inclusivamente utilizar outro tipo de base de dados para a qual exista uma *API* assíncrona que possa ser utilizada em *.NET*. Por exemplo, *ADO.NET* tem métodos assíncronos para acesso à base de dados.

Suporte para mais operadores nas queries sobre *Streams*

O motor de *queries* utilizado do lado do servidor é o *jLinq*. No âmbito deste trabalho, apenas foram disponibilizados alguns operadores suportados pelo *jLinq* (17). O objeto que encapsula a *query* na API cliente e o objeto que reconstrói a *query* em *jLinq* do lado do servidor foram implementados de forma a facilitar a incorporação de novos operadores. Seria interessante suportar todos os operadores de *jLinq*.

Avaliar a utilização de *Pub/Sub* com a nova versão do 0MQ

A versão do 0MQ utilizada neste trabalho é a versão 2.2, por ser a versão estável e recomendada quando se iniciou o desenvolvimento deste trabalho. Esta versão, embora permita criar configurações de *messaging* como o *Publish/Subscribe* com base em tópicos, tem a limitação de todos os subscritores receberem todas as mensagens que os *publishers* enviam, e internamente descartam aquelas que não interessam, causando maior tráfego na rede e desperdiçando processamento.

A versão 3.2 tráz melhorias no padrão *Publish/Subscribe*, na medida em que o *Publisher* só envia as mensagens para os subscritores que de facto subscrevem o tópico. No limite, se não houver nenhum subscritor num determinado tópico, o *publisher* nem chega a enviar mensagens.

Seria interessante avaliar a utilização desta versão de 0MQ com *Publish/Subscribe* em vez da utilização do *Redis*.

6 CONCLUSÃO

A realização deste trabalho permitiu disponibilizar uma plataforma de caracter inovador, que consegue colmatar muitos problemas existentes hoje em dia, na maior parte das aplicações *Web*, não só a nível de utilização eficiente de largura de banda, bem como a nível de interoperabilidade entre tecnologias.

Devido às características *cross-platform* da plataforma, é possível realizar comunicação de baixa latência entre sistemas desenvolvidos em tecnologias completamente diferentes, como é o caso de *Javascript* e *.NET*.

A implementação desta plataforma representou não só um desafio a nível de engenharia de *software*, mas também um desafio tecnológico, tendo em conta o número de tecnologias diferentes que são utilizadas nas diversas camadas.

Neste trabalho é também disponibilizada uma *Framework* de comunicação assíncrona, *cross-platform*, que permite integrar componentes de várias tecnologias diferentes, como é o caso de *NodeJS* e *.NET*, através da exposição de serviços com o mesmo tipo de interfaces utilizados em *WCF*. A implementação desta *Framework* representou um grande desafio e poderá vir a ser evoluída no futuro, como projeto *open source*.

7 REFERÊNCIAS

1. *WebSocket Specification.* [Online] Janeiro 1, 2012. http://dev.w3.org/html5/websockets/.

2. *HTML5 specification.* [Online] Janeiro 1, 2012. http://dev.w3.org/html5/spec/Overview.html.

3. *Comet (programming).* [Online] 2 1, 2012. http://en.wikipedia.org/wiki/Comet_(programming).

4. *Representational State Transfer.* [Online] 2 1, 2012. http://en.wikipedia.org/wiki/Representational_state_transfer.

5. *Node.js.* [Online] Janeiro 1, 2012. http://nodejs.org/.

6. *SockJS (Node).* [Online] 2 1, 2012. https://github.com/sockjs/sockjs-node.

7. *Socket.IO.* [Online] 2 1, 2012. http://socket.io/.

8. *HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer.* [Online] 2 1, 2012. http://haproxy.1wt.eu/.

9. *Redis.* [Online] 2 1, 2012. http://redis.io/.

10. *NoSQL.* [Online] 2 1, 2012. http://en.wikipedia.org/wiki/NoSQL.

11. *Redis PubSub documentation.* [Online] 2 1, 2012. http://redis.io/topics/pubsub.

12. *Redis clients.* [Online] 2 1, 2012. http://redis.io/clients.

13. *Task Parallel Library.* [Online] 2 1, 2012. http://msdn.microsoft.com/en-us/library/dd460717.aspx.

14. *ZeroMQ Guide.* [Online] 2 1, 2012. http://zguide.zeromq.org/page:all.

15. *ZeroMQ bindings.* [Online] 2 1, 2012. http://www.zeromq.org/bindings:_start.

16. *MongoDB.* [Online] 2 1, 2012. http://www.mongodb.org/.

17. *jLinq.* [Online] 7 1, 2012. http://hugoware.net/Projects/jlinq.

18. *WCF WEB API.* [Online] 2 1, 2012. http://wcf.codeplex.com/wikipage?title=wcf%20http.

19. *Asynchronous scalable web applications with real-time persistent long-running connections with SignalR.* [Online] Janeiro 1, 2012.

http://www.hanselman.com/blog/AsynchronousScalableWebApplicationsWithRealtimePersistentLongrunningConnectionsWithSignalR.aspx.

20. *Download WCF WebSockets prototype.* [Online] Janeiro 1, 2012. http://html5labs.interoperabilitybridges.com/prototypes/websockets/websockets/download.

21. *Getting started with WebSockets in the Windows 8 developer preview.* [Online] Janeiro 1, 2012. http://www.paulbatum.com/2011/09/getting-started-with-websockets-in.html.

22. *SignalR.* [Online] Janeiro 1, 2012. https://github.com/SignalR/SignalR.

23. *WCF Duplex via WebSocket.* [Online] Janeiro 1, 2012. http://developers.de/blogs/damir_dobric/archive/2011/11/26/wcf-duplex-via-websocket.aspx.

24. *WCF WebSockets prototype documentation.* [Online] Janeiro 1, 2012. http://html5labs.interoperabilitybridges.com/prototypes/websockets/websockets/documentation.

25. *WCF WebSockets prototype limitations.* [Online] Janeiro 1, 2012. http://html5labs.interoperabilitybridges.com/media/51621/readme.htm.

26. *What's New in the .NET Framework 4.5 Developer Preview.* [Online] Janeiro 1, 2012. http://msdn.microsoft.com/en-us/library/ms171868(v=vs.110).aspx.

8 ANEXOS

```
GET /mychat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

LISTAGEM 17 - WEBSOCKET HANDSHAKE REQUEST

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

LISTAGEM 18 - WEBSOCKET HANDSHAKE RESPONSE

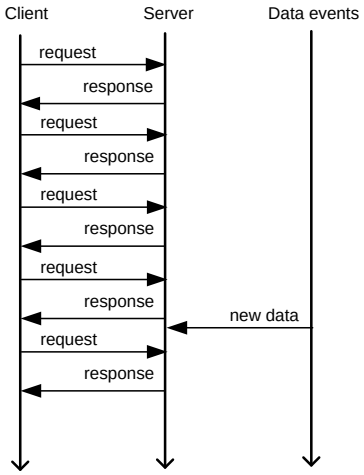

FIGURA 24 – AJAX POLLING

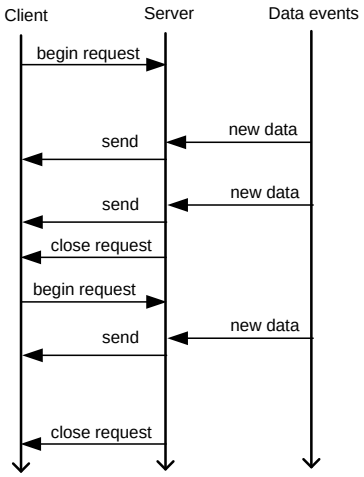

FIGURA 25 - AJAX LONG POLLING

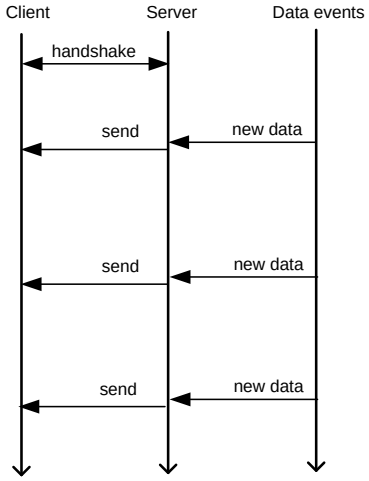

FIGURA 26 – WEBSOCKET

```
this._messageHandlers = new Dictionary<int, Action<string>>();

this._messageHandlers[(int)MessageType.AuthResponse] =
    msg => this.OnAuthenticationResponseMessage(AuthenticationResponseMessage.Deserialize(msg));
this._messageHandlers[(int)MessageType.WebTokenRequest] =
    msg => this.OnWebTokenRequestMessage(WebTokenRequestMessage.Deserialize(msg));
this._messageHandlers[(int)MessageType.SubscribeResponse] =
    msg => this.OnSubscribeResponseMessage(SubscribeResponseMessage.Deserialize(msg));
this._messageHandlers[(int)MessageType.UnsubscribeResponse] =
    msg => this.OnUnsubscribeResponseMessage(UnsubscribeResponseMessage.Deserialize(msg));
this._messageHandlers[(int)MessageType.Data] =
    msg => this.OnDataMessage(DataMessage.Deserialize(msg));
```

LISTAGEM 19 - REGISTO DE *EVENTHANDLERS* PARA OS VÁRIOS TIPOS DE MENSAGEM

```
void IConnectionInterceptor.OnData(string data)
{
    var envelope = MessageEnvelope.Deserialize(data);

    var messageHandler = _messageHandlers[envelope.MessageType];
    messageHandler(envelope.Message);
}
```

LISTAGEM 20 - RECEÇÃO DE MENSAGENS E CHAMADA DO HANDLER RESPETIVO

```
public static Task<string> RequestToken(TokenRequest request)
{
    TaskCompletionSource<string> result = new TaskCompletionSource<string>();

    if (!request.IsValid())
    {
        result.SetException(new ApplicationException("TokenRequest is not valid"));
        return result.Task;
    }

    var authUri = string.Format("https://{0}/services/auth/generateToken", request.Server);
    var queryString = request.ToQueryString();

    var requestUri = string.Format("{0}?{1}", authUri, queryString);

    var httpRequest = WebRequest.Create(requestUri);

    httpRequest.BeginGetResponse(
        asyncResult =>
        {
            try
            {
                var response = httpRequest.EndGetResponse(asyncResult);
                Result<Token> tokenResult;
                using (var stream = response.GetResponseStream())
                {
                    tokenResult = stream.FromJson<Result<Token>>();
                }

                if (tokenResult.Errors.Count > 0)
                {
                    result.SetException(new ApplicationException(tokenResult.Errors[0].Message));
                }
                else
                {
                    result.SetResult(tokenResult.Data.Value);
                }
            }
            catch (Exception ex)
            {
                result.SetException(ex);
            }

        }, null);

    return result.Task;
}
```

LISTAGEM 21 - IMPLEMENTAÇÃO DO MÉTODO *REQUESTTOKEN* NA API CLIENTE

```
LightStreamApi.RequestToken(new TokenRequest
{
    Server = "ubuntu/lightstream",
    AppKey = "the appkey",
    PrivateKey = "the private key",
    TTL = 120,
    Roles = new string[] { "public" },
    WebToken = true
}).ContinueWith(t =>
    {
        var token = t.Result;

        var client = LightStreamApi.CreateClient(TestsConfig.Create("acc1", token));

        client.Open((res, err) =>
        {
            if (err != null)
            {
                Console.WriteLine(err);
                return;
            }

            //ready to publish/subscribe
            //client.
        },
        (appKey, authToken, connectionId, onWebTokenValidationCallback) =>
        {
            //handler invoked to validate web token
            Console.WriteLine("doing some work to fetch the secret key for account acc1..");
            Thread.Sleep(500);

            var validationData = Security.Sign(string.Format("{0}:{1}:{2}", appKey, authToken,
connectionId), ("the secret key");

            onWebTokenValidationCallback(validationData);
        });
    });
```

LISTAGEM 22 – CRIAÇÃO DE UM CLIENTE COM *WEBTOKEN*

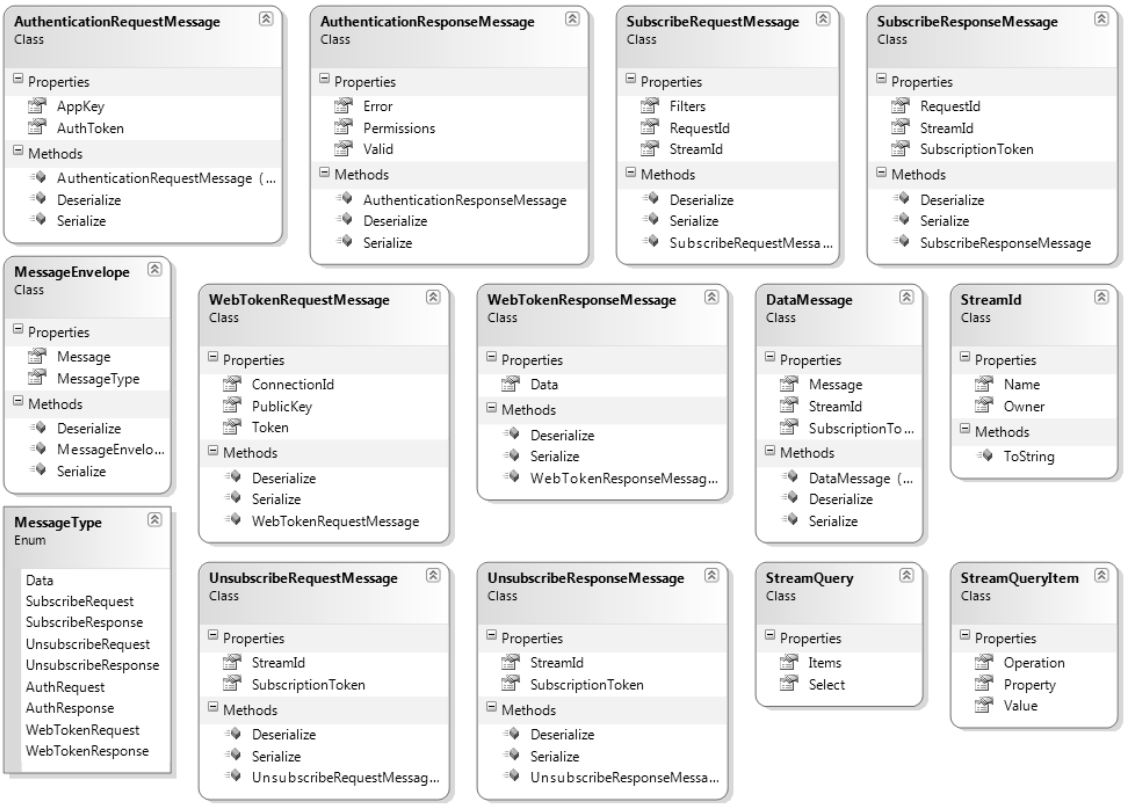

FIGURA 27 - MENSAGENS DE SISTEMA

```
public class Stream
{
    private StreamPermission _clientPermission;
    private StreamId _streamId;
    private IConnection _connection;

    private int _nextRequestId;

    private ConcurrentDictionary<int, Subscription> _pendingSubscriptions;
    private ConcurrentDictionary<string, Subscription> _activeSubscriptions;

    internal Stream(IConnection connection, StreamId id, StreamPermission permission)
    {
        this._connection = connection;
        this._streamId = id;
        this._clientPermission = permission;
        this._nextRequestId = 0;

        this._pendingSubscriptions = new ConcurrentDictionary<int, Subscription>();
        this._activeSubscriptions = new ConcurrentDictionary<string, Subscription>();
    }

    public void Publish<T>(T data)
        where T : class
    {
        if (!this._clientPermission.CanWrite())
        {
            throw new ApplicationException("No permissions to write");
        }

        string rawData;
        if (data is string)
        {
            rawData = data as string;
        }
        else
        {
            rawData = data.ToJson();
        }

         var message = new DataMessage(this._streamId, rawData);
         var envelope = new MessageEnvelope(MessageType.Data, message.Serialize());

         this._connection.Send(envelope.Serialize());
    }

    public void Subscribe(Query query, Action<string> onMessage, Action<string> onSubscribed)
    {
        this.Subscribe<string>(query, onMessage, onSubscribed);
    }
```

LISTAGEM 23 - IMPLEMENTAÇÃO DA CLASSE *STREAM* - PARTE 1

```
public void Subscribe<T>(Query query, Action<T> onMessage, Action<string> onSubscribed)
            where T : class
{
    if (!this._clientPermission.CanRead())
    {
        throw new ApplicationException("No permissions to read");
    }

    var requestId = Interlocked.Increment(ref this._nextRequestId);

    Action<string> onMessageCallback;
    if (typeof(T) == typeof(string))
    {
        onMessageCallback = onMessage as Action<string>;
    }
    else
    {
        onMessageCallback = (msg) => onMessage(msg.FromJson<T>());
    }

    this._pendingSubscriptions[requestId] = new Subscription(requestId, onMessageCallback,
onSubscribed);

    var streamQuery = query != null ? query.ToStreamQuery() : null;

    var message = new SubscribeRequestMessage(this._streamId, requestId.ToString(), streamQuery);
    var envelope = new MessageEnvelope(MessageType.SubscribeRequest, message.Serialize());

    this._connection.Send(envelope.Serialize());
}
public void Unsubscribe(string subscriptionToken, Action<string> onUnsubscribed)
{
    Subscription subscription;
    if (this._activeSubscriptions.TryGetValue(subscriptionToken, out subscription))
    {
        subscription.OnUnsubscribed = onUnsubscribed;

        var message = new UnsubscribeRequestMessage(this._streamId, subscriptionToken);
        var envelope = new MessageEnvelope(MessageType.UnsubscribeRequest, message.Serialize());

        this._connection.Send(envelope.Serialize());
    }
}
public Query CreateQuery(string fromProperty = "")
{
    return new Query(fromProperty);
}
internal void OnSubscribeResponse(SubscribeResponseMessage message)
{
    Subscription subscription;
    if (this._pendingSubscriptions.TryGetValue(int.Parse(message.RequestId), out subscription))
    {
        this._activeSubscriptions[message.SubscriptionToken] = subscription;

        subscription.OnSubscribed(message.SubscriptionToken);
    }
}
internal void OnUnsubscribeResponse(UnsubscribeResponseMessage message)
{
    Subscription subscription;
    if (this._activeSubscriptions.TryRemove(message.SubscriptionToken, out subscription))
    {
        subscription.OnUnsubscribed(message.SubscriptionToken);
    }
}
internal void OnDataMessage(DataMessage message)
{
    Subscription subscription;
    if (this._activeSubscriptions.TryGetValue(message.SubscriptionToken, out subscription))
    {
        subscription.OnMessage(message.Message);
    }
}
}
```

LISTAGEM 24 - IMPLEMENTAÇÃO DA CLASSE *STREAM* - PARTE 2

```
var httpPort = process.argv[2];

//var http = require('http');
var http = require('express');
var sockjs = require('sockjs');

var ls = require('lightstream');

ls.init(ls.configuration.Default);

// 1. sockjs server
var sockjs_opts = { sockjs_url: "http://cdn.sockjs.org/sockjs-0.3.min.js", websocket: true };

var sockjs_server = sockjs.createServer(sockjs_opts);
sockjs_server.on('connection', function (conn) {

    //'sockjs' is the provider name for SockJS connections
    ls.ConnectionManager.handle(conn, 'sockjs');
});

// 2. Http Express server
var httpServer = http.createServer();

sockjs_server.installHandlers(httpServer, { prefix: '/lightstream/websocket/sockjs' });

httpServer.listen(httpPort, '0.0.0.0');
console.log(' [*] Http WebSocket Server Listening on 0.0.0.0:' + httpPort);
```

LISTAGEM 25 - WEBSOCKET SERVER

```
var express = require('express');
var static = require('node-static');
var ls = require('lightstream');

ls.init(ls.configuration.Default);

// 2. Http Express server
var httpServer = express.createServer();

// auth handler
ls.ServiceManager.installServices(httpServer, { prefix: '/lightstream/services' }); //lightstream
services over http

httpServer.listen(8010, '0.0.0.0');
console.log(' [*] Http Web Server Listening on 0.0.0.0:8010');

//Static file server
var fileHandler = new static.Server(__dirname + '/public');

var fileServer = express.createServer(function (req, res) {
        fileHandler.serve(req, res);
});

fileServer.listen(8020, '0.0.0.0');
console.log(' [*] Static Http Server Listening on 0.0.0.0:8020');
```

LISTAGEM 26- WEB SERVER

```
defaults
    mode http

    timeout client  5s
    timeout connect 5s
    timeout server  5s

frontend all 0.0.0.0:80
    mode http
    timeout client 120s

    option forwardfor
    # Fake connection:close, required in this setup.
    option http-server-close
    option http-pretend-keepalive

    reqadd X-Forwarded-Proto:\ http

    acl is_websocket path_beg /lightstream/websocket
    acl is_webservices  path_beg /lightstream/services
    acl is_stats  path_beg /stats

    use_backend websocket if is_websocket
    use_backend webservices if is_webservices
    use_backend stats if is_stats

    default_backend static

listen ngnix_https 127.0.0.1:81
    mode http
    timeout client 120s

    option forwardfor
    # Fake connection:close, required in this setup.
    option http-server-close
    option http-pretend-keepalive

         reqadd X-Forwarded-Proto:\ https
    acl is_websocket path_beg /lightstream/websocket
    acl is_webservices  path_beg /lightstream/services
         acl is_stats  path_beg /stats

    use_backend websocket if is_websocket
    use_backend webservices if is_webservices
         use_backend stats if is_stats

    default_backend static


backend websocket
    # Load-balance according to hash created from first two
    # directories in url path. For example requests going to /1/
    # should be handled by single server (assuming resource prefix is
    # one-level deep, like "/echo").
    balance uri depth 5
    #balance roundrobin
    timeout server  120s
    server srv_websocket1 127.0.0.1:8000
    server srv_websocket2 127.0.0.1:8001
    #server srv_websocket2 127.0.0.1:8002
    #server srv_websocket2 127.0.0.1:8003

backend webservices
    # Load-balance according to hash created from first two
    # directories in url path. For example requests going to /1/
    # should be handled by single server (assuming resource prefix is
    # one-level deep, like "/echo").
    balance roundrobin
    timeout server  120s
    server srv_webservices1 127.0.0.1:8010
    #server srv_webservices2 127.0.0.1:8011

backend static
    balance roundrobin
    server srv_static 127.0.0.1:8020

backend stats
    stats uri /stats
    stats enable
```

LISTAGEM 27 - CONFIGURAÇÃO DO BALANCEADOR *HAPROXY*

```
worker_processes  1;

events {
    worker_connections  1024;
}

http {
    upstream haproxy {
        server 127.0.0.1:81;
    }

    server {
        listen       443;
        server_name  my.host.name default_server;
        ssl                  on;
        ssl_certificate      /home/dev/Development/lightstream/certificate.pem;
        ssl_certificate_key  /home/dev/Development/lightstream/privatekey.pem;
        ssl_session_timeout  5m;

        ssl_protocols  SSLv2 SSLv3 TLSv1;
        ssl_ciphers  HIGH:!aNULL:!MD5;
        ssl_prefer_server_ciphers   on;

        location / {
            proxy_pass http://haproxy;

            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

            proxy_set_header X-Forwarded-Proto https;
        }
    }
}
```

LISTAGEM 28 - CONFIGURAÇÃO DO COMPONENTE *NGINX*

```
public class BackendWorker
{
    public static void Main(string[] args)
    {
        Console.Title = string.Format("LightStream Backend Worker {0}", args[0]);

        RequestReplyManager.Instance.Init(RequestReplyMode.Server);

        //register services
        RequestReplyManager.Instance.RegisterService(new ContractService<AuthService,
IAuthService>());
        RequestReplyManager.Instance.RegisterService(new ContractService<DbService, IDbService>());

        Console.ReadLine();
    }
}
```

LISTAGEM 29 - INICIALIZAÇÃO DE UM *WORKER*

```
public class BackendBroker
{
    public static void Main(string[] args)
    {
        Console.Title = "LightStream Backend Broker";

        BrokerManager.Instance.Init(BrokerMode.RequestReply);
        Console.ReadLine();
    }
}
```

LISTAGEM 30 - INICIALIZAÇÃO DO *BROKER*

```
internal class RequestReplyBroker : BaseZmqComponent, IDisposable
{
    private static ILog Log = LogManager.GetLogger(MethodBase.GetCurrentMethod().DeclaringType);

    private volatile bool _running;

    public RequestReplyBroker(ZMQ.Context zmqContext)
        : base(zmqContext)
    {
        this._running = false;
    }

    protected override void OnInit()
    {
        Task.Factory.StartNew(() => this.BrokerTask(), TaskCreationOptions.LongRunning);
    }

    private void BrokerTask()
    {
        this._running = true;

        var frontendAddress = base._configuration.RequestReply_Frontend_BindAddress;
        var backendAddress = base._configuration.RequestReply_Backend_BindAddress;

        while (this._running)
        {
            //frontend
            using (var zmqFrontendSocket = base._zmqContext.Socket(ZMQ.SocketType.ROUTER))
            {
                //backend
                using (var zmqBackendSocket = base._zmqContext.Socket(ZMQ.SocketType.DEALER))
                {
                    zmqFrontendSocket.Bind(frontendAddress);
                    Log.InfoFormat("RequestReplyBroker frontend bound on {0}", frontendAddress);

                    zmqBackendSocket.Bind(backendAddress);
                    Log.InfoFormat("RequestReplyBroker backend bound on {0}", backendAddress);

                    ZMQ.Socket.Device.Queue(zmqFrontendSocket, zmqBackendSocket);
                }
            }
        }
    }

    public void Dispose()
    {
        this._running = false;
    }
}
```

LISTAGEM 31 - IMPLEMENTAÇÃO DO *BROKER*

```
var ls = require('lightstream');
var events = require('events');
var zmq = require('zmq');

var AsyncClient = function () {

    var _this = this;
    var _serverConfig = null;

    var _zmqSocket = zmq.createSocket('dealer');

    var _requestTimeout = 30 * 1000; //30 seconds

    var _nextRequestId = 0;
    var _sentRequests = {};

    //serverConfig should be { host: 'x.x.x.x', port: 'x', protocol: 'tcp' }
    this.init = function (serverConfig) {

        _serverConfig = serverConfig;

        ls.Logger.debug('AsyncClient connecting to ' + _getConfigurationAddress());
        _zmqSocket.connect(_getConfigurationAddress());

        _zmqSocket.on('message', function (id, type, result) {

            //id, type, and result are Buffer objects

            _onReply({
                id: parseInt(id.toString('utf8')), //could be optimized using Buffer readInt32
                type: type.readInt8(0),
                result: result.toString('utf8')
            });
        });
        _zmqSocket.on('error', _onError);

        ls.Logger.debug('AsyncClient connected to ' + _getConfigurationAddress());
    };

    var _onError = function (err) {

        ls.Logger.error('AsyncClient error: ' + err);
    };

    var _onReply = function (reply) {

        var request = _sentRequests[reply.id];
        if (request !== undefined) {

            if(request.timeout){

                clearTimeout(request.timeout);
                request.timeout = null;
            }

            _sentRequests[reply.id] = null;
            delete _sentRequests[reply.id];

            if (reply.type != 1) {

                request.onReply(null, reply.result);
            }
            else {

                request.onReply(_parseResult(reply.result));
            }
        }
    };
```

LISTAGEM 32 - CLIENTE EM *NODEJS* PARA A *FRAMEWORK* DE *0MQ* – PARTE 1

```
    var _getConfigurationAddress = function () {

        return _serverConfig.protocol + '://' + _serverConfig.host + ':' + _serverConfig.port;
    };

    var _parseResult = function(result){

        if(result == null){

            return null;
        }

        return JSON.parse(result);
    };

    this.requestData = function (req, callback) {

        var request = {
            original: req,
            id: ++_nextRequestId,
            onReply: callback,
        };

        _sentRequests[request.id] = request;

        request.timeout = setTimeout(function(){

            if(_sentRequests[request.id] !== undefined){
                //still waiting for an answer
                //fake reply
                _onReply({ id: request.id, type: 3, result: 'operation timed out' });
            }
        }, _requestTimeout);

        _zmqSocket.send(request.id, zmq.ZMQ_SNDMORE);
        _zmqSocket.send(req.serviceName, zmq.ZMQ_SNDMORE);
        _zmqSocket.send(req.body);
    };

};

module.exports = AsyncClient;
```

LISTAGEM 33 – CLIENTE EM *NODEJS* PARA A *FRAMEWORK* DE *0MQ* – PARTE 2

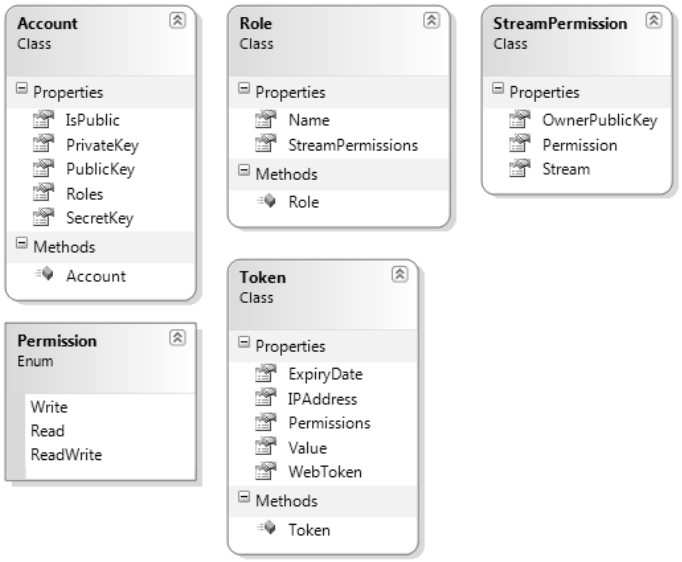

FIGURA 28 - MODELO DE DADOS DO SISTEMA

```
<?xml version="1.0" encoding="utf-8" ?>
<ls>
  <accounts>
    <account key="svc1">
      <privateKey>svc1_private</privateKey>
      <secretKey>svc1_secret</secretKey>
      <isPublic>true</isPublic>
      <roles>
        <role name="public">
          <streamPermission>
            <owner>svc1</owner>
            <name>weather</name>
            <permission>2</permission>
          </streamPermission>
        </role>
        <role name="publisher">
          <streamPermission>
            <owner>svc1</owner>
            <name>weather</name>
            <permission>3</permission>
          </streamPermission>
        </role>
      </roles>
    </account>
    <account key="acc1">
      <privateKey>acc1_private</privateKey>
      <secretKey>acc1_secret</secretKey>
      <isPublic>false</isPublic>
      <roles>
        <role name="public">
          <streamPermission>
            <owner>acc1</owner>
            <name>s1</name>
            <permission>2</permission>
          </streamPermission>
          <streamPermission>
            <owner>acc1</owner>
            <name>s2</name>
            <permission>3</permission>
          </streamPermission>
        </role>
        <role name="publisher">
          <streamPermission>
            <owner>acc1</owner>
            <name>s1</name>
            <permission>3</permission>
          </streamPermission>
        </role>
        <role name="weather-reader">
          <streamPermission>
            <owner>svc1</owner>
            <name>weather</name>
            <permission>2</permission>
          </streamPermission>
        </role>
      </roles>
    </account>
  </accounts>
</ls>
```

LISTAGEM 34 - FICHEIRO XML COM OS DADOS DE TESTE