APPSOCKETS

Tiago Marques Soares Lima

Número: 50046

Instituto Superior de Tecnologias Avançadas do Porto

R. Dr. Lopo de Carvalho 4350-162 Porto

Portugal

Porto, 15 de junho de 2023

APPSOCKETS

Tiago Marques Soares Lima
Número: 50046

Trabalho realizado no âmbito da unidade curricular de Projeto, da Licenciatura em Engenharia Informática, do Instituto Superior de Tecnologias Avançadas do Porto, orientado pelo Dr. João Rebelo.

Instituto Superior de Tecnologias Avançadas do Porto
Portugal

Porto, 15 de junho de 2023

"A vontade de vencer, o desejo de ter sucesso, o impulso de alcançar todo o seu potencial… essas são as chaves que abrirão a porta para a excelência pessoal."

"The will to win, the desire to succeed, the urge to reach your full potential… these are the keys that will unlock the door to personal excellence."

Confúcio

RESUMO

Neste projeto foi desenvolvida uma aplicação para comunicação em *soft real-time*, utilizando o sistema de comunicação *Publish/Subscribe*, com o propósito de substituir um sistema previamente em utilização. Utilizando as tecnologias *gossip*, *hash ring* e *gRPC*, foi criada uma aplicação distribuída e horizontalmente escalável, que é capaz de substituir o sistema anterior na empresa NAPPS, enquanto mantém todas as suas funcionalidades.

Este projeto foi desenvolvido na modalidade de projeto inovador na empresa NAPPS, este enquadra-se na área de especialização de Desenvolvimento/programação (Sistemas de informação, *Web* e Móvel). Adicionalmente, este projeto permitiu a utilização dos conhecimentos técnicos e teóricos adquiridos ao longo do curso, no desenvolvimento de um projeto cuja aplicação prática permite validar todos os ensinamentos que foram aprendidos, durante a Licenciatura em Engenharia Informática, concluindo assim esta etapa tão importante do meu percurso académico.

Palavras-chave: *Soft Real-Time, Publicar/Subscrever, WebSockets, Message Broker*.

ABSTRACT

In this project, an application for soft real-time communication was developed using the Publish/Subscribe communication system, with the purpose of replacing a previously used system. Using gossip, hash ring, and gRPC technologies, a horizontally scalable distributed application was created, which is capable of replacing the previous system at NAPPS while maintaining all of its functionalities.

This project was developed as an innovative project within the specialization area of Development/Programming (Information Systems, Web, and Mobile) at NAPPS. Additionally, this project allowed for the application of the technical and theoretical knowledge acquired throughout the course, in the development of a project whose practical application validates all of the teachings that were learned during the Bachelor's degree in Computer Engineering, thus completing this important stage of my academic journey.

Keywords: *Soft Real-Time, Publish/Subscribe, WebSockets, Message Broker.*

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer à minha família pelo apoio incansável durante todo o meu percurso académico. Por diversas vezes contribuíram positivamente para o meu desempenho interagindo e questionando as opções tomadas.

Agradeço ao Professor Engenheiro João Rebelo, o meu orientador de projeto e coordenador do projeto final pela sua recetividade, apoio, orientação e gentileza que sempre demonstrou ao longo de toda a licenciatura e, em especial no desenvolvimento deste projeto.

Da mesma forma, gostaria também de agradecer a todos os docentes e não docentes que fizeram parte do meu percurso na licenciatura do ISTEC Porto.

Um especial agradecimento aos colegas de curso Gonçalo Nogueira que me acompanhou em todo o meu percurso académico.

Não esquecendo os meus colegas na NAPPS por todo o companheirismo que tanto me fizeram evoluir a nível pessoal e profissional

.

ÍNDICE GERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

LISTA DE SIGLAS E ACRÓNIMOS

AMQP – Advanced Messaging Queuing Protocol

APNS – Apple Push Notification Service

AWS – Amazon Web Services

B2B – Business to Business

DHT – Distributed Hash Table

ECS – Elastic Container Service

FCM – Firebase Cloud Messaging

HTTP – Hypertext Transfer Protocol

IETF – Internet Engineering Task Force

JSON – JavaScript Object Notation

MQTT – Message Queuing Telemetry Transport

NATS – Neural Autonomic Transport System

OTP – Open Telecom Platform

SaaS – Software as a Service

SSE – Server-Sent Events

STOMP – Simple/Streaming Text Oriented Messaging Protocol

GLOSSÁRIO

Backoffice – O *backoffice* é a parte administrativa e de gestão de uma empresa que acontece nos bastidores e é responsável por fornecer suporte e recursos para as atividades principais da empresa, garantindo que as operações funcionem sem problemas e que os objetivos estratégicos sejam alcançados.

Cluster – Um *cluster* é um grupo de computadores que trabalham juntos para realizar uma tarefa ou fornecer um serviço específico. Esses computadores são conectados por meio de uma rede e são geridos como se fossem um único sistema, o que permite uma maior capacidade de processamento e de armazenamento de dados. Adicionalmente, *clusters* também podem ser usados para fornecer serviços de alta disponibilidade, balanceamento de carga e tolerância a falhas.

Dashboard – Um *dashboard* é uma interface visual que apresenta informações importantes e fáceis de entender sobre um determinado conjunto de dados ou processo. Este ajuda a monitorizar o desempenho de diferentes áreas ou processos de negócio de forma rápida e eficiente, permitindo que os utilizadores tomem decisões informadas com base nos dados apresentados.

Heartbeat – O termo *heartbeat* é utilizado para descrever um mecanismo de monitorização que permite que um sistema verifique continuamente o estado de outro sistema ou componente, através da troca regular de mensagens ou sinais entre os sistemas. Se o sistema não receber um sinal de "batimento cardíaco" dentro de um intervalo de tempo definido, pode ser considerado como inoperacional ou com falha, e medidas apropriadas podem ser tomadas para lidar com a situação. O *heartbeat* é frequentemente utilizado em ambiente de alta disponibilidade ou em sistemas distribuídos para garantir a confiabilidade e a disponibilidade dos sistemas.

Metadata – Metadata é um termo utilizado para descrever dados que fornecem informações sobre outros dados. Em outras palavras, são dados que descrevem características, propriedades ou atributos de um determinado conjunto de dados, como informações sobre a sua origem, formato, data de criação, entre outros.

peer-to-peer – *Peer-to-peer* (P2P) é um modelo de comunicação descentralizado em que os dispositivos ligados à rede, chamados de *peers*, comunicam diretamente entre si, sem a necessidade de um servidor central. Cada *peer* na rede atua como um cliente e um servidor, tornando a rede mais resiliente e independente, além de permitir uma maior escalabilidade.

Prometheus – *Prometheus* é uma ferramenta de monitoramento de código aberto utilizada para coletar e armazenar métricas de sistemas e serviços. Este é muito utilizado em ambientes de *cloud* e *containers*, sendo capaz de monitorar aplicativos distribuídos e escaláveis.

Pub/Sub – *Pub/Sub*, ou *publish/subscribe*, é um modelo de comunicação em que os participantes se comunicam por meio de mensagens transmitidas por um intermediário (*broker*). Neste modelo, os participantes são divididos em duas categorias: *publishers* e *subscribers*. *Publishers* são responsáveis por enviar mensagens para o intermediário, enquanto *subscribers* se inscrevem em determinados tópicos de interesse. Quando um *publisher* envia uma mensagem, o intermediário envia a mensagem para todos os *subscribers* que estão inscritos no tópico relevante. Este modelo é muito utilizado em sistemas distribuídos para comunicação assíncrona e escalável entre diferentes partes do sistema, permitindo a comunicação eficiente entre muitos participantes sem a necessidade de cada participante saber com quem está a comunicar. O *Pub/Sub* é amplamente utilizado em aplicações de *IoT* (Internet das Coisas), sistemas de mensagens e sistemas de eventos, permitindo que os participantes se comuniquem de forma eficiente e escalável. Ao longo deste documento o nome *PubSub* ou *Pub/Sub* será utilizado, tendo ambos o mesmo significado.

Rate Limit – O *rate limit* é uma forma de controlar a quantidade de pedidos que uma aplicação pode fazer num determinado período. Por exemplo, se um serviço tem um *rate limit* de 100 pedidos por minuto, isso significa que um utilizador só poderá realizar 100 pedidos nesse período. Caso este tente fazer mais que esses, estes serão impedidos.

Redis – *Redis* é uma base de dados em memória de código aberto, utilizado para armazenar dados chave-valor. Este é rápido e escalável, adequado para aplicações que necessitam de alta velocidade e baixa latência, como aplicações *web*, sistemas de mensagens e jogos online (*Redis Labs*, s.d.).

RPC – *Remote Procedure Call* (Chamada de Procedimento Remoto) é um protocolo de comunicação entre diferentes sistemas de computação. É usado para permitir que um programa em um dispositivo possa chamar uma função ou método em outro dispositivo através da rede, como se essa função estivesse a ser executada localmente. O *RPC* é uma tecnologia muito utilizada em sistemas distribuídos e é suportado por várias linguagens de programação e plataformas. Este permite que diferentes sistemas operacionais e ambientes de rede se comuniquem de forma transparente, tornando a programação e a integração de sistemas muito mais fáceis.

Snapshot – Um snapshot é uma imagem instantânea ou cópia exata de um estado de um sistema ou conjunto de dados em um determinado momento. Em outras palavras, é uma "fotografia" do estado atual de um sistema ou conjunto de dados que pode ser armazenada e utilizada posteriormente para referência, restaurar um sistema ou conjunto de dados em um estado anterior.

Tenant – Um *tenant* refere-se a uma entidade, como uma organização ou utilizador, que possui acesso e controle exclusivo sobre um conjunto de recursos dentro de um ambiente compartilhado. O conceito de *tenant* é amplamente utilizado em serviços de computação em nuvem, como Software como Serviço (*SaaS*), Plataforma como Serviço (*PaaS*) e Infraestrutura como Serviço (*IaaS*), para garantir que várias entidades possam compartilhar recursos de forma segura e eficiente.

Timestamp – Uma *timestamp* é uma informação que indica o momento em que ocorreu um evento, como uma transação, uma alteração num documento ou a criação de um ficheiro. Geralmente, é representada por uma sequência de caracteres que inclui a data e a hora em que o evento ocorreu, seguindo um formato predefinido. A *timestamp* é útil para fins de rastreabilidade e para garantir que as informações são sincronizadas e organizadas de forma cronológica. Este podem ser representados em vários formatos, como *ISO 8601* e *Unix*.

PARTE I – RELATÓRIO DE PROJETO

1 INTRODUÇÃO

Neste projeto é explicado a criação de infraestrutura para o envio de informação em *soft real-time* entre clientes e servidores, e ao mesmo tempo substituir um sistema com objetivos similares, mantendo o máximo de compatibilidade possível de forma a facilitar a migração para o novo sistema.

1.1 Enquadramento e Motivação

Este projeto foi desenvolvido na empresa NAPPS, com o objetivo de resolver um problema existente, e permitir que novas funcionalidades sejam desenvolvidas com o resultado do desenvolvimento deste projeto. Assim sendo, será explicado o contexto em que a empresa trabalha e a motivação para o desenvolvimento deste projeto.

1.1.1 Enquadramento de empresa

NAPPS é uma empresa *startup SaaS* (software como serviços) *B2B* (de empresa para empresa) que desenvolve aplicações personalizáveis para dispositivos móveis, nomeadamente para *Android* e *IOS*, que são vendidas como um serviço a lojas de *e-commerce*. Atualmente as plataformas suportadas são *Shopify* e *WooCommerce*, no entanto, integrações com outros *plugins*/*apps* nas plataformas também são alvo para integração nas aplicações. Neste contexto de aplicações *e-commerce*, surgiu a necessidade de comunicar com as aplicações móveis de forma quase instantânea sempre que a aplicação estiver em execução.

Para esse propósito, foi necessário criar infraestrutura para o envio de informação em tempo real de forma bidirecional entre clientes e servidores. A infraestrutura não será exclusiva às aplicações móveis, tornando possível a sua utilização por outros serviços que possam necessitar de comunicação em tempo real.

1.1.2 Motivação

Este projeto consiste numa aplicação *Publish/Subscribe* com suporte para múltiplos *tenants*, onde existem elementos que subscrevem a um tópico (*Subscribe*) e recebem todas mensagens ou eventos publicados neste mesmo tópico (*Publish*).

O sistema criado ao longo deste projeto, tem como propósito substituir o sistema anterior enquanto mantém todas as suas funcionalidades, adiciona novas funcionalidades e facilita a sua utilização.

As motivações para o desenvolvimento deste novo sistema foram baseadas em alguns pontos principais, sendo estes:

- O sistema a ser substituído não ser horizontalmente escalável;
- A não existência de ferramentas de monitorização e deteção de erros;
- Arquitetura não preparada para novas funcionalidades;
- Falta de testes no projeto.

O sistema a ser substituído, foi desenvolvido de forma rápida, e durante o seu desenvolvimento não existia a necessidade de que este fosse horizontalmente escalável, e a adaptação seria complicada exigindo modificar grande parte do seu funcionamento. Inicialmente, este sistema foi projetado para ser utilizado maioritariamente por *dashboards* e *backoffices* como subscritores enquanto alguns eventos eram emitidos por outros servidores.

No entanto, novas funcionalidades a serem planeadas necessitam que a utilização deste sistema seja ampliada para a aplicações móveis, onde existe um valor muito mais elevado de conexões a serem realizadas, de forma a quantificar a diferença de conexões esperadas, no sistema a ser substituído era somente esperado ter no máximo 50 conexões diárias, um valor muito baixo, enquanto o valor esperado para os utilizadores atuais é de aproximadamente 9000 conexões, um valor muito superior. Adicionalmente, sempre que se adquire um novo cliente, é esperado que este valor suba entre algumas centenas a alguns milhares (aproximadamente entre 600 e 2000), sendo que o novo sistema tem de ser capaz de suportar este aumento de utilizadores.

Outro ponto relacionado com a necessidade de escalar horizontalmente, consiste em permitir que o sistema seja tolerante a falhas, algo que não é possível se somente um servidor puder ser executado ao mesmo tempo. O motivo pelo qual o sistema não é horizontalmente escalável, deve-se ao funcionamento geral de um sistema de comunicação *PubSub*, onde independentemente do servidor a que o cliente está conectado, este tem de receber eventos que podem ser enviados noutros servidores.

A inexistência de ferramentas de monitorização e de deteção erros dificulta a manutenção do sistema, no entanto, sendo que nenhuma funcionalidade em que este era utilizada era considerada crítica, não houve nenhum incentivo para desenvolver estas, no entanto, sendo que este sistema passou a ser utilizado por clientes finais, é importante ser capaz de identificar os erros o mais rápido possível, assim como ser capaz de monitorizar a sua utilização de forma a planear o melhor possível o escalamento automático.

1.1.3 Problemas

No sistema a ser substituído, além dos pontos acima mencionados, existem outros problemas ou inconveniências identificadas durante a utilização deste. Portanto, analisando o funcionamento do projeto atual temos a seguinte informação.

Por cada *tenant* é criado um objeto *App* que contém um nome que é utilizado como identificador único, estes têm de ser explicitamente criados previamente antes da sua utilização. Assim que criado, a primeira conexão a um destes *tenants* um *Hub* é criado para gerir todas conexões e *channels* (tópicos) deste *tenant*, tornando assim o *Hub* como o elemento que agrupa conexões e tópicos de um *tenant*. Resumindo, temos a hierarquia de *Hub* gere várias conexões e vários *channels*, sendo o objeto *App* apenas uma forma de criar um *tenant*.

Outros problemas encontrados atualmente são a necessidade de criar explicitamente cada tópico individualmente. Sendo um sistema separado, manter a sincronização de quais *tenants* estão ativos acaba por ser um problema na presença de falhas, quanto à necessidade de criar tópicos explicitamente dificulta em casos onde o número de tópicos é dinâmico e a necessidade de criação aumenta a complexidade de gestão. Por exemplo, caso seja necessário um tópico para cada produto, seria necessário criar um número elevado de tópicos, e para piorar a situação, muitas vezes os produtos apresentam variações dos mesmos (Exemplo: Camisola versão azul, versão vermelha e versão verde), sendo necessário criar tópicos para cada variação. Por fim quando um produto fosse apagado seria necessário voltar a apagar os tópicos por cada variante.

De forma a evitar este problema é necessário que a criação de cada tópico seja de forma dinâmica, evitando explicitamente a criação deste permitindo ter configurações num grupo de tópicos e apenas opcionalmente para cada tópico explicitamente, quanto à criação de *tenants* estes também podem ser opcionalmente dinâmicos com configurações por defeito de forma a evitar acessos não autorizados.

Outros problemas menores passam por não permitir conexões não autenticadas, sendo que é sempre necessário um *token* de acesso (objeto *JSON* compacto assinado) para iniciar uma conexão, mas em alguns cenários esta exigência dificulta o processo. Voltando ao exemplo de produtos, caso uma aplicação cliente necessite ser notificado das alterações de stock de um produto e o utilizador não tenha uma conta, este tem de pedir a um servidor um *token* de acesso como utilizador anónimo, e só após poderá se conectar e subscrever ao tópico de stock do produto.

Adicionalmente, atualmente para autenticar ou mudar o *token* de acesso é necessário desconectar e reconectar com o novo *token* de acesso criando um período onde atualizações não são recebidas.

Neste sistema, a capacidade de rastreamento de presenças apresenta algumas falhas quando o servidor é desligado devido a uma falha, este não é capaz de corrigir o estado das presenças armazenadas, sendo necessário intervenção manual para corrigir o problema.

Tendo brevemente apresentado o projeto atualmente em produção e os seus problemas, estes são os pontos a ter em consideração no planeamento do novo projeto:

- Restringir o acesso a tópicos de acordo com as configurações ou autorizações;
- Configurações dos tópicos devem permitir configurar:
    - O armazenamento das mensagens enviadas;
    - O rastreamento da presença no tópico;
    - Definir um tópico como público ou privado, permitindo que este seja acedido por conexões anônimas caso seja público;
    - Permitir ou não anónimos mesmo que o tópico seja público.
- Criação dinâmica de tópicos;
- Agrupamento de tópicos, tendo uma configuração aplicada a todos os tópicos presentes no grupo, e que a sua alteração seja refletida nos mesmos.
- Permitir que o cliente esteja autenticado ou não, permitindo autorizações extras caso este esteja autenticado;
- Suporte para múltiplos *tenants*, possivelmente a criação destes de forma dinâmica;
- Rastrear a presença dos utilizadores em cada tópico;
- De preferência ser capaz de enviar mensagens na mesma ordem que a infraestrutura recebeu;
- Ser capaz de escalar horizontalmente;
- Permitir comunicação com serviços internos via *NATS*;
- Envio de mensagens autorizadas para uma *stream* no *NATS*.

1.2 Objetivos

Tendo sido explicado os problemas que levaram a desenvolver um novo sistema, é necessário definir os objetivos a serem cumpridos pelo novo sistema, lembrando que o novo sistema vai substituir um existente, é necessário que este seja capaz de suportar os casos de uso atuais, assim como criar o máximo de compatibilidade possível. Portanto, sendo os requerimentos do novo sistema similares com o anterior em produção, é necessário analisar como o atual funciona e ver que problemas apresenta. Os principais pontos a ter em conta no projeto são:

- Comunicação bidirecional entre cliente e servidor através *WebSockets*;
- Comunicação utilizando *Pub/Sub* (publicar e subscrever) em tópicos (nomeados de *channels*);
- Restringir o acesso a tópicos de acordo com as autorizações;
- Suporte para múltiplos *tenants*, existindo configurações por cada *tenant*;
- Criação explícita de tópicos e com configurações por cada;
- Rastreamento da presença dos clientes em cada tópico;
- Armazenamento das mensagens enviadas em cada tópico.

1.3 Estrutura do Relatório

O presente relatório está organizado em 6 capítulos onde é abordado todo o processo de planeamento e de desenvolvimento do projeto.

Capítulo 1 – Introdução ao projeto, contextualização do tema e motivação para o desenvolvimento da plataforma;

Capítulo 2 – Estado da Arte, explicação de métodos de comunicação em aplicações *web*, e apresentação de tecnologias existentes e serviços;

Capítulo 3 – Metodologia e planeamento do projeto;

Capítulo 4 – Desenvolvimento do projeto, explicação de partes principais do projeto, escolha de tecnologias a ser utilizados, funcionamento de componentes do projeto e ferramentas criadas;

Capítulo 5 – Discussão de resultados obtidos;

Capítulo 6 – Conclusão do desenvolvimento do projeto e apresentação das reflexões.

2 ESTADO DA ARTE

Nesta parte vai ser mencionado técnicas utilizadas para enviar informação em tempo real tem evoluído, protocolos que tenham vindo a ser criados e qual foi o escolhido para este projeto. Adicionalmente, são selecionados projetos de código aberto e serviços comerciais que podem potencialmente ser utilizados de forma a tentar a cumprir os objetivos deste projeto.

2.1 Evolução de comunicação em tempo real

Comunicação em tempo real não é um tópico novo e está presente em várias aplicações, principalmente em aplicações de mensagens, no entanto, em aplicações *web* nem sempre existiu uma forma de criar uma ligação bidirecional entre cliente e servidor. Sendo necessário que aplicações *web* tenham a possibilidade de realizar uma comunicação com os servidores primeiro é necessário conhecer as opções existentes e como estas foram evoluindo.

Inicialmente, em aplicações *Web* não existia a possibilidade de criar ligações bidirecionais com servidores utilizando as *APIs* fornecidas pelos *browsers*, de forma a resolver esta limitação, em 2011 um novo protocolo foi padronizado *Fette* e *Melnikov (2011)* como *RFC 6455*, este protocolo ficou conhecido como *WebSockets* e é atualmente a forma padrão de comunicação bidirecional com servidores em aplicações *Web*. Em outras aplicações não *web*, estas limitações não existiam, portanto cabia a cada desenvolvedor utilizar a sua implementação ou reutilizar uma existente.

Antes da criação do protocolo *WebSockets*, a técnica *long polling* era uma forma comum de simular comunicação bidirecional, assim como mencionado pela *Internet Engineering Task Force* (2011), "web applications that need bidirectional communication between a client and a server [...] has required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls" (*The WebSocket Protocol)* (capítulo 1.1, 1º parágrafo), visto que os pedidos *HTTP* funcionam como *request-reply* (pergunta-resposta) de forma unidirecional (cliente para servidor), não existia forma de um servidor notificar o utilizador que um evento tenha acontecido no momento, ou seja, uma aplicação cliente teria que periodicamente realizar um pedido *HTTP* ao servidor de forma a verificar que novos eventos tenham ocorrido. Tendo como exemplo uma aplicação de chat, onde existem largos períodos sem atividade, é possível que grande parte destes pedidos não tenham informação nova desperdiçando recursos, ou então, caso o período entre pedidos seja longo é possível que demore demasiado tempo para receber nova informação. Utilizando o mesmo exemplo, numa

conversa entre duas pessoas e com intervalo entre pedidos de 5 segundos, uma mensagem pode demorar até esse mesmo intervalo só para ser recebida pela outra pessoa.

De forma a evitar a quantidade de pedidos realizados e a reduzir o tempo que demora a receber informação, o servidor artificialmente demora mais tempo para enviar uma resposta, esperando que exista nova informação ou que tempo limite de conexão tenha sido atingido. Esta parte é a origem do nome *Long* na técnica *Polling*. Desta forma, o tempo de atraso a receber a mensagem seria no máximo o tempo de receber a última resposta mais o tempo de iniciar um novo pedido, algo que poderia demorar segundos que passou para milissegundos, além de reduzir consideravelmente a quantidade de pedidos a serem feitos.

Com a criação do protocolo *WebSockets*, a técnica *long polling* deixou de ser usada em novos projetos e serve como alternativa caso uma conexão *WebSocket* não seja possível. No entanto, embora *WebSockets* seja o padrão existem outras opções para permitir que o servidor comunique com o cliente tais como: *Server-Sent Events, Web Push* e *HTTP Streaming*.

Server-Sent Events ou *SSE*, assim como definido por *Roome* e *Yang* (2020) no *RFC 8895*, permite ao servidor enviar informação para o cliente por *HTTP* pela duração da conexão, ao contrário do protocolo *WebSockets*, este somente permite uma comunicação unidirecional de servidor para cliente, e não suporta o envio de informação em formato binário. Visto que este protocolo somente permite o envio de informação de servidor para cliente, pedidos adicionais têm de ser feitos caso o cliente precise de enviar informação para o servidor.

O *Web Push*, conforme definido por *Thomson* e *Damaggio* (2016) no *RFC 8030*, torna possível o envio de informação para o cliente, no entanto, este costuma ser utilizado para o envio de notificações e não de dados em geral, sendo as mensagens enviadas acompanhadas por título, conteúdo, e exigem que os clientes aceitem uma permissão para receber esta informação. Embora esta opção não seja adequada para envio de informação em tempo real, esta pode servir como alternativa para o envio de informação quando é necessário que a informação seja recebida mesmo que o cliente não esteja ligado a um dos servidores.

O *HTTP Streaming* é relativamente similar ao *Server-Sent Events*, este também permite o envio de informação para o cliente de forma unidirecional. Este funciona enviando informação sem tamanho definido, pondo a aplicação cliente constantemente à espera dos próximos dados até a conclusão do pedido HTTP.

Tendo revisto os meios de comunicação disponíveis, o protocolo *WebSockets* aparenta ser a melhor opção, principalmente por ser o protocolo padrão na indústria e pela sua capacidade de comunicação bidirecional, no entanto, outros protocolos poderão ser implementados quando a bidirecionalidade não for necessária, preferencialmente utilizando *Server-Sent Events*.

Escolhido o protocolo *WebSockets*, convém conhecer o seu funcionamento, assim como mencionado previamente, este permite comunicação bidirecional entre cliente e servidor, esta é estabelecida utilizando *HTTP* inicialmente que após um *handshake* é estabelecida. Este protocolo é fundamentalmente dividido em duas partes: o *handshake* e a transferência de dados.

No *handshake*, o pedido é realizado pelo cliente enviando a intenção de transformar a conexão unidirecional em uma bidirecional (com o nome de *Upgrade* no protocolo) ao qual o servidor deverá responder que está a trocar o protocolo, após esta parte a conexão é considerada estabelecida. Na transferência de dados, é usado o conceito de mensagens, sendo cada composta por um ou mais *frames*. Cada *frame* tem um tipo associado, tendo cada *frame* pertencente à mesma mensagem o mesmo tipo. De forma geral, existem 3 tipos de dados, sendo textual, binário e de controle. No tipo textual a informação é interpretada como *UTF-8* enquanto no tipo binário a interpretação é deixada à responsabilidade da aplicação, para o controle, que não tem como objetivo transferir dados da aplicação, são usados como sinalização da conexão, como por exemplo *PING*, *PONG* e *CLOSE*. Estes últimos *PING* e *PONG* tem como propósito verificar se a conexão ainda se encontra ativa, principalmente quando a aplicação envolve pouco tráfego. O transporte de mensagens numa conexão com protocolo *WebSocket* é similar a uma conexão *TCP*, este apenas junta um mecanismo de *framing* que reduz essa responsabilidade na aplicação, quanto ao formato dos *frames* não será mencionado tendo em conta que não faz parte do objetivo deste documento.

2.2 Soluções existentes

Tendo em conta o protocolo escolhido e os pontos a serem considerados, foi realizada pesquisa sobre soluções já existentes que suportam os pontos definidos e ao mesmo tempo tentar perceber de que forma estas soluções estruturam as soluções e o que estas permitem. Estas soluções incluem tanto projetos e bibliotecas de código aberto como serviços, as principais soluções encontradas são as seguintes.

- Código aberto*:*
    - *Centrifugo;*
    - *Mercure;*
    - *Phoenix;*
    - *VerneMQ;*
    - *Emitter;*
    - *HiveMQ;*
    - *EMQX;*
    - *SocketCluster;*
    - *Soketi;*
    - *Signal-R;*
- Serviços:
    - *Ably;*
    - *PubNub;*
    - *Pusher;*
    - Fanout.

Deste conjunto existem algumas opções que funcionam como um *broker* de mensagens, utilizando protocolos já existentes como *MQTT*, deste conjunto temos os seguintes:

Centrifugo (s.d.) é uma aplicação que serve como um b*roker* de mensagens. Esta aplicação suporta a distribuição de mensagens com os protocolos *WebSockets* e *gRPC* e com o envio de mensagens por pedido *HTTP*. É possível de escalar horizontalmente utilizando através da utilização de um dos *engines* suportados pela aplicação. De forma a permitir que os clientes possam enviar mensagens, estes precisam de uma autorização extra criada por servidores, ou que estes sirvam como intermediários para o envio de mensagens.

O *Mercure* (s.d.) é um *broker* de mensagens, com distribuição de mensagens utilizando *SSE* (unidirecional) e com o envio de mensagens por pedido *HTTP*. A possibilidade de escalar horizontalmente exige o uso de um serviço oferecido pelos desenvolvedores para a gestão da infraestrutura. Sendo o protocolo de comunicação principal *SSE* este remove a possibilidade de comunicação bidirecional, para que os clientes possam enviar mensagens, precisam de uma autorização extra criada por servidores, ou que estes sirvam como intermediários para o envio de mensagens.

O *Phoenix Framework* (*s.d.*) é um *framework* para a linguagem de programação *Elixir*, com suporte para comunicação em tempo real e escalável horizontalmente. Sendo desenvolvido em *Elixir* permite a utilização da *Erlang VM*, desenvolvida com suporte para tolerância a falhas e maioritariamente utilizada em sistemas de telecomunicações tornando uma excelente escolha. O protocolo de comunicação é utilizado é *WebSockets* e tem suporte para praticamente todos os outros protocolos sendo *WebSockets* o principal. Infelizmente, *Elixir* ou *Erlang* são linguagens ao qual não existe conhecimento interno para a sua utilização.

VerneMQ (*Octavo Labs AG.*, s.d.), *HiveMQ* (*HiveMQ*, s.d.), *EMQX* (*EMQ Technologies Inc.*, s.d.) e *Emitter* (*Emitter*, s.d.) são tecnologias são baseadas no protocolo *MQTT*, embora com algumas diferenças nas suas implementações, todas estas oferecem possibilidade de escalar horizontalmente. A utilização do protocolo *MQTT* permite que a comunicação seja feita diretamente por *TCP* ou *WebSockets*. O protocolo *MQTT* tem como meio de comunicação principal *Pub/Sub*, no entanto, algumas funcionalidades extras podem a vir ser necessárias, algo que podem ser implementadas utilizando tópicos no *MQTT*.

SocketCluster (*SocketCluster*, s.d.) é uma biblioteca de *javascript* que permite a comunicação no formato de *Pub/Sub* e é capaz de escalar horizontalmente. Infelizmente a documentação não é extensiva, principalmente quanto ao subprotocolo. Adicionalmente, esta opção tem como objetivo primário servir como processador direto das mensagens recebidas, enquanto o objetivo pretendido é somente a distribuição, mas é possível adaptar para o caso necessário.

Soketi (s.d.) é um servidor de *WebSockets* compatível com o subprotocolo *Pusher v7*, permitindo que clientes desenvolvidos para esta plataforma possam ser reutilizados, adicionalmente, é capaz de escalar horizontalmente através da aplicação Redis.

Signal-R (*Microsoft*, s.d.) é uma biblioteca criada pela *Microsoft* que oferece a possibilidade de comunicação em tempo real com clientes, esta biblioteca funciona somente em servidores desenvolvidos em *C#* com a tecnologia *ASP.NET*. Esta opção é capaz de escalar utilizando a aplicação adicional *Redis* ou um serviço desenvolvido pela *Microsoft* disponível na *Azure Cloud*.

Desta lista de opções com código aberto a opção que mais se adequa é o *framework Phoenix*. Este é desenvolvido em *elixir* que por sua vez é executado na *Erlang VM*, a qual tem acesso a um conjunto de bibliotecas nomeadas de *OTP (Open Telecom Platform)* que facilita o desenvolvimento de aplicações distribuídas. Adicionalmente esta linguagem é utilizada por grandes plataformas como *WeChat* e *WhatsApps*, que servem como comprovativo para a sua escalabilidade. No entanto, *Elixir* ou *Erlang* são linguagens ao qual não existe conhecimento interno para a sua utilização.

Quanto à opção *Mercure*, esta não suporta o envio de mensagens bidirecionais, incluindo de clientes não autenticados, este exige que outros servidores sejam capazes de enviar mensagens pelos clientes ou que sirvam como meio de autenticação dos mesmos.

VerneMQ, *HiveMQ*, *EMQX* e *Emitter* são possíveis opções, no entanto estas ficam somente pelo protocolo *MQTT*, no entanto funcionalidades adicionais além das definidas no protocolo *MQTT*, teriam de ser desenvolvidas à parte, visto que o suporte para modificações é relativamente reduzido.

A opção *Soketi*, apresenta dois problemas, primeiro ser desenvolvida em *javascript* que por sua vez é executado em *node.js*, embora seja plataformas viáveis, este tipo de aplicação exige processamento simultâneo e paralelismo, tendo em conta que o *node.js* é executado como um processo de um único *thread*, este apresenta desvantagens quanto as outras possibilidades, adicionalmente, para utilizar eficientemente os recursos disponíveis seria necessário várias instâncias da mesma aplicação a correr em simultâneo com espaços de memória separados.

Por fim, *Signal-R* é uma boa opção para empresas que já usam *C#*, no entanto, este não é o caso, adicionalmente, de forma a escalar horizontalmente a aplicação *Redis* pode ser utilizada, mas o principal método é com um serviço desenvolvido pela *Microsoft* disponível na *Azure Cloud*, algo que também não é usado internamente.

Quanto aos serviços, a maior parte destes oferecem uma plataforma para a comunicação em tempo real, com suporte com vários protocolos e com escalabilidade gerida, no entanto, grande parte destes tem limitações no número de conexões.

Ably (s.d.) é uma plataforma de mensagens *Pub/Sub* com garantia de envio, ordem de envio, e com suporte para vários protocolos tais como *MQTT, STOMP, AMQP, PUSHER* e *PubNub*. Permite conexões com os protocolos *WebSockets*, *SSE* e o envio de mensagens por *HTTP*. Adicionalmente, permite o rastreamento da presença dos clientes, envio de notificações *push*, oferece um histórico de mensagens e com suporte para restaurar desconexões abruptas.

PubNub Inc (2022) é uma plataforma de mensagens *Pub/Sub* sem garantia de envio ou ordem de envio, os protocolos utilizados não são especificados, no entanto, segundo os exemplos apresentados utilizam a técnica *long-polling*. Esta plataforma também permite o rastreamento da presença dos clientes, envio de notificações *push* e processamento de mensagens enviadas.

Pusher Ltd (s.d.) é uma plataforma similar às anteriores, funciona igualmente com mensagens *Pub/Sub* mas sem garantia de ordem e envio. Esta utiliza conexões com o protocolo *WebSockets* e sub-protocolo *Pusher*, um protocolo proprietário. Assim como as opções anteriores também permite o rastreamento da presença dos clientes. Algumas funcionalidades que não oferecem são um histórico de mensagens, recuperação de mensagens perdidas. Notificações *push* são possíveis, mas fazem parte de um serviço à parte oferecido pela mesma empresa.

Fanout (*Fanout*, s.d.) é uma que opção oferece tanto uma versão com código aberto quanto um serviço. A opção de código aberto serve como um intermediário entre outros serviços onde estes podem enviar atualizações para serem distribuídas pelos clientes, esta opção não é horizontalmente escalável sem adaptação dos serviços para o envio de mensagens utilizando um protocolo de comunicação *ZeroMQ* ou então publicando para todas as instâncias. A versão de serviço, oferece mais funcionalidades, como organização de *channels* (equivalente a um tópico) por *realms* (um elemento que agrupa *channels*). Ambas opções permitem conexões com os protocolos *WebSocket, SSE* e *long-polling*. Ao contrário das opções anteriores, o rastreamento da presença de clientes, envio de notificações *push* não suportadas, adicionalmente o suporte para ordem e garantia de envio das mensagens é parcial.

Estas quatro opções, são plataformas que oferecem uma maior abstração aos sistemas *Pub/Sub*, estas oferecem funcionalidades tipicamente não existentes em um *message broker* tais como o rastreamento de presenças, envio de notificações *push* e histórico de mensagens. Desenvolver estas funcionalidades em algumas das opções apresentadas que não as oferecem necessitam modificações no projeto em si, algo que iria exigir familiaridade com o funcionamento interno destes. Quanto às plataformas apresentadas, nomeadamente *Ably*, *PubNub*, *Pusher* e *Fanout*, as que mais cumprem os pontos a ter em consideração são *Ably*, *Pusher* e *PubNub* na ordem que melhor cumprem. Embora estas opções não tenham integração com a aplicação *NATS*, seria possível adaptar para o que a plataforma oferece ou então desenvolver uma ferramenta adicional que se realiza a conversão.

A plataforma *Pusher* quando falamos de meios de comunicação e funcionamento dos mesmos, cumpre os requisitos, incluindo o rastreamento de presença através de tópicos especializados para o caso, tópicos públicos e privados utilizando um prefixo no seu nome. No entanto, nenhum dos tipos de tópicos tem a capacidade de armazenar um histórico de mensagens. Outro problema comum em plataformas, que ocorre neste caso é o número de conexões, cada loja tem a sua aplicação e o seu conjunto de clientes, e a empresa tem de estar preparada para uma elevada quantidade de conexões em simultâneo, no caso do *Pusher* o plano maior listado oferece no máximo 30 mil conexões, exigindo além disso negociar com a empresa.

A plataforma *PubNub*, não estabelece conexões utilizando o protocolo *WebSocket*, em vez disso utiliza pedidos *HTTP* e uma espécie de *long-polling* o custo de performance e energia para as aplicações acabar por ser mais elevado, e não sendo uma conexão bidirecional este não permite o envio bidirecional de mensagens, no entanto, todas outras funcionalidades estão presentes.

Por fim, *Ably* é a plataforma que melhor cumpre os pontos previamente mencionados, esta permite conexões por *WebSockets* e outros protocolos, rastreamento de presenças, garantia na ordem e entrega de mensagens, armazenamento opcional das mensagens, e agrupamento de tópicos permitindo um conjunto de tópicos ter a mesma configuração. No entanto, assim como no *Pusher* o limite de conexões se mantém.

2.3 Solução personalizada

Após todas estas possibilidade terem sido analisadas, foi decidido desenvolver um novo sistema em vez de reutilizar as opções mencionadas pelos seguintes motivos:

- Extensibilidade;
- Limites da *API*;
- Adaptação ao caso de uso;
- Imprevisão de custo;
- Conhecimento existente na empresa.

Muitos destes serviços oferecem sistemas simples de *PubSub*, no entanto, pouca personalização além disso, sendo que caso seja necessário funcionalidades além das oferecidas em conjunto com o sistema *PubSub*, é necessário as desenvolver num sistema separado. Por exemplo, um sistema de presença em conjunto com meta dados sobre todos utilizadores subscritos num tópico é uma funcionalidade que pode estar embutida num tópico, mas desenvolver um sistema só para esta funcionalidade não é prático.

Dentro de todos serviços apresentados, o que mais se destacou por ser o mais próximo de atender a todos requisitos é o serviço *Ably*, no entanto, assim como os serviços em geral apresenta limites na utilização da sua *API*, como por exemplo, limites de eventos num tópico por segundo e máximo de utilizadores subscritos num *channel*. Adicionalmente, sendo o número de conexões um valor que flutua bastante, assim como o número de eventos enviados para tópicos, prever os custos dos serviços torna-se difícil e sem forma de implementar um teto máximo

Quanto às opções de código aberto, muitas destas não cumprem os requisitos necessários, sendo necessário adaptar os projetos e ter o custo extra de manutenção de manter o projeto atualizado com novas funcionalidades implementadas no código base. De todas as opções, a que melhor cumpre os requisitos necessários é o *framework Phoenix*, utilizando a tecnologia presente na *Erlang VM* este permite criar um sistema distribuído, e adicionalmente o *framework Phoenix* permite customizar o funcionamento dos tópicos. No entanto, este *framework* utiliza as linguagens *Elixir* e *Erlang*, que são linguagem ao qual não existe conhecimento interno para sua utilização.

Tendo esta informação em conta, a criação de um novo sistema foi o caminho decidido de forma reutilizar o conhecimento existente da linguagem *Go* (*The Go Programming Language*, s.d.) e ferramentas já utilizadas internamente como a aplicação *NATS*.

3 METODOLOGIA

Para levantamento de requisitos, foi usado como base o sistema já presente em produção, visto que grande parte das suas funcionalidades são necessárias por outros serviços dentro da empresa NAPPS. Estando a substituir um sistema em utilização internamente, já existe um conhecimento prévio de problemas que existiam, ou melhorias desejadas. Portanto, utilizando o *feedback* dos utilizadores do sistema, em conjunto com funcionalidades futuras previstas, foi realizado um *brainstorming* onde se definiu o que o projeto precisava, assim como vai ser visto ao longo deste documento.

3.1 Tarefas

O projeto está dividido em várias tarefas, algumas das tarefas vão envolver vários pontos que serão descobertos ao longo da fase de pesquisa e possivelmente em adaptações a novas funcionalidades. As tarefas definidas até ao momento são:

- Tarefa 1 – Pesquisa de possíveis soluções existentes e avaliação das mesmas;
- Tarefa 2 – Pesquisa do funcionamento das atuais soluções;
- Tarefa 3 – Elaborar funcionamento do projeto;
- Tarefa 4 – Avaliar possíveis problemas de migração para novo projeto;
- Tarefa 5 – Desenvolvimento de protótipo;
- Tarefa 6 – Teste de protótipo e avaliar possíveis problemas;
- Tarefa 7 – Corrigir possíveis problemas ou adaptar para possíveis utilizações;
- Tarefa 8 – Teste em Cloud (AWS);
- Tarefa 9 – Criação de testes para cobrir lógica de projeto;
- Tarefa 10 – Implementação em produção em fase de teste.

Após mencionadas as tarefas para realização, passo a elaborar o que cada constitui.

Na tarefa 1, é realizada uma pesquisa por possíveis soluções comerciais ou de código aberto e análise rápida se estas podem cobrir os casos de utilização atual, na tarefa 2 após a eliminação de soluções que não se adaptam aos casos de utilização, iremos verificar mais profundamente o seu funcionamento, e como se comportaria em funcionalidades planeadas e custos para as mesmas. Utilizando o conhecimento do funcionamento obtido pelas tarefas 1 e 2, é elaborado um plano geral com todas as funcionalidades necessárias e o seu funcionamento interno, após esta será elaborado uma análise de problemas que possam existir ao realizar a migração do

projeto anterior para o atual, quanto menor o custo de migração menor será o tempo para introduzir em produção e atualização de sistemas em produção, e esta etapa será a tarefa 4.

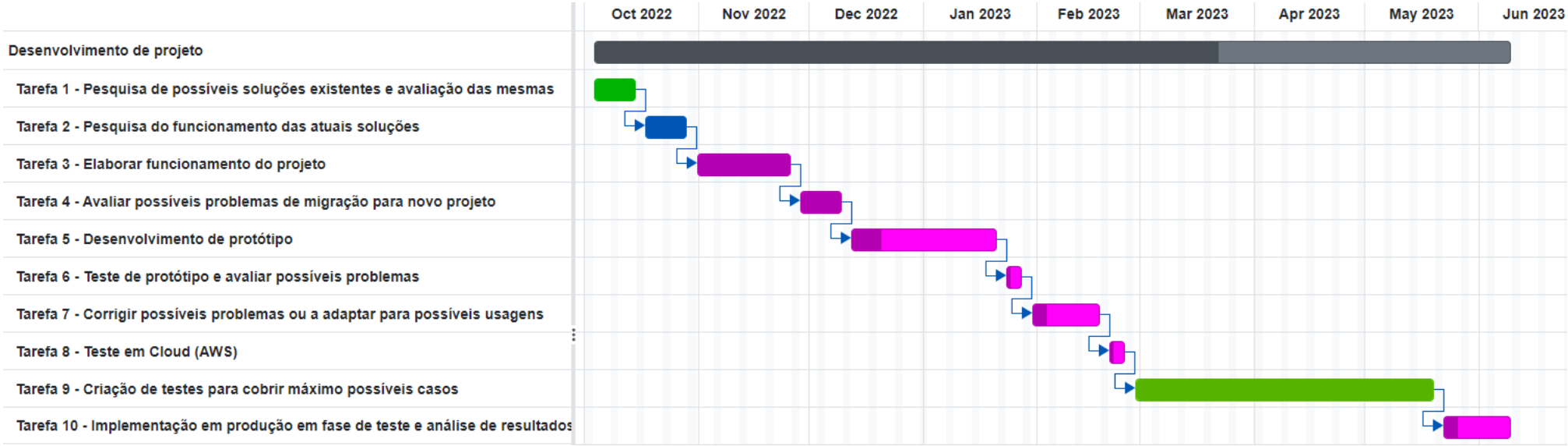
Após ter sido realizada uma análise do funcionamento e tendo sido verificado possíveis partes problemáticas, é realizado o desenvolvimento de um protótipo do projeto como tarefa 5, os testes mais manuais serão realizados e serão avaliados possíveis problemas que tenham ocorrido, este passo corresponde à tarefa 6, para tarefa 7, será a correção dos erros que tenham sido encontrados e adaptação para funcionalidades que tenham surgido ou adaptação das atuais.

Por fim, o funcionamento será testado na *cloud AWS* e o desenvolvimento de testes e ferramentas de análise para ser possível inspecionar os funcionamento e erros que ocorram com o projeto em funcionamento na *cloud*, e como última etapa o projeto será posto em produção, mas em fase de teste com tráfego real, mas em componentes que não sejam críticos, estas três partes serão as tarefas 8, 9 e 10.

3.2 Cronograma

O cronograma que representa as tarefas previamente definidas para o projeto é representado na seguinte forma, como na figura 1.

Figura 1 - Diagrama de Gantt



Fonte: Própria

4 DESENVOLVIMENTO

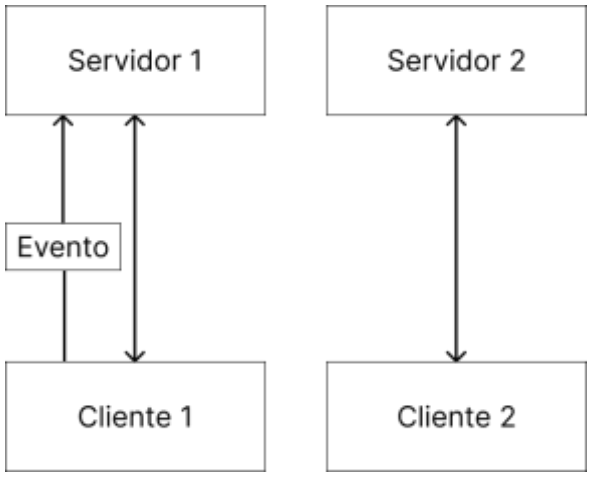
Neste capítulo são apresentadas as decisões que foram tomadas inicialmente, nomeadamente a estrutura inicial e a utilização da aplicação *NATS*. Também é apresentado os motivos que levaram a desconsiderar a aplicação *NATS*, assim como a alternativa que foi implementada e por fim as funcionalidades existentes na aplicação.

4.1 Princípio de funcionamento

Portanto seguindo o sistema anterior, existe somente um servidor onde todos os clientes estão conectados. Caso este servidor falhe, os clientes ficam sem forma de utilizar o serviço. De forma a evitar que isso aconteça, é necessário adicionar mais servidores, assim caso um falhe existem outros que podem receber as conexões. A isto nomeamos de ser horizontalmente escalável, caso um servidor falhe ou não seja capaz de aguentar o número de clientes atual, existem outros servidores para receber estes clientes.

No entanto, quando falamos num sistema *PubSub* é necessário que quando um evento é publicado num tópico, este tem de ser transmitido para todos os clientes subscritos neste mesmo tópico, independentemente a qual servidor estes estão conectados. Se olharmos para a figura 2, temos os Servidores 1 e 2, e um cliente conectado a cada um. Estando ambos clientes subscritos ao mesmo tópico é necessário que o evento que esta a ser publicado no servidor seja transmitido para o Cliente 2, o que não acontece visto que o Cliente 2 não está conectado no mesmo servidor que recebe o evento.

Figura 2 - Sistema anterior



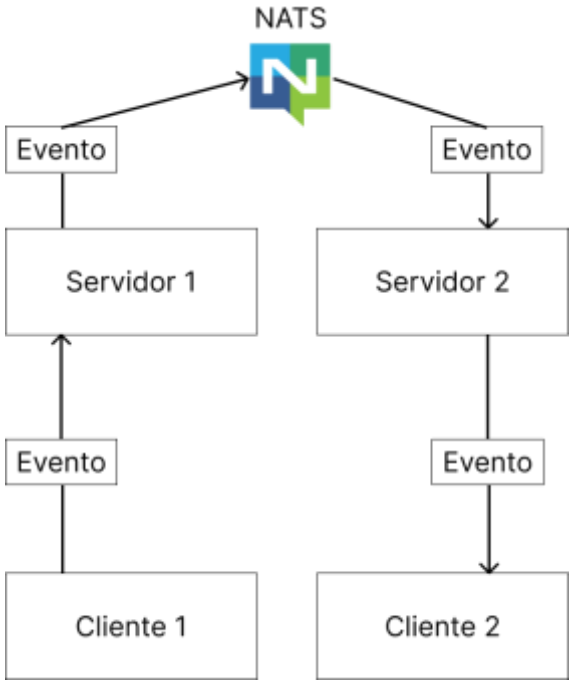
Fonte: Própria

De forma a resolvermos este problema, tinha sido inicialmente planeado a utilização da aplicação *NATS*, para realizar a intercomunicação entre os servidores.

O *NATS.io* (s.d.) é uma aplicação de mensagens de código aberto que fornece um sistema de mensagens de alta performance e baixa latência. Este é usado principalmente para conectar diferentes partes de um sistema distribuído, permitindo que as diferentes partes se comuniquem e troquem informações de maneira eficiente e confiável. Inclusive, a aplicação é utilizada de forma a ter outros serviços a comunicar com a aplicação desenvolvida neste projeto, estes são considerados serviços autenticados que podem utilizar uma *API* similar à de administração.

Desta forma, o evento publicado pelo Cliente 1 é transmitido pelo *NATS* para todos os servidores interessados no tópico, assim como pode ser visto na figura 3.

Figura 3 - Intercomunicação com NATS



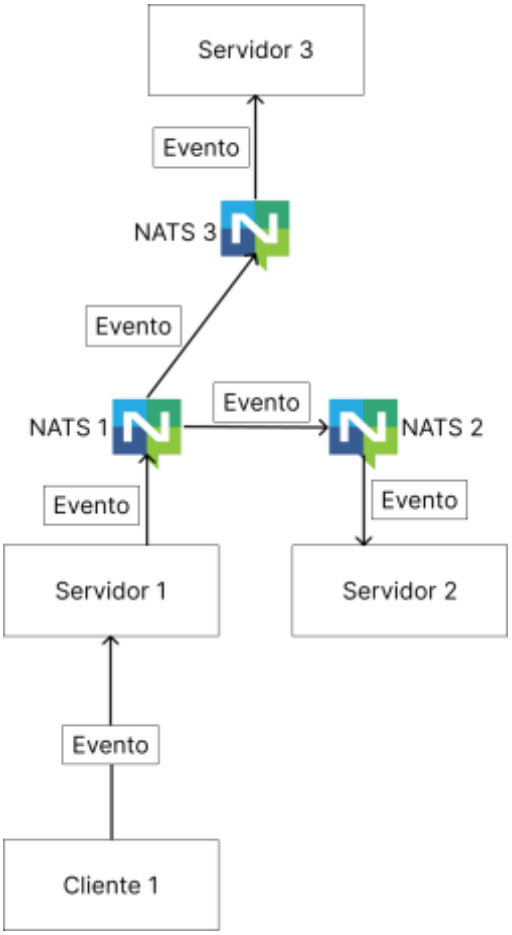
Fonte: Própria

Com esta solução, foi observado um problema com a utilização do *NATS*, sendo este a ineficiência introduzida na passagem de um evento, principalmente quando é aumentado o número de servidores em funcionamento.

Na figura 4, existem 3 servidores e 3 instâncias da aplicação *NATS*. Existe um número mais elevado de servidores de forma a ter redundância em caso de falhas e de forma a ser capaz de receber um maior número de conexões. Nesta figura temos o Cliente 1 que publica um evento que tem de chegar aos Servidores 2 e 3, para isso, assim que o Servidor 1 receba o evento

publicado pelo Cliente 1, este tem de enviar o evento para o *NATS* 1, que por sua vez envia para o *NATS* 2 e 3 que por fim enviam aos Servidores 2 e 3. Portanto, foi necessário que o evento fosse passado 5 vezes por rede, relembrando que cada passagem exige a codificação da mensagem por quem envia e descodificação por quem recebe.

Figura 4 - Ineficiência na intercomunicação com a aplicação NATS



Fonte: Própria

4.2 Alternativa à aplicação *NATS*

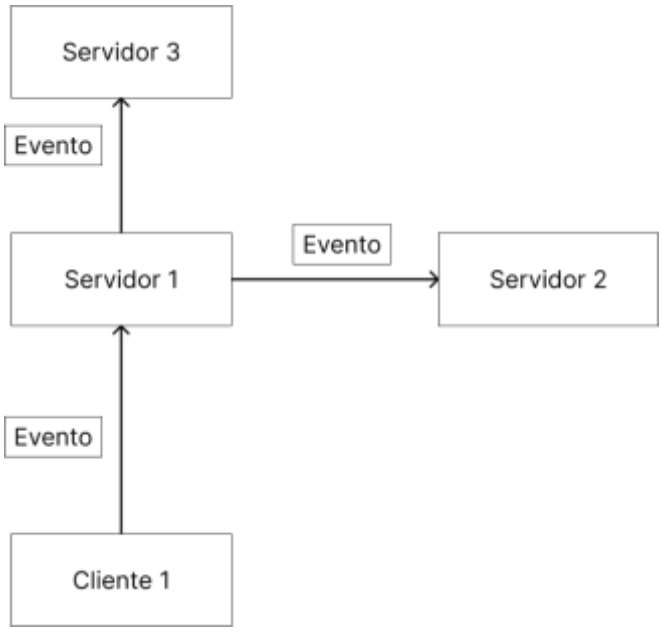
De forma a evitar esta ineficiência, podemos ter os servidores a comunicar entre si em vez de utilizar a aplicação *NATS* como intermediário. Para isso, é necessário implementar algo que seja capaz de substituir a utilização da aplicação *NATS*, ou seja, é necessário resolver os 3 seguintes pontos:

- Consenso;
- Intercomunicação;
- Distribuição.

O consenso consiste em ter conhecimento de quais servidores estão ativos. A utilização da aplicação *NATS* evitava esta necessidade, afinal os eventos eram publicados no *NATS* e este iria distribuir o evento por quem está interessado. A intercomunicação consiste no envio de informações e eventos entre os servidores. A distribuição consiste em como os tópicos ou *channels* são distribuídos pelos servidores.

Se os servidores forem capazes de se comunicarem entre si, então o número de vezes que um evento tem de ser enviado por rede diminui, assim como pode ser observado na seguinte figura 5.

Figura 5 - Intercomunicação direta



Fonte: Própria

Antes explicar como os 3 pontos mencionados foram resolvidos, será explicado o porquê de ser atribuído um conjunto de *channels* a um servidor de forma a que se torne mais claro o porque do ponto “distribuição”.

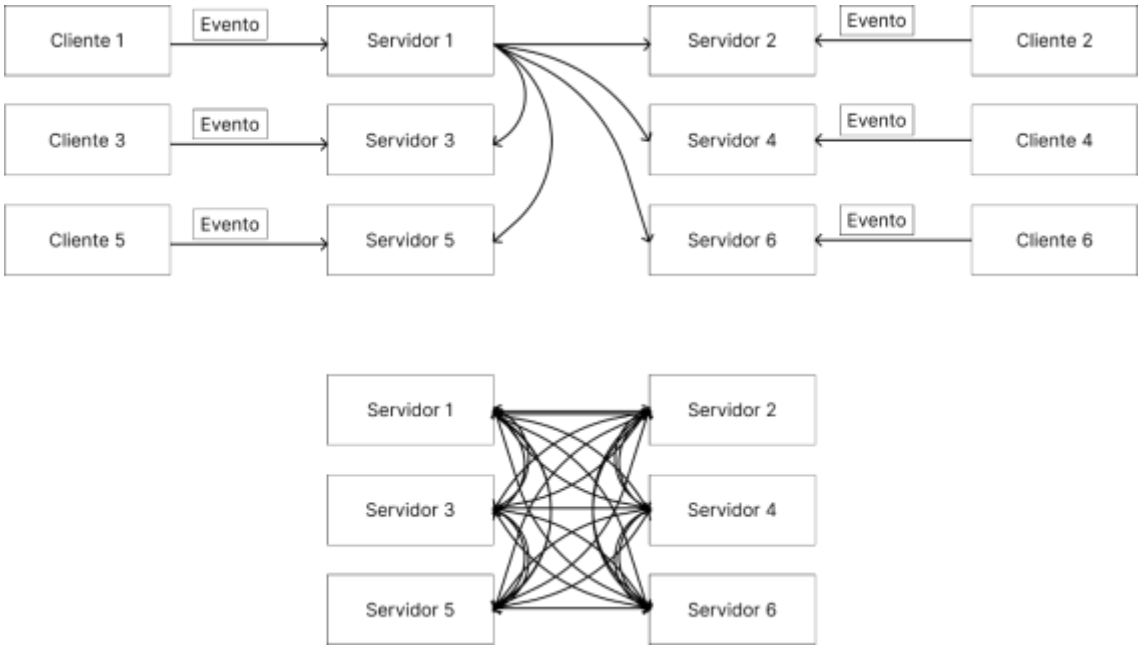
4.2.1 Centralização do *channel*

Portanto, se seguirmos o modelo anterior, onde sempre que um evento é publicado este é enviado para todos os servidores, iremos aumentar o número de vezes que um evento é enviado por rede, assim como podemos enviar eventos para servidores que não têm interesse no evento.

De forma a exemplificar, temos a figura 6 onde existem 6 servidores, cada um com um cliente conectado, e todos estes subscritos ao mesmo tópico ou *channel*.

Imaginando que todos os clientes querem publicar um evento no mesmo *channel*, cada um dos servidores vai ter que enviar o evento recebido pelo cliente para todos os outros servidores. Portanto cada servidor envia 5 mensagens, com 6 servidores são 30 mensagens enviadas como pode ser visto na mesma figura na parte de baixo.

Figura 6 - Transmissão de mensagens



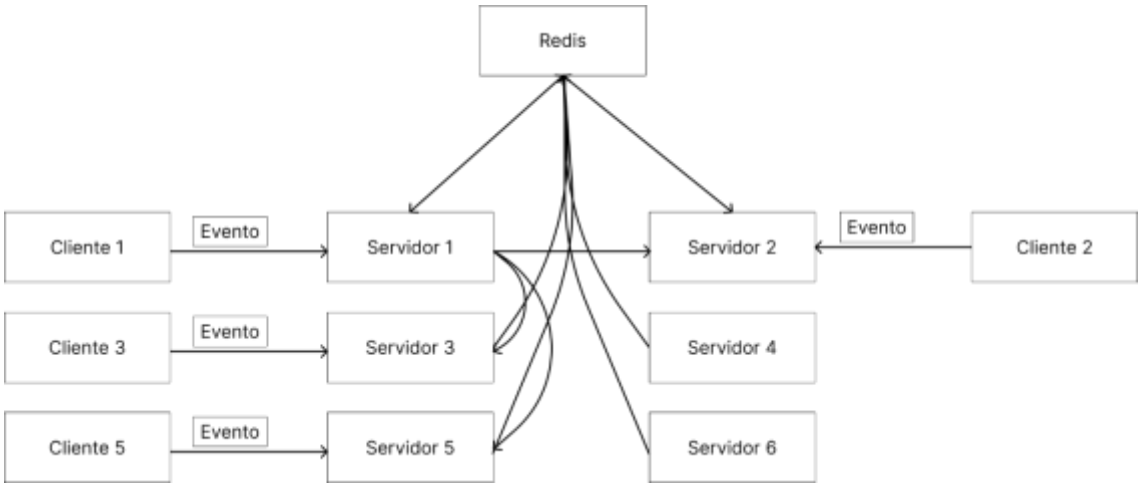
Fonte: Própria

Caso um dos servidores não tenha interesse nos eventos enviados, assim que receber o evento, irá descartá-lo, ou seja, uma mensagem enviada por rede que é desnecessária.

Para enviar eventos apenas para servidores interessados no *channel*, cada servidor tem que registar quais *channels* está interessado, para isso, a aplicação *Redis* pode ser utilizada, lembrando que também tem que existir várias instâncias da aplicação *Redis* de forma a ter redundância.

Portanto, por cada vez um evento é publicado, o servidor tem de consultar a aplicação *Redis* de forma a saber para quais servidores deve enviar o evento, assim como na figura 7.

Figura 7 - Transmissão com Redis



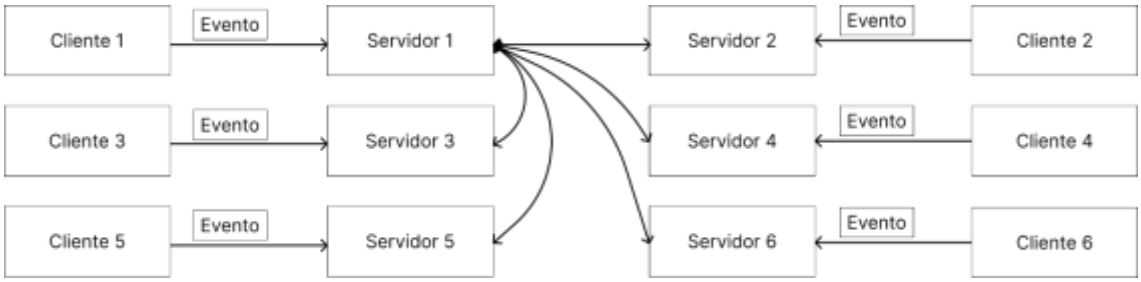
Fonte: Própria

Embora este método permita evitar enviar mensagens desnecessárias, a necessidade de ter de consultar outra aplicação externa por rede por cada evento publicado pode aumentar consideravelmente a latência do envio de mensagens, e caso a aplicação *Redis* apresente falhas o sistema em geral vai apresentar falhas.

Um método alternativo, consiste em atribuir a um servidor a responsabilidade de gerir um conjunto de *channels*, ou seja, para publicar um evento num *channel*, este tem de ser enviado para o servidor responsável pelo *channel*. Desta forma conseguimos manter uma ordem no envio de eventos assim como reduzimos a complexidade do envio de eventos e evitamos ter que consultar uma aplicação externa por cada evento publicado.

Voltando ao exemplo anterior, que pode ser observado na seguinte figura 8, onde existem 6 servidores com um cliente conectado a cada, e todos clientes estão subscritos ao mesmo *channel* e todos querem publicar um evento. Neste exemplo, vamos assumir que o responsável pelo *channel* onde os clientes estão subscritos e pretendem publicar é o Servidor 1. Portanto, os servidores 2 a 6 vão enviar o evento recebido pelos seus clientes para o Servidor 1 (5 mensagens), o Servidor 1 vai enviar os eventos publicados para os servidores 2 a 6, ou seja, 6 × 5 = 30 mensagens, tendo no total 35 mensagens enviadas por rede.

Figura 8 - Channel centralizado



Fonte: Própria

É verdade que são mais mensagens transmitidas do que no exemplo anterior, no entanto, se os servidores forem capazes de saber qual o responsável pelo *channel* evitam de ter de consultar a aplicação *Redis* e ao mesmo tempo a complexidade do sistema em geral é reduzida. Adicionalmente, caso os eventos sejam todos publicados ao mesmo tempo, estes podem ser agrupados e enviados numa só mensagem, ou seja, em vez das 30 passariam para somente 5.

Voltando aos 3 pontos anteriormente mencionados, sendo estes:

- Consenso;
- Intercomunicação;
- Distribuição.

Iremos começar com o consenso, o que este é, que opções foram avaliadas e qual foi a escolhida.

4.3 Consenso

O consenso é um dos problemas fundamentais em sistemas distribuídos, este exige que múltiplos membros concordem em um conjunto de valores mesmo na presença de falhas. Um protocolo de consenso que seja capaz de tolerar falhas deve cumprir as seguintes propriedades: terminação, sendo que eventualmente todos membros concordam com um valor; integridade, caso os membros proponham o mesmo valor, então outros devem decidir no mesmo valor; concordância, todos membros devem concordar no mesmo valor. Algoritmos de consenso tendem a confirmar um valor quando a maioria dos membros do *cluster* esteja disponível, por exemplo, um *cluster* de 5 membros pode continuar a operar com a falha de dois membros, no entanto, caso mais que dois falhem estes deixam de conseguir alterar os valores e somente retornam os valores previamente acordados.

Portanto, o consenso em sistemas distribuídos é um processo em que vários membros de um sistema distribuído trabalham em conjunto para tomar uma decisão em comum. Este processo

é necessário quando há vários componentes no sistema e é preciso chegar a um acordo sobre qual ação deve ser tomada. Por exemplo, num *cluster*, é necessário que todos os membros saibam qual membro é responsável por determinada tarefa ou quais dados estão disponíveis em cada membro.

Para alcançar o consenso, os sistemas distribuídos utilizam algoritmos de consenso, como o algoritmo *Paxos* ou o algoritmo *Raft*, que são projetados de forma a garantir que todos os membros no sistema tenham a mesma visão dos dados e das ações a serem tomadas. Estes algoritmos permitem que os membros elejam um líder ou coordenador que tomará as decisões, enquanto os outros membros seguirão as instruções do líder.

O consenso em sistemas distribuídos é fundamental para garantir a consistência e a integridade dos dados em todo o sistema. Neste são considerados os protocolos *Raft* e *Gossip* como protocolos de consenso.

4.3.1 Raft

Raft é um algoritmo de consenso com propósito de ser simples de compreender, este é equivalente ao algoritmo *Paxos* a nível de tolerância de falhas e desempenho. Este atinge o consenso através de um e somente um líder eleito. Neste protocolo, cada membro tem o cargo de líder ou seguidor e pode ser um candidato caso um líder não exista. O membro com o cargo de líder tem a responsabilidade de replicar *logs* para os seguidores, adicionalmente, este regularmente informa os seus seguidores da sua existência através do envio de um *heartbeat*. Cada seguidor tem um ciclo de intervalos de tempo em qual espera receber um *heartbeat* do líder que é reiniciado sempre que o receba, no entanto, caso o intervalo de tempo termine sem o receber, então, o seguidor muda o seu cargo para candidato e começa uma eleição para um novo líder. Portanto, o protocolo *Raft* está dividido fundamentalmente em duas partes: eleição de líder e replicação de *logs*.

Quando o algoritmo inicializa ou um líder falha, um novo líder tem de ser eleito. Neste caso, é iniciado um novo termo no *cluster*. Um termo é um período arbitrário no *cluster* para o qual um novo líder precisa ser eleito, cada termo começa com a eleição de um novo líder. A eleição de um líder é iniciada por um membro candidato, este aumenta o contador de termo, vota em si mesmo como novo líder e envia uma mensagem para todos os outros membros a pedir o seu voto. Cada membro só pode votar uma vez por cada termo, e estes votam a favor do primeiro pedido de voto que receberam. Caso um candidato receba uma mensagem de outro membro com um contador de termo superior então este é automaticamente desqualificado e muda o seu

cargo de volta para seguidor. Caso um membro receba a maioria de votos então este torna-se o novo líder, caso exista um empate de votos então um novo termo é começado e o processo é repetido, adicionalmente, de forma a evitar ciclos de empate de votos, cada membro escolhe um intervalo de tempo aleatório, com valores reduzidos, antes de voltar a tentar a nova eleição.

Quanto à segunda parte, a replicação de *logs*, esta é a responsabilidade do líder, este recebe pedidos de clientes, sendo que cada pedido consiste num comando a ser executado e replicado por todos membros do cluster. Após o comando seja adicionado à lista de *logs* do líder, este envia este comando para todos os seguidores. Caso os seguidores não estejam disponíveis, o líder volta a tentar enviar o comando por vezes indefinidas até que o *log* seja eventualmente adicionado à lista dos seguidores. Assim que o líder recebe a confirmação, de metade ou mais dos seus seguidores, que o comando foi replicado, este aplica o comando ao seu estado local e o pedido é considerado como aplicado.

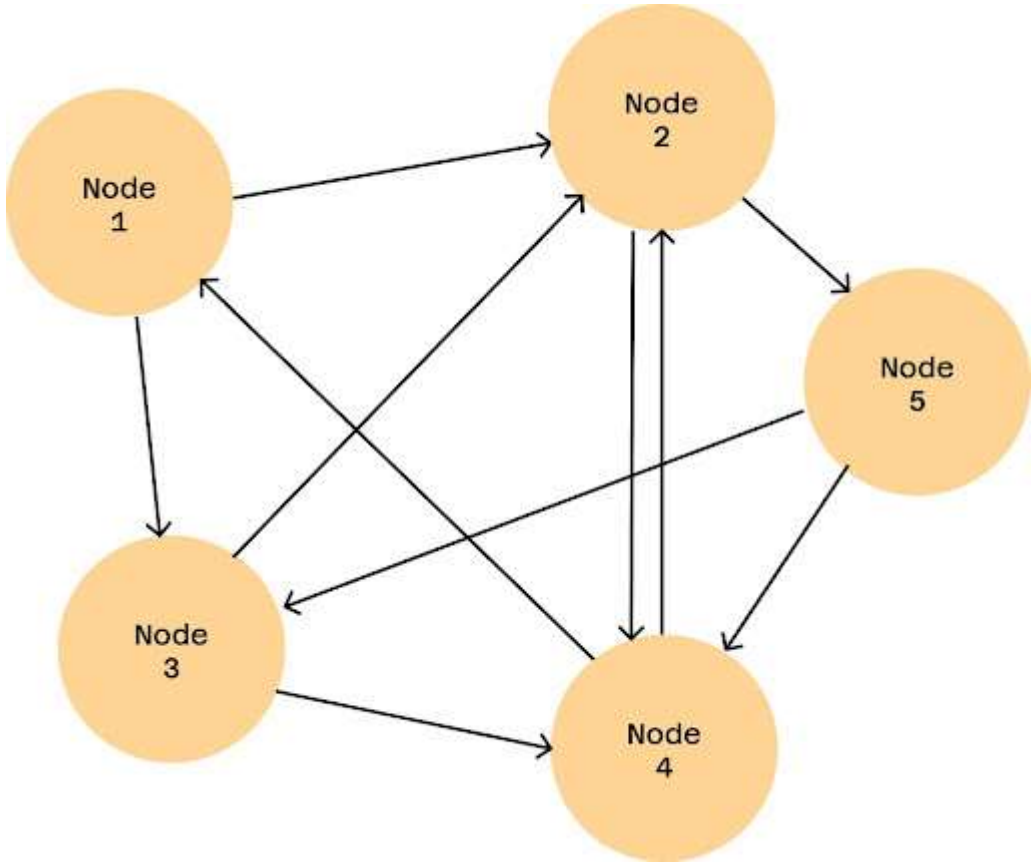
Este protocolo é utilizado quando é necessário que exista uma forte consistência de informações no *cluster*, sendo permitido apenas ao líder realizar alterações, um exemplo comum de utilização deste protocolo pode ser encontrado nas bases de dados *CockroachDB*, *MongoDB*, *Neo4j*, *TiDB* e *YugabyteDB*. Sendo que somente um membro do *cluster* é capaz de realizar alterações, a capacidade do *cluster* é limitada pela capacidade do líder. De forma a resolver este problema, é utilizado o *Multi-Raft*, este utiliza múltiplos grupos tendo cada um o seu líder e gerindo uma secção da informação. No caso de uma base de dados, podemos ter um grupo por cada tabela e aplicar alterações a grupos separados aumentando a quantidade de alterações possíveis e distribuindo a carga entre mais membros. Adicionalmente, caso somente seja necessário a consulta de informações, esta pode ser realizada a qualquer seguidor, com o risco de receber informação desatualizada ou então realizar a consulta ao líder para ter a garantia de ter a última informação.

4.3.2 Gossip

O protocolo *gossip* ou protocolo epidêmico consiste em um procedimento de comunicação *peer-to-peer* que assimila a forma como as epidemias ou rumores se espalham, neste protocolo cada membro de grupo periodicamente troca informação com outros membros sobre o seu próprio estado e sobre o estado de outros membros. Este protocolo permite que um sistema distribuído tenha a garantia que a informação é eventualmente distribuída por todos os membros do grupo sem precisar de um sistema centralizado a coordenar esse aspeto. Visto não precisar de um sistema centralizado este protocolo é dos mais robustos e escaláveis para

consistência eventual dos membros do *cluster*, deteção de falhas e permite o envio de informações adicionais durante as trocas de informação.

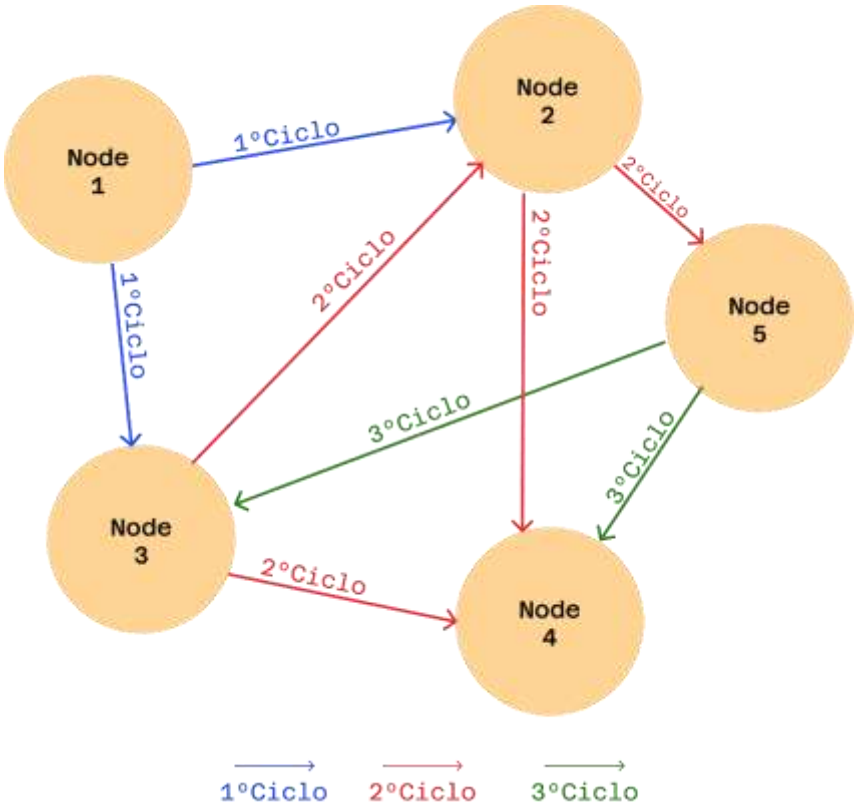
Figura 9- Exemplo de cluster a utilizar protocolo gossip.



Fonte: Própria

Na figura 9 podemos ver um exemplo de um *cluster* com 5 membros, neste exemplo cada membro comunica somente com outros 2 membros, assim como representado pelas setas. De forma a propagar uma informação entre todos os membros seriam necessários 3 ciclos, sendo cada ciclo uma troca de informação entre os membros após cada intervalo de tempo definido no *cluster*. Uma exemplificação da propagação com origem no *Node 1* pode ser observada na figura 10.

Figura 10 - Exemplo de propagação



Fonte: Própria

No caso de um *cluster* com 40 membros e cada membro comunique somente com outros 4 membros seriam necessários somente 4 ciclos. O artigo "Epidemic Algorithms for Replicated Database Maintenance" (Demers et al., 1987) descreve algoritmos de replicação de bases de dados que usam a propagação de informações entre membros de um sistema distribuído, este apresenta uma fórmula para estimar o tempo de convergência do algoritmo de propagação de informações com base no número de membros do sistema e na taxa de propagação de informações. Essa fórmula é $T = O(log(N) \ / \ p)$, onde N é o número de membros do sistema e p é a taxa de propagação de informações e $O(log(N))$ o número de ciclos necessários para que a informação seja propagada por todo o sistema. Portanto, de forma a calcular aproximadamente quantos ciclos são necessários para a propagação de uma informação, iremos nos focar apenas na parte O(log(N)), desta forma, iremos utilizando o seguinte cálculo $C = log_P(N)$ onde c é o número de ciclos.

Portanto, com 40 membros e propagação de 4 temos $log_4(40) = 2.66$, ou seja, aproximadamente 3 ciclos, no caso de um *cluster* com 5 membros e propagação de 2 temos $log_2(5) = 2.32$ que também são aproximadamente 3 ciclos.

Vendo o protocolo de alto nível, num *cluster*, cada membro mantém uma lista de um subconjunto dos membros a que tem conhecimento, os seus endereços e alguns dados adicionais (metadata), e periodicamente, cada membro atualiza na sua lista de "vizinhos" os contadores de *heartbeat* de acordo com os dados emitidos por outros membros e envia a informação atualizada para alguns dos membros. Assim que um membro tenha recebido uma das mensagens, esta junta a lista na mensagem com a sua lista e mantém os dados com o contador de *heartbeat* mais elevado no caso de colisões. Assim sendo, enquanto o valor do contador for subindo para um membro é garantido que este esteja *healthy* (ativo e sem problemas) e é considerado *unhealthy* (desativo ou com problemas) caso o contador de *heartbeat* não seja aumentado durante um intervalo de tempo. Adicionalmente, durante a troca de informações entre membros é possível enviar informações extra como por exemplo, carga média e memória livre para que outros membros possam utilizar essa informação para balancear a carga entre membros. Outra forma de explicar o protocolo *gossip* é comparando com a disseminação de rumores numa comunidade. Assim como no protocolo *gossip*, um rumor começa com uma pessoa que o compartilha com alguns amigos próximos. Esses amigos, por sua vez, compartilham o rumor com outros amigos, e assim sucessivamente. Conforme o rumor se espalha, este pode ser confirmado, negado ou até mesmo modificado por diferentes pessoas ao longo do caminho. O resultado é uma ampla disseminação de informações pela comunidade, com a possibilidade de chegar a um consenso ou opinião comum. Da mesma forma, o protocolo *gossip* permite a disseminação de informações em sistemas distribuídos, onde diferentes membros compartilham e modificam informações entre si até chegarem a um consenso ou estado comum.

No caso deste projeto, o protocolo *gossip* é baseado em "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol" (Das, Gupta & Motivala, 2002) com algumas modificações. A implementação foi criada pela empresa *Hashicorp* e foi nomeada de *Serf*. Explicando de forma breve e incompleta, um membro começa por se juntar a *cluster* já existente ou cria um novo, caso se esteja a juntar, é realizada uma sincronização completa com um membro já existente do *cluster* utilizado o protocolo *TCP* e depois começa a realizar trocas de informação assim como referido previamente. Neste caso, a comunicação utilizada para troca de informações utiliza o protocolo *UDP* com o número de propagação de intervalo configurável. Nesta implementação é apenas enviado alterações de informação com o protocolo *UDP*. Mesmo após um membro se juntar ao grupo algumas sincronizações completas ocorrem com outro membro aleatório utilizando o protocolo *TCP*, no entanto, estas ocorrem

com menor frequência, o intervalo destas transmissões também pode ser configurado ou desativado.

De forma a detetar uma falha, um pedido de verificação é enviado aleatoriamente num intervalo de tempo configurável, caso o destinatário falhe a responder dentro de um prazo de tempo razoável então um pedido de verificação é enviado indiretamente. Um pedido de verificação indireto passa por pedir a um número configurável de membros para realizarem um pedido de verificação ao membro, isto permite perceber se um membro não está acessível por problemas que estejam a ocorrer na rede. Caso ambas tentativas falhem, então o membro é marcado como suspeito e estas informações são enviadas para todo o *cluster* utilizado o mesmo mecanismo de propagação. Por fim, caso o membro suspeito não responda à suspeita num intervalo de tempo configurável então o membro é considerado como morto, e novamente esta informação é propagada pelo *cluster*. Outra funcionalidade desta implementação passa por permitir o envio de eventos e consultas utilizando o mecanismo de propagação, algo que pode ser utilizado, por exemplo, quando a configuração do *cluster* muda e é necessário que esta alteração seja propagada por todos os membros.

4.3.3 Escolha de protocolo de consenso

Tendo revisto as opções *gossip* e *Raft*, a opção escolhida para ser utilizada neste projeto passa pelo *gossip*. Tendo como objetivo que todos membros concordem com quais membros estão ativos, o algoritmo *Raft* oferece mais funcionalidades do que as necessárias e mais restrições do que a opção *gossip*, adicionalmente, não sendo necessário armazenar informação ou sendo exigido uma forte consistência de informação é preferível a utilização do protocolo *gossip* sendo este mais eficiente no consumo de recursos de processamento e de rede e permite uma quantidade mais elevada de membros sendo que o algoritmo *Raft* tem o seu melhor desempenho num *cluster* com 3 a 9 membros enquanto em *gossip* um número muito mais elevado é possível, por exemplo, em *gossip* um *cluster* com 100 membros e propagação de 4 leva aproximadamente 3 ciclos a propagar a informação.

A utilização de ambos protocolos em simultâneo também é possível, utilizando o protocolo *gossip* de forma a manter uma lista de membros ativos, e utilizar o algoritmo *Raft* apenas para gerir a consistência de informação, no entanto, como previamente mencionado, a utilização do *Raft* limite consideravelmente o número de membros a serem utilizados num *cluster*.

Utilizando o *gossip*, é possível manter uma consistência eventual dos membros presentes no *cluster*, e esta informação é somente utilizada de forma a realizar intercomunicação entre os

membros do *cluster*. Não tendo o protocolo *gossip* como objetivo de enviar informação de forma rápida, será antes utilizada a informação que este gere para utilizar outro método de envio de informação para o resto dos dados aplicacionais, adicionalmente, também é necessário organizar os membros de forma a evitar e reduzir o número de vezes que uma mensagem tem de ser transmitida.

4.4 Intercomunicação

De forma a realizar a intercomunicação entre membros existem várias possibilidades, no entanto, as mais utilizadas são *Apache Thrift, gRPC* ou então usar diretamente uma conexão *TCP* e gerir diretamente o envio de dados. De forma a simplificar e reutilizar conhecimento já existente na empresa, o método de comunicação escolhido é o *gRPC*.

O *gRPC* é um *framework* de comunicação remota de alta performance, este permite que aplicativos clientes e servidores troquem dados entre si de maneira rápida, confiável e eficiente, utilizando protocolos de comunicação padronizados e uma interface de programação simples e fácil de utilizar. O *gRPC* é baseado no protocolo *HTTP/2*, o que significa que este suporta funcionalidades avançadas, como *streaming* bidirecional e unidirecional, compressão de dados e multiplexação de pedidos. Este é frequentemente utilizado em sistemas distribuídos e em arquiteturas baseadas em microserviços para facilitar a comunicação entre diferentes componentes do sistema, outras funcionalidades deste *framework* podem ser consultadas no apêndice A. De forma a serializar os dados enviados, o *gRPC* utiliza *Protocol Buffers*. O *Protocol Buffers* (*Google Developers*, 2019) é uma tecnologia de serialização de dados também desenvolvida pela *Google*, que permite que estruturas de dados sejam definidas em um formato de linguagem neutra e compacta. Estas estruturas são então compiladas em código fonte para várias linguagens de programação, o que permite que as aplicações cliente e servidor possam facilmente trocar dados estruturados entre si. Adicionalmente, os dados serializados utilizando *Protocol Buffers* são geralmente menores e mais rápidos de serem processados do que outros formatos de serialização, como o *JSON* ou o *XML*. Por fim, o *Protocol Buffers* é amplamente utilizado em sistemas distribuídos, aplicativos móveis e outras aplicações de alta performance.

Internamente o *gRPC* utiliza *HTTP/2* e por consequência *TCP*, embora estes protocolos sejam eficientes, o *gRPC* permite mudar o método de transporte utilizado, utilizando essa funcionalidade, o transporte foi mudado para o protocolo *KCP* de forma a reduzir a latência da comunicação em troca de ser produzido mais tráfego de rede. A utilização deste protocolo vai ser utilizado de forma experimental. Caso seja encontrado algum problema então será revertido

para o transporte normal do *gRPC*. Adicionalmente, embora o protocolo utilize *UDP* este garante o envio de mensagens para os destinatários assim como *TCP*, assim sendo, a perda de informações não deverá ocorrer tal como se fosse utilizado o transporte por defeito do *gRPC*.

4.5 Distribuição

Tendo uma forma de saber quais membros estão presentes no *cluster* e forma de comunicação entre cada membro, é necessário estabelecer a forma como estes serão organizados. De forma a aumentar e evitar problemas de desempenho, o *cluster* não terá nenhum membro central que terá toda a responsabilidade ou que irá atribuir responsabilidades, em vez disso, cada membro vai ser responsável por um conjunto da carga a ser processada, e a designação de qual membro tem qual responsabilidade vai ser definida através do *hash ring*. Sendo um *channel* a parte onde irá ocorrer quase todo o processamento da aplicação, o nome deste em conjunto o nome do *hub* vão ser utilizados como chave para distribuição.

4.5.1 Hashing

O *hash* consiste numa função que mapeia dados de entrada com um tamanho variável em valor de saída de tamanho fixo, alguns exemplos comuns de algoritmos de *hash* incluem *MD5*, *SHA-1*, *SHA-256* e *SHA-512*. É possível utilizar uma função de *hash* de forma a mapear chaves a valores numa tabela, onde a função de *hash* é utilizada para calcular um índice para a chave que é utilizado para obter um valor armazenado numa tabela. As tabelas de *hash* são ideais para armazenar informação que precisa ser acessada rapidamente. Neste caso, uma tabela de *hash* poderia utilizar os identificadores dos membros como chaves, desta forma para calcular a qual membro um *channel* pertence o seguinte cálculo pode ser efetuado:

$$index = hash(channel)\ mod\ N$$

sendo *index* o membro, *channel* o identificador do *channel* e *N* o número de membros no *cluster*. Existem alguns problemas relacionados com colisões que não serão mencionados.

4.5.2 Distributed Hash Table

Uma tabela *hash* distribuída, ou *distributed hash table* (*DHT*), é uma extensão à tabela *hash* em que divide a informação em vários servidores. Esta utiliza um algoritmo de *hash* de forma a distribuir as chaves pelos membros do *cluster*, atribuindo a cada membro a responsabilidade de gerir um subconjunto das chaves e valores. Comparando com uma tabela de *hash*, a *DHT* oferece maior escalabilidade e tolerância a falhas. O principal problema com a *DHT* neste projeto, ocorre quando o número de membros do *cluster* é mudado, como por exemplo, devido

a uma falha, quando isso ocorre, todas as chaves têm de ser redistribuídas de forma a ter em conta um membro a menos. Vendo novamente o cálculo anterior vemos que o valor de *N* mudou, invalidando todos os cálculos previamente realizados. Recalcular todos os valores impacta significativamente o desempenho do *cluster*, visto que a informação tem de ser movida para os seus novos responsáveis, outro problema, é que embora somente um membro tenha falhado, os responsáveis serão muito provavelmente diferentes do que eram previamente. Vendo as tabelas 1 e 2, sendo a primeira a representação dos membros no *cluster* e a segunda os membros atribuídos a cada *channel* vemos como a distribuição está a ser realizada, no entanto, imaginando que o membro "Node 1" falha, o valor de *N* passa 2 e o índice dos membros desce em 1. O membro atribuído a cada *channel* terá de ser recalculado, com resultado como na tabela 3. Como pode ser observado, embora o balanceamento entre membros fosse o resultado o esperado, também podemos observar que os *channels* "test:channel_um", "test:channel_dois" e "test:channel_quatro" mudaram de responsável, mesmo estando o membro responsável por estes, ainda operacional. Em poucos valores como neste exemplo, o impacto seria negligível, mas em escalas de milhares e acima a redistribuição de todos estes membros pode tornar um *cluster* inoperacional enquanto processa todas estas alterações.

Tabela 1- Membros num cluster e seus índices

Índice	Membro
0	Node 1
1	Node 2
2	Node 3

Fonte: Própria

Tabela 2- Mapeamentos de channels para membros de um cluster com n = 3

Chave	Hash	Hash Mod N | N = 3
test:channel_um	12013487716029574172	2 (Node 3)
test:channel_dois	6072146722532578387	1 (Node 2)
test:channel_tres	5352869851951309179	0 (Node 1)
test:channel_quatro	6795858808070030270	2 (Node 3)

Fonte: Própria

Tabela 3- Mapeamentos de channels para membros de um cluster com n = 2

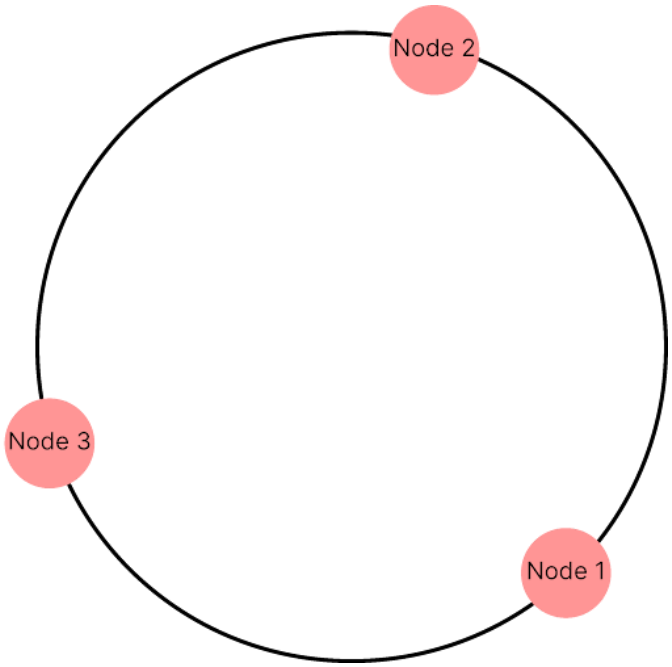
Chave	Hash	Hash Mod N | N = 2
test:channel_um	12013487716029574172	0 (Node 2)
test:channel_dois	6072146722532578387	1 (Node 3)
test:channel_tres	5352869851951309179	1 (Node 3)
test:channel_quatro	6795858808070030270	0 (Node 2)

Fonte: Própria

4.5.3 Hash Ring

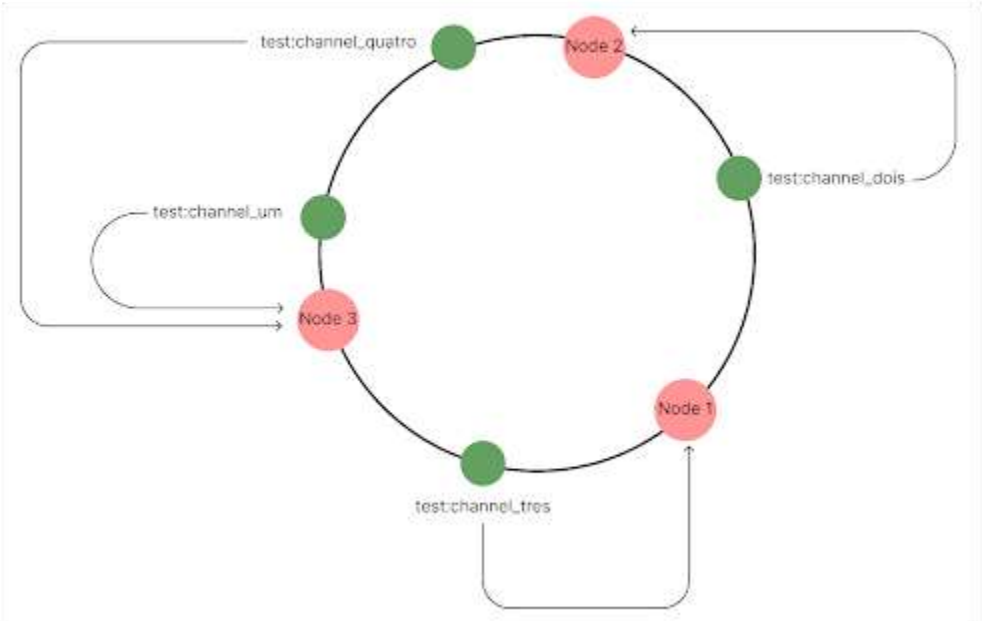
Portanto, de forma a evitar este problema, temos a técnica de anel de *hash*, ou *hash* ring, esta técnica forma um anel virtual (figura 11), em que cada membro é responsável por um intervalo contínuo de valores no anel. Exemplificando os cenários anteriores podemos ver na figura 12 como a distribuição dos *channels* é representada no *hash* ring. Portanto, tendo em conta o mesmo exemplo, podemos ver na figura 13 o resultado do mesmo cenário de falha do membro "Node 1". Como pode ser visto, somente um *channel* precisa de ser redistribuído, além de serem precisos menos redistribuições, também podemos somente recalcular os *channels* a que pertenciam aquele membro, tornando esta técnica ainda mais eficiente. Imaginando a situação em que um novo membro se junta ao *cluster* com o nome "Node 4" e a sua posição no anel é calculada entre o "Node 3" e "Node 2" podemos buscar todos *channels* a que o "Node 2" é responsável e recalcular o seu responsável, mais uma vez evitando recalcular todos os *channels*. Utilizando esta mesma técnica, temos a possibilidade de saber quem poderá ser o próximo responsável de um certo *channel*, algo que pode ser utilizado de forma criar um sistema de redundância.

Figura 11- Membros do cluster representados num anel virtual



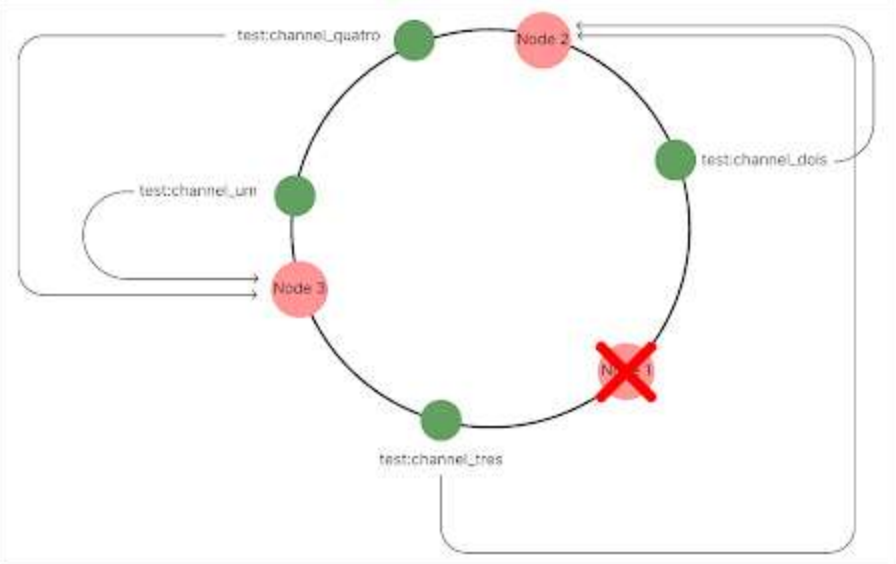
Fonte: Própria

Figura 12 - Anel virtual com membros de um cluster e channels



Fonte: Própria

Figura 13- Anel virtual com membros de um cluster e channels com a falha de um membro



Fonte: Própria

4.5.4 Consistência Eventual

Portanto, utilizando *gossip* e *consistent hashing* conseguimos atribuir *channels* para diferentes membros de forma a distribuir a carga pelo *cluster*, no entanto, tudo de forma eventualmente consistente. Tendo o *cluster* consistência eventual é importante perceber no que isto consiste. Portanto, a consistência eventual é um modelo de consistência em sistemas distribuídos em que as atualizações feitas em um membro são propagadas para os outros membros do *cluster* em um período não imediato, ou seja, pode haver um certo atraso até que todos os membros recebam a mesma atualização. Este modelo de consistência é utilizado em sistemas que podem lidar com uma pequena inconsistência temporária na informação, mas que ainda garantem que eventualmente todos os membros terão a mesma informação. Assim sendo, a consistência eventual pode apresentar alguns problemas, como a possibilidade de leituras inconsistentes entre membros, ou seja, um membro pode ter informações mais atualizadas do que outro. No caso deste projeto, visto não se tratar de um sistema que gere informação e onde todas as informações importantes são armazenadas numa base de dados, a consistência eventual não é problemática, e é utilizado como ferramenta para aumentar a escalabilidade do sistema.

4.6 Novo sistema

Tendo agora as partes fundamentais do sistema, com o consenso a ser resolvido com o protocolo *gossip*, a intercomunicação com o *framework gRPC* e a distribuição utilizando a junção do *gossip* e *hash ring*, é importante perceber como estes irão funcionar em conjunto.

Em primeiro lugar, temos a parte responsável por chegar ao consenso de quantos membros existem no *cluster*, esta parte assim como previamente referida é gerida pelo protocolo *gossip*. Portanto, sempre que um novo membro se junta ou sai do *cluster*, este irá refletir no *hash ring*. Lembrando, que no *hash ring* vão ser mapeados todos os identificadores dos membros do *cluster*. Assim sendo, quando precisamos de distribuir um *channel* ou localizá-lo, será calculada a localização do *channel* no *hash ring* utilizando o identificador deste. Sabendo a posição no *hash ring*, podemos facilmente calcular a qual membro o *channel* pertence. Portanto, quando um membro sai ou se junta, a sua posição será adicionada ou removida do *hash ring* e potencialmente será necessário recalcular a quais membros os *channels* pertencem.

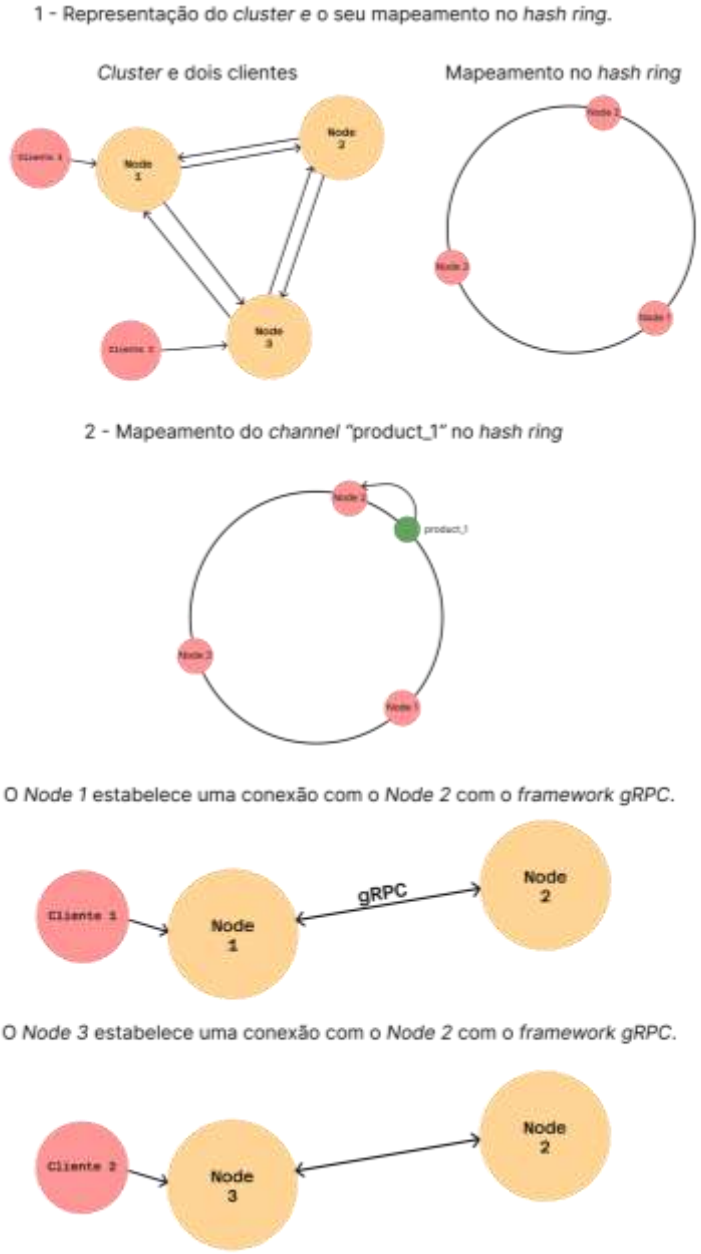
O ponto de intercomunicação, neste sistema é introduzido quando é necessário enviar informação entre membros, por exemplo, publicar um evento num *channel*, exige que um pedido seja feito ao membro responsável por este, ou seja, uma conexão será criada ou reutilizada ao membro destino onde será enviado o pedido para publicar o evento. Assim como mencionado, esta intercomunicação será realizada com o *framework gRPC*, de forma a saber os endereços para qual a conexão será criada, será utilizado o protocolo *gossip* para descobrir esses endereços.

Portanto, o protocolo *gossip* é o elemento principal destes três pontos, este em junção com os outros dois pontos permite ter bases para a criação de um sistema distribuído.

Exemplificando o funcionamento destes componentes temos a figura 14 e figura 15. Nesta figura temos um *cluster* com os *Nodes* 1, 2 e 3, sendo que no *Node* 1 está conectado o Cliente 1 e no *Node* 3 está conectado o Cliente 2, assim como representado na parte esquerda da primeira etapa da figura. Na parte direita, temos o mapeamento dos 3 *Nodes* representada no *hash ring*. Neste exemplo, o Cliente 1 subscreve ao *channel* “product_1” , portanto o *Node* 1 calcula no *hash ring* a posição deste *channel* que resulta no *Node* 2 assim como representado na segunda etapa. Sendo o *Node* 2 o resultado obtido, significa que este é o responsável pelo *channel* “product_1”, portanto, o *Node* 1 vai estabelecer uma conexão bidirecional utilizando o *framework gRPC* com o *Node* 2, assim como pode ser visto na terceira etapa. Assim que a conexão é estabelecida e a intenção de subscrever é enviado para o *Node* 2, este cria o channel (caso não exista), adiciona o Cliente 1 à lista de subscritos e vai enviar todos os futuros eventos

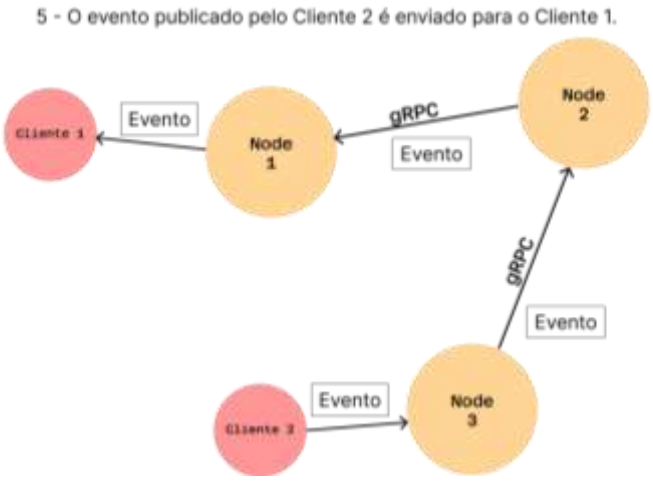
que receber para o *Node* 1 e este para o Cliente 1. Na quarta etapa, o Cliente 2 pretende enviar um evento para o mesmo *channel*, para isso, o *Node* 3 repete o mesmo processo anterior e estabelece uma conexão com o Cliente 2. Por fim, na quinta etapa o *Node* 3 envia o evento recebido para o Node 2, que por sua vez envia para o Node 1 e que por fim envia para o Cliente 1.

Figura 14 - Exemplo de funcionamento parte 1



Fonte: Própria

Figura 15 - Exemplo de funcionamento parte 2



Fonte: Própria

Desta forma temos os 3 componentes em funcionamento, protocolo *gossip* para manter uma lista de *Nodes* ativos, o *hash ring* para distribuir os *channels* pelos *Nodes* e o *framework* para *gRPC* para comunicar entre os membros do *cluster* ou *Nodes*.

Existe ainda um ponto, relativamente à inicialização do *cluster*, ou como este é primeiramente formado, sendo que a mutação deste é gerida também pelo protocolo *gossip*, que será mencionado mais à frente.

4.7 Funcionamento

Nesta parte, será explicado em mais detalhe o funcionamento dos componentes da aplicação, desde a sua inicialização até a sua terminação, começando pela inicialização da aplicação e como um *cluster* é iniciado, seguido os seus componentes mais importantes e funcionalidades destes.

4.7.1 Inicializar

Para iniciar um *cluster* é necessário a existência de mais que uma instância da aplicação. A uma destas instâncias é indicado os endereços de rede da outra, para que uma conexão seja estabelecida. Assim que estabelecida, ambas instâncias formam um *cluster* de acordo com o protocolo *gossip*, e novas instâncias têm de se juntar ao *cluster* utilizando o mesmo processo.

Para este processo, é necessário o conhecimento dos endereços de rede das novas instâncias da aplicação, algo que costuma ser gerido por um *service discovery*, ou descoberta de serviços. Este é um serviço utilizado em arquiteturas de sistemas distribuídos para encontrar e se conectar a serviços disponíveis numa rede, este tem um endereço de rede conhecido por todas

aplicações que o usam para registarem a sua presença e publicarem informações sobre si, tornando mais fácil para outros serviços localizá-los e se comunicarem com eles. Isto permite que os sistemas distribuídos sejam mais flexíveis, escaláveis e resilientes, uma vez que os serviços podem ser facilmente adicionados ou removidos sem afetar a operação geral do sistema.

Na empresa não existe um *service discovery*, visto que o *NATS* serve como um serviço central que permite a comunicação entre serviços, evitando assim a necessidade de um *service discovery*. Portanto, para esta aplicação um *service discovery* seria útil, mas visto a inexistência de um, foram utilizados meios mais simples de forma a descobrir outras instâncias desta aplicação.

AWS ECS

O serviço *AWS ECS* (apêndice B), é onde a aplicação vai ser executada, utilizando *Docker containers*. Este serviço oferece uma *API*, que permite que sejam consultadas informações sobre as instâncias em execução. Portanto, quando uma nova instância é criada, esta utiliza esta *API*, para consultar todas as placas de rede do mesmo tipo da aplicação, retornando assim os endereços de rede a que estas é atribuído, com estes endereços a aplicação realiza um pedido para se juntar ao *cluster*, que será aceite caso as credenciais da nova instância estejam corretas.

UDP

Para ambientes locais de desenvolvimento, é utilizado o *UDP Broadcast*. Sendo isto uma técnica de comunicação em rede que envia mensagens de um emissor para vários dispositivos, sem que o emissor precise saber exatamente quem são esses dispositivos ou onde estes estão localizados na rede. Nesse método, o emissor envia uma mensagem de difusão (*Broadcast*) para um endereço *IP* especial, que é reconhecido por todos os dispositivos conectados na rede. Assim, todos os dispositivos conectados na rede que estão à escuta nesse endereço *IP* especial, podem receber a mensagem enviada pelo emissor. Quando os outros membros recebem a mensagem enviada, estes podem anunciar sua presença na rede e permitir que outros membros os descubram de maneira fácil e rápida.

4.7.2 Hub

Um *hub* é um elemento que representa um *tenant* na aplicação, cada *Hub* tem as suas próprias configurações, método de autenticação, *channels*, *namespaces* e é o elemento que agrupa as sessões de clientes para cada *tenant*. Adicionalmente, cada *hub* é representado por

um identificador de texto que seja codificável em *UTF-8*. De forma a evitar criar cada *hub* explicitamente, este pode ser criado de forma dinâmica pelo cliente ou serviço utilizando configurações por defeito.

Ao iniciar um *hub* as suas configurações são consultadas à base de dados, e caso não existam estas são criadas de acordo com as configurações por defeito. As configurações por defeito são definidas através de um ficheiro de configuração necessário para inicializar a aplicação. Após o *hub* ser criado e inicializado, este finalmente está pronto para criar sessões e *channels*. De forma a evitar que um *hub* que não esteja a ser utilizado se mantenha ativo em memória, é definido um intervalo de tempo que começa sempre que o número de sessões no membro em questão chegue a zero, adicionalmente, caso uma nova sessão seja criada, o intervalo de tempo é cancelado e o *hub* permanece ativo. Caso o intervalo de tempo termine, o *hub* é terminado. Durante a sua terminação qualquer nova sessão ou criação de *channel* que pertença a este *hub* será posto em espera até que o processo termine. A terminação do *hub* é praticamente instantânea, no entanto, existe a possibilidade de que um pedido para a criação de um *channel* ou sessão seja recebido durante a sua terminação. Nestes casos, o *hub* é primeiro terminado e nova instância é de seguida criada para responder a estes pedidos.

4.7.2.1 Configurações

Cada *hub* tem um conjunto de configurações que serão utilizadas para cada sessão a que este pertence. Adicionalmente, este também define um conjunto de regras para os *channels* que ainda não tenham configurações definidas.

Default Public

Começando pela configuração *Default Public*, este apenas define se todos os *channels* que pertencem a este *hub* são de acesso público por defeito, ou seja, este não necessita que o utilizador esteja autenticado ou que tenha autorização definida para aceder a este. Esta configuração permite que novos utilizadores ainda não registados sejam capazes de subscrever a informação recebida no *channel* em questão. Esta configuração pode ser ignorada caso estejam definidas regras mais específicas para esta *channel*.

Allow Anonymous

Embora a configuração *Default Public* permita que utilizadores não autenticados tenham acesso aos *channels*, existe a possibilidade que o *Hub* não queira permitir utilizadores não autenticados, é nestes casos que a funcionalidade *Allow Anonymous* pode ser ativada. Portanto,

um *Hub* pode permitir somente utilizadores autenticados enquanto define channels de acesso público, permitindo que qualquer sessão autenticada tenha acesso a estes *channels* sem que uma permissão explicita seja definida.

User Channel

A configuração *User Channel* permite que o *hub* crie automaticamente um *channel* para uma sessão que esteja autenticada. Este *channel* é partilhado por todas sessões que pertençam ao mesmo utilizador. O identificador do *channel* é definido utilizando um *namespace* de utilizador que por defeito é "u" e a junção do identificador do utilizador com um separador ":" no meio, ou seja, um utilizador com o identificador "user_123" resultaria no identificador "u:user_123". Este *channel* pode ser utilizado quando é necessário que eventos sejam publicados para um utilizador em específico, como por exemplo, uma mensagem. Sendo que este *channel* é partilhado por todas as sessões do mesmo utilizador, existe a possibilidade que um dos dispositivos emita eventos para os outros. Um exemplo da utilização desta funcionalidade, passa por um utilizador com uma *wishlist* (lista de produtos desejados), onde num dos dispositivos o utilizador adiciona ou remove um item da sua *wishlist* e um evento com esta alteração é enviado para os outros dispositivos, estes por sua vez podem atualizar a sua informação local da *wishlist* de acordo com o evento recebido, permitindo desta forma a sincronização em tempo real da *wishlist*.

Default Rules

Por fim, temos o *Default Channel Rules,* este é um identificador para as definições a serem utilizadas por defeito em *channels* sem configurações definidas. As configurações disponíveis para cada *Channel Rule* serão mencionadas durante os *channels*. Portanto, em vez de definir configurações para todos os *channels* é possível escolher configurações que vão ser aplicadas a todos os *channels* pertencentes ao *hub*, no entanto, caso configurações mais especificas estejam definidas essas vão se sobrepor a esta.

4.7.2.2 Atualizações

Cada membro do *cluster*, mantém as configurações do *hub* em memória de forma a reduzir o número de consultas à base de dados e a reduzir o tempo que demora a criar uma sessão. Visto o *hub* ser somente responsável por agrupar *channels*, sessões e configurações, não é necessária coordenação deste entre os membros do *cluster*. No entanto, sempre que houver uma atualização, é necessário que todos os membros do *cluster* atualizem as suas configurações em

memória. Assim sendo, sempre que uma alteração às configurações de um *hub* ocorra, o protocolo *gossip* é utilizado para propagar a informação de que a configuração foi modificada. Por fim, os recetores por sua vez têm a responsabilidade de realizar uma consulta à base de dados de forma a buscarem as configurações mais recentes. Se por algum motivo, o membro do *cluster* não receber uma mensagem a notificar que as configurações foram alteradas, existe um temporizador configurável, que quando este termina o *hub* volta a consultar as suas configurações e atualiza para as mais recentes.

4.7.3 Channel Rules

De forma a definir as configurações diferentes entres *channels* e *namespaces* existe um objeto que define todas configurações possíveis de um *channel*, com o nome *Channel Rules*, que também será referenciado como configurações. Cada configuração tem como identificador um valor de texto em conjunto com o identificador do *hub* a que pertence, ou seja, uma chave composta. Utilizando uma chave composta, é possível utilizar nomes iguais de configurações entre *hubs*, isto facilita a criação de configurações e manutenção destas.

4.7.3.1 Funcionalidades

As configurações possíveis poderão não ser suportadas por todos tipos de *channels*, algo que será explicado durante a informação sobre *channels*, assim sendo, existem as seguintes configurações disponíveis:

- *Allow Retain Message*;
- *Store Message*;
- *Push Message*;
- *Presence*;
- *Public*;
- *Client Publish*;
- *Allow Anonymous*;
- *Occupancy*.

As configurações *Public* e *Allow Anonymous* funcionam exatamente da mesma forma como as configurações no *hub*, no entanto têm como alvo um *channel* em específico ou num *namespace*. O resto das funcionalidades serão mencionadas a seguir em conjunto com os *channels*.

4.7.4 Channel

Um *channel* é o elemento onde as informações são recebidas, processadas e enviadas. Este é o elemento que é coordenado entre os membros do *cluster* utilizando *consistent hashing*. Este é semelhante a um tópico em outras aplicações *PubSub*, no entanto, neste caso o *channel* pode não ser somente um tópico para *PubSub*.

Em geral um *channel* só existe num único membro do *cluster* ao mesmo tempo, no entanto, em certos casos existe a possibilidade que mais que uma instância do mesmo channel esteja ativa. Um exemplo deste caso pode ser observado quando um novo membro é adicionado ao *cluster*, neste caso existe a possibilidade de que este seja o novo responsável pelo *channel*, e enquanto a informação de que o membro foi adicionado ao *cluster* não seja propagada para o antigo responsável pelo *channel* este vai continuar a assumir a sua responsabilidade por este. Todos os membros que tenham conhecimento do novo responsável pelo *channel* vão enviar eventos para este, enquanto membros que não tenham ainda recebido essa informação vão enviar para o antigo responsável. Caso o antigo responsável já tenha recibo a informação este irá recusar todos os eventos que receba para o *channel* em questão.

Em casos em que o membro falha, todos os *channels* a que este era atribuído irão fica temporariamente indisponíveis até que a sua falha seja propagada pelo *cluster* e um novo responsável seja atribuído.

4.7.4.1 Inicialização

A inicialização de cada *channel* funciona de forma similar a de um *hub*, é realizada uma consulta para as configurações do *channel* que é representada pelas configurações previamente mencionadas, que são também mantidas em memória. Posteriormente, é iniciado um ciclo para processar mensagens em espera e é iniciado um processo similar ao de terminação de um *hub*.

4.7.4.2 Atualização

Tendo em conta que um *channel* somente utiliza as configurações de um *Channel Rules* é necessário que sempre que este seja atualizado que essa informação seja propagada utilizando o mesmo mecanismo que é utilizado para as configurações de um *hub*. Por fim, quando o membro receber essa informação, este irá consultar a base de dados para atualizar para a nova informação e irá atualizar todos *channels* que estejam a utilizar o *Channel Rules*. Na eventualidade de ser aplicada uma configuração para um *channel* em específico, então utilizando *gRPC* o membro responsável pelo *channel* será notificado da atualização. Para

namespaces o mecanismo de propagação também é utilizado, e todos os *channels* do *hub* que tenham o *namespace* irão reavaliar qual *Channel Rules* será utilizado. Em certos casos, existe a possibilidade de que um membro não receba a informação devido a problemas de rede. De forma a garantir que eventualmente a informação é atualizada, cada *channel* inicia um intervalo de tempo interno, que sempre que termina é consultada novamente a base de dados de forma a buscar a última informação e é novamente começado o intervalo de tempo.

4.7.4.3 Processamento de mensagens

As mensagens nos *channels* são processadas serialmente de forma *FIFO* (*First In, First Out*), ou primeiro a entrar, primeiro a sair. Embora o processamento de todos *channels* seja de forma concorrente, o processamento de mensagens de cada *channel* ocorre de forma sequencial, permitindo manter a ordem das mensagens. No entanto, devido à consistência eventual do *cluster*, a ordem de mensagens pode não se manter nos primeiros tempos após a alteração dos membros do *cluster* devido a possibilidade de existência de 2 instâncias de um *channel* no *cluster* assim como previamente mencionado.

4.7.4.4 Tipos de *channels*

Sendo o *channel* o elemento distribuído pelo *cluster*, este vai ser utilizado para implementar diferentes funcionalidades que usam os mesmos mecanismos de distribuição, reduzindo a complexidade de manter várias implementações de distribuição. Atualmente, existem 3 tipos de *channels*, sendo possível adicionar mais. Durante a implementação de um tipo de *channel*, pode ser decidido não implementar algumas funcionalidades definidas no *channel rules*, seja por não ser aplicável ou por não ser necessário.

4.7.4.5 Default

O tipo de *channel Default* ou normal, consiste num simples tópico *PubSub*, onde mensagens podem ser publicadas e distribuídas por todos os clientes interessados. Estas mensagens são processadas de forma sequencial sempre que possível. Este tipo de *channel* é capaz de utilizar todas as funcionalidades definidas no *channel rules*.

4.7.4.6 Document

O tipo de *channel Document*, tal como o nome indica, consiste num documento com estrutura similar a *JSON*, onde clientes podem pedir para realizar operações que depois são transmitidas para clientes interessados no *channel*. As operações no documento são baseadas

no *RFC 6902 (IETF*, 2013*)*, nomeado de *JavaScript Object Notation (JSON) Patch*, adicionalmente, o tipo de dados binário é permitido neste documento ao contrário do formato *JSON*. Embora seja baseado num documento *JSON*, esta versão utiliza *Protocol Buffers* para que possa ser serializado em binário de forma eficiente e mais compacta. Este tipo de *channel*, permite que múltiplos clientes tenham um conjunto de dados sincronizados enquanto estes são alterados. De forma reduzir a quantidade de informação a ser enviada por rede, em cada alteração realizada ao documento, somente as alterações são enviadas e cabe ao cliente aplicar as alterações à sua versão do documento local.

Cada documento é armazenado na base de dados, a cada intervalo de tempo de configurável ou quando o *channel* é terminado. O documento é serializado em binário, comprimido e por fim armazenado. No caso de o membro falhar, qualquer alteração não guardada irá naturalmente ser perdida. De forma a evitar perder alterações, todas as alterações não guardadas serão armazenadas numa lista na aplicação *Redis* para que quando o *channel* volte a ser inicializado este seja capaz de reconstruir o documento até ao estado anterior.

O documento suporta as seguintes operações: *add, remove, replace, move* e *copy*. A operação *add*, tal como o nome indica consiste em adicionar uma propriedade ao documento. O *remove*, remove uma propriedade, *replace* substituir uma propriedade e move e *copy* consiste em utilizar uma propriedade do documento e mover ou copiar para o novo destino.
Cada operação no documento pode ser representada com as seguintes propriedades:

- *op* - O tipo de operação a ser aplicada;
- *path* - O caminho no documento onde a operação será aplicada, por exemplo, o *path* "a/b/c" define que existe uma hierarquia onde *a* é o nível acima de *b* e *b* de *c*. Adicionalmente, tendo *a* e *b* propriedades associadas estas serão convertidas, caso necessário, na representação de um objeto em *JSON*. De forma a trabalhar com listas, índices podem ser utilizadas no *path*, por exemplo, o *path* "a/0" define que *a* é uma lista e a operação será realizada no índice 0, mais uma vez, *a* será convertido numa lista caso necessário;
- *from* - Em alguns comandos como o *copy* e *move* é necessário providenciar um caminho de fonte e um caminho de destino, sendo esta a fonte e o *path* o destino;
- *value* - Por fim, o *value* representa o valor a ser utilizado, este pode ser qualquer tipo permitido pelo documento incluindo objetos complexos com múltiplos valores.

Cada pedido de alteração ao documento pode conter várias operações, e são estas que são transmitidas aos outros clientes para que estes as apliquem localmente. Este documento é acompanhado de um número incremental que representa a sua versão que é atualizado por cada pedido de alteração realizado, este é utilizado para que os clientes consigam perceber se perderam alterações e para que possam comparar com a sua versão local. Por fim, este tipo de documento também suporta a funcionalidade de *PubSub* do *channel Default*.

4.7.4.7 Notification

O tipo de *channel Notification*, tal como o nome indica, tem como objetivo gerir as notificações. Este permite a criação de notificações e gerir o seu estado de leitura de forma não individual, ou seja, todos clientes no *channel* partilham o mesmo estado de leitura. A criação de notificações não pode ocorrer através de clientes, somente através de outros serviços autenticados, no entanto, o cliente tem a capacidade de marcar as notificações como lidas. Este *channel* gere o número de notificações não lidas, enviando o número para o cliente sempre que este seja alterado, ou seja, caso o *channel* tenha 2 notificações não lidas, esta informação será enviada para o cliente, caso o cliente marque uma como lida, então o *channel* envia a informação de qual notificação foi lida para que atualizem a sua informação local e volta a enviar a quantidade de notificações não lidas. Ao contrário dos outros tipos de *channels*, a grande parte das funcionalidades do *channel rules*, não são suportadas, sendo possível somente utilizar as funcionalidades *Public* e *Allow Anonymous*.

4.7.4.8 Extensão

Assim como visto nos tipos de *channels* anteriores, é possível reutilizar o método de distribuição do *channels* e criar um tipo que se adapte à situação necessária. Por exemplo, um *channel* para conversações ou para rastreamento de encomendas pode ser criado utilizando funcionalidades já existentes e adicionando específicas para o caso necessário. Adicionalmente, devido às funcionalidades desenvolvidas de forma modular em cada *channel* é possível criar um *channel* que suporte todas funcionalidades previamente mencionadas ao mesmo tempo, um exemplo disto pode ser observado no tipo de *channel Document*, onde todas as funcionalidades do *Default* estão presentes enquanto adiciona a funcionalidade do documento.

4.7.4.9 Funcionalidades

Utilizando as *channel rules* é possível definir quais funcionalidades devem estar ativas num *channel*, no entanto, nem sempre o *channel* tem suporte para as funcionalidades, como previamente mencionado. Atualmente existem 9 funcionalidades definidas, e com suporte para a adição de novas.

Retain Message

A funcionalidade, *retain message* faz uma cópia da última mensagem marcada para ser retida e armazena-a em memória local e na aplicação *Redis* para que possa ser recuperada. Esta mensagem, é depois enviada sempre que um cliente subscreva a este *channel*. Esta funcionalidade é baseada na funcionalidade do protocolo *MQTT*, esta é útil para casos em que é necessário que novos clientes que subscrevem ao *channel* tenham a última informação publicada no *channel*. Um exemplo simples para esta funcionalidade passa por ter um *channel* para receber o stock atual de um produto, desta forma o cliente sabe qual a última atualização de stock e irá receber novas atualizações.

Store Message

Em certos casos é necessário que algumas mensagens fiquem armazenadas para serem acedidas posteriormente, assim, com esta funcionalidade, qualquer mensagem enviada marcada para ser armazenada irá ser primeiro armazenada na base de dados e só depois enviada para os clientes. Esta funcionalidade é bastante útil quando é preciso manter um histórico de mensagens, como por exemplo, um *chat* ou então manter um registo para auxiliar a depuração de um problema.

Push Message

Esta funcionalidade está mais relacionada com aplicações móveis embora também funcione com aplicações *Web*. Esta consiste em enviar uma notificação *Push* utilizando as plataformas nativas da *Apple* (*APNS - Apple Push Notification Service*) e *Google* (*FCM - Firebase Cloud Messaging*) para os dispositivos móveis, e aplicações *Web* com o *FCM*. Assim como as outras funcionalidades, esta é ativada quando uma mensagem marcada com esta funcionalidade é recebida, embora a notificação seja enviada, a mensagem continua a ser enviada para os clientes subscritos ao *channel*. Ao contrário das outras funcionalidades, o envio da notificação não é garantido ficando ao cargo das plataformas o seu envio.

Presence

Esta funcionalidade permite o rastreamento da presença das sessões subscritas num *channel*. Sempre que uma nova sessão se subscreva ao *channel*, esta recebe o estado atual da presença de todas as sessões atualmente subscritas, enquanto as outras sessões subscritas no *channel* recebem um evento de que uma nova sessão subscreveu ao *channel*. Após ter recebido estado inicial, a sessão só irá receber alterações que ocorram, como uma nova sessão subscreveu ou uma sessão removeu a sua subscrição. A presença de cada cliente é definida pelo seu identificador de sessão, identificador de utilizador caso existente, *metadados* definidos durante a autenticação e uma *timestamp* de quando se subscreveu. Esta informação é gerida dentro do *channel*, em memória e não é armazenada, portanto, caso o membro do *cluster* falhe esta informação tem de ser reconstruída na nova instância do *channel*.

Public

A funcionalidade *Public* é igual à previamente descrita no *hub* como *Default Public*, sendo que esta se aplica num nível diferente. Esta define que os *channels* afetadas pela configuração são de acesso público, ou seja, qualquer sessão pode se subscrever aos *channels* sem ter permissões definidas.

Client Publish

Esta funcionalidade define se as sessões podem publicar informação no *channel*. Em casos em que o channel é publico, nem sempre existe o interesse de permitir que as sessões publiquem eventos neste. Nesses casos, a publicação de eventos no channel pode ser desativada para todas sessões, sem precisar de definir permissões especificas na autenticação de uma sessão. É possível sobrepor a esta configuração caso a sessão tenha permissão definida para escrita neste *channel*, dando assim a capacidade a somente algumas sessões de publicarem eventos.

Allow Anonymous

Novamente, esta funcionalidade, simplesmente define se utilizadores não autenticados podem aceder a este *channel*. Esta funcionalidade é útil quando é necessário ter um *channel* público com a funcionalidade *public*, mas que só utilizadores autenticados têm autorização.

Occupancy

A funcionalidade *occupancy* permite rastrear a quantidade de clientes subscritos ao *channel*, é uma versão mais leve do que a funcionalidade *Presence* sendo que esta só gere um contador em vez de uma lista de sessões. Esta funcionalidade pode ser utilizada, em casos em que é necessário mostrar quantos utilizadores estão neste momento a ver um produto ou informação, enquanto consome menos recursos do que a funcionalidade *Presence*. Ter ambas funcionalidades ativas é completamente redundante visto ser possível calcular o número de cliente com a funcionalidade *Presence*.

Channel Live History

Outra funcionalidade aplicada apenas a *channels default*, e opcionalmente desativada nas configurações da aplicação é a *Live History*, esta mantém as últimas mensagens enviadas nos *channels* em memória. Esta funcionalidade, tem como objetivo prevenir a perda de informação durante desconexões rápidas, principalmente em dispositivos móveis, por exemplo, durante a troca de rede ou durante passagem num túnel. O número de mensagens armazenadas e a ativação da funcionalidade podem ser definidas nas configurações da aplicação.

Terminação

Um *channel* naturalmente consome recursos de memória e de processamento, portanto, de forma a reduzir os *channels* que estejam sem uso, um mecanismo similar ao do *hub* é utilizado. No caso do *channel*, sempre que o número de sessões subscritas chegue a 0 um intervalo de tempo configurável começa e caso não ocorra nenhuma nova subscrição o *channel* é terminado limpando toda a memória local incluindo as mensagens da funcionalidade *live history*.

4.7.5 Namespace

O *namespace* é uma forma de agrupar *channels* a um conjunto de regras, em vez de definir um *channel rules* para cada *channel* individualmente. É possível definir um *namespace* atribuir-lhe um *channel rules* e todos os *channels* dentro do *namespace* utilizam o mesmo *channel rules*. Um *namespace* é somente aplicável no *hub* a que este pertence, podendo existir configurações diferentes para o mesmo identificador de *namespace* em vários *hubs*.

Um *channel* é considerado pertencente a um *namespace* caso o seu identificador comece com o identificador do *namespace*. Por exemplo, um *channel* com identificador “product:id_1_stock” é considerado como pertencente ao *namespace* “product”, sendo

utilizado ":" como separador. Somente o texto até ao primeiro separador é avaliado como *namespaces*, caso um separador não esteja presente no *channel*, então este não pode pertencer a um *namespace*. Esta funcionalidade é especialmente importante em casos em que o *channels* podem ter nome dinâmicos, mas é necessário que estes contenham as mesmas regras, utilizando o exemplo descrito podemos definir um conjunto de regras para todos os *channels* com informação de stock dos produtos sem precisar de definir cada um individualmente.

4.7.6 Regras de *Channel*, *Namespace* e de *Hub*

Existindo *channel rules* para *channels*, *namespaces* e *hubs* é necessário definir quais *channel rules* se aplicam quando vários estão presentes a diferentes níveis. Assim sendo, o objetivo é seguir sempre a regra mais específica sendo a ordem de prioridade *channel*, *namespace* e por fim *hub*, ou seja, caso um *channel* tenha regras definidas para o seu identificador estas têm prioridades perante as definidas no *namespaces* e *hub*. Caso não exista para o *channel*, mas exista para o seu *namespace* então esta tem prioridade perante as do *hub*, por fim, não existindo mais níveis, qualquer *channel* que não tenha regras definidas para o seu identificador ou *namespace* pertencente irá utilizar as regras definidas no *hub*.

4.7.7 Auth Provider

De forma a permitir que cada *tenant* tenha o seu próprio meio de autenticar os seus clientes, foi criado o conceito de *Auth Provider*. Este consiste em um conjunto de configurações que define como o processo de autenticação deve ocorrer. Assim que um cliente pedir para se autenticar, a informação que o cliente envia é redirecionada por *HTTP* ou *NATS* para o destino configurado, que por sua vez deve responder com estado de sucesso, identificador do utilizador, meta dados do utilizador e permissões. Nas configurações disponíveis é possível definir o método de redireccionamento estando disponíveis as opções *HTTP* e *NATS*, na mensagem de redireccionamento é enviado o identificador do *hub* e da sessão. Adicionalmente, é possível definir cabeçalhos a serem enviados na mensagem. Portanto, cada *hub* pode somente utilizar um *Auth Provider*. Por defeito um *hub* é criado com um *Auth Provider* que utiliza um caminho que inclui o seu nome e utiliza *NATS* como meio de redireccionamento.

4.7.8 Session

A *session* ou sessão é o elemento que gere a conexão com o cliente, este processa as mensagens recebidas pelo cliente, gere o *heartbeat* da conexão e verifica as permissões do utilizador antes de realizar ações. Um utilizador pode ter várias sessões tendo estes um

identificador próprio, as sessões podem ser autenticadas e não autenticados, sendo possível autenticar posteriormente.

Este elemento foi desenvolvido de forma a ser agnóstico ao tipo de comunicação utilizado, permitindo que vários protocolos sejam implementados como *WebSocket*, *SSE*, *gRPC* e *TCP*. Adicionalmente, de forma a garantir a estabilidade e o bom funcionamento da aplicação um *rate limit* é aplicado individualmente a cada sessão independente de quantas conexões simultâneas um utilizador tenha. O *rate limit* é aplicado à quantidade de mensagens recebidas por segundo, sendo a quantidade configurável, mensagens como *heartbeat* não são tidas em conta visto estas serem necessárias para garantir que a conexão se mantenha aberta. De forma a reduzir a quantidade de *heartbeats* necessários, sempre que um cliente envia uma mensagem, independentemente de que tipo seja, esta conta como um *heartbeat* e por consequência adia o envio da próxima verificação.

4.7.8.1 Tipo de conexão

Assim como mencionado anteriormente, a sessão é agnóstica ao protocolo de comunicação utilizado e permite que vários protocolos sejam implementados. Atualmente, somente dois foram implementados, sendo estes *WebSocket* e *SSE*. Em ambos os casos toda a informação transmitida está em formato binário, no caso de *WebSocket* é possível enviar e receber mensagens do cliente, no caso do *SSE* é apenas suportado enviar mensagens para cliente, sendo necessário que o cliente envie quais *channels* pretende subscrever quando estabelece a conexão.

4.7.8.2 Protocolo

De forma ao cliente comunicar com aplicação, foi estabelecido um protocolo de mensagens para que ambas as partes saibam que tipo de mensagens existem, quais ações são possíveis e permitir que ambas saibam o formato das mensagens com antecedência.

Início com *JSON*

Inicialmente o protocolo de mensagens foi definido utilizando o formato de serialização *JSON* aproveitando as habilidades dinâmicas do mesmo. No entanto, em certos casos o envio de mensagens com conteúdo em binário era necessário, e de forma a enviar este tipo de dados em *JSON* este teria de ser convertido em *base64*. Infelizmente, converter dados binários em *base64* além de ter um custo de desempenho envolvido, este também aumenta o tamanho do

mesmo conteúdo, sendo o aumento de cerca de mais 33% do tamanho original. O valor de aumento pode ser calculado utilizando a seguinte fórmula:

$$ceil\big((tamanho \times 8) \times 6\big) - (tamanho \times 8)$$

Sendo o "tamanho" o número de *bytes*, portanto, num exemplo de 1000 *bytes* o valor final seria aproximadamente 13330 *bytes*.

Transição para *Protobuf*

Portanto, mantendo as mensagens, o formato de serialização foi convertido para protobuf, assim, passa a ser possível o envio de dados em binário sem passos intermediários. Outras vantagens passam por melhor performance de serialização e com tamanhos finais mais pequenos do que seria possível em *JSON*, além de ser possível utilizar as definições em *protocol buffers* de forma a evitar problemas na evolução das propriedades das mensagens e também a validar o tipo dos dados durante a descodificação.

Estrutura de envelope

No protocolo de mensagens, todos os tipos de mensagens ou eventos são enviados numa estrutura comum nomeada de envelope, este envelope contém somente duas propriedades: tipo de evento e conteúdo. O tipo de evento é representado por uma enumeração e o conteúdo é simplesmente um conjunto de *bytes*. Em certos casos é necessário relacionar uma mensagem com outra, por exemplo, vários pedidos podem ser realizados onde se espera uma resposta de cada um destes, e por muitas vezes as respostas podem não ser recebidas da ordem em que os pedidos foram realizados. De forma, a permitir o cliente associar uma resposta com o pedido realizado, as mensagens incluem a possibilidade de enviar um número inteiro como identificador de pedido, que será devolvido ao cliente com a resposta. Este identificador é transparente para aplicação e em certos pedidos até são opcionais, nesses casos, quando o cliente não preenche o identificador de pedido o servidor não envia uma resposta.

Tipos de eventos

Para cada tipo de mensagem está definido um evento, no entanto, um evento pode representar conteúdos diferentes de acordo com quem envia e quem recebe, por exemplo, quando uma mensagem com o tipo de evento *Auth* é enviada pelo cliente, é assumido que o cliente está a realizar um pedido de autenticação, enquanto quando uma mensagem com o mesmo tipo de evento é enviada pela aplicação é assumido que esta é uma resposta ao pedido anterior. Em primeiro lugar temos os tipos de evento base de uma conexão, estes são: *Ping*,

Pong e *Close*. Os tipos de evento *Ping* e *Pong* são sempre usados em conjunto sendo geralmente o *Ping* enviado pela aplicação quando é necessário um *heartbeat* ao qual este espera um *Pong* de resposta pelo cliente. Caso este não o envie, a conexão é considerada perdida e a aplicação termina a conexão e a sessão associada. Já o tipo de evento *Close* é enviado somente pelo cliente para notificar o servidor que vai terminar a conexão por vontade própria, este tipo de evento permite que o servidor não tente guardar qualquer tipo de informação sobre a sessão.

Para tipos de eventos relacionados com a gestão da sessão temos *SessionInfo, SessionRestore* e *SessionRestored*. O *SessionInfo* é sempre a primeira mensagem enviada pela aplicação, e contém o contexto da sessão, sendo as suas propriedades:

- *HubID* - Identificador o *hub* a que a sessão pertence;
- *SessionID* - Identificador gerado para a sessão em questão;
- *HubAllowAnonymous* - Se o *hub* permite conexões anônimas, serve para informar o cliente que tem um intervalo de tempo configurável na aplicação, em que a sessão pode-se autenticar, caso não o faça esta é terminada pela aplicação;
- *HubAllowUserChannel* - Se o *hub* permite o *channel* de utilizador;
- *DefaultPublic* - Se os *channels* no *hub* são públicos por defeito;
- *Authenticated* - Se esta sessão já está autenticada, visto ser possível autenticar através de parâmetros enviados enquanto a conexão está a ser estabelecida.

Os outros dois tipos de eventos são utilizados para a recuperação de sessão. A recuperação de sessão é uma funcionalidade que permite que uma sessão já autenticada seja armazenada de forma temporária para que possa ser recuperada em casos de reconexões rápidas, algo que acontece regularmente com dispositivos móveis. A sessão armazenada ou *Session State*, guarda a autenticação, permissões, identificadores e *channels* subscritos. Quando o cliente tenta reconectar, este envia uma chave gerada que identifica a sessão a ser restaurada, caso esta seja encontrada, é restabelecido todo o estado anterior e tentando restabelecer a subscrições aos *channels*, visto que configurações deste possam ter mudado de forma a tirar o acesso à sessão. Sempre que uma chave é utilizada, a informação armazenada é apagada para evitar reutilização. Para o funcionamento desta funcionalidade, é enviado o tipo de evento *SessionRestore* após a sessão ser autenticada, este contém somente a propriedade *RestoreKey* que é a chave gerada pela aplicação para restaurar a sessão. Por fim, quando o utilizador se reconecta e a sessão é restaurada, é enviada a mensagem do tipo *SessionRestored* com as seguintes propriedades:

- *UserID* - O identificador de utilizador da sessão;
- *Authorizations* - Lista de autorizações dos *channels*;
- *Extra* - Metadata definida na autenticação;
- *SubscribedChannels* - Lista de channels em que a subscrição foi recuperada;
- *RPCs* - Lista de autorizações de *RPCs*.

4.7.8.3 Autenticação e autorização

De forma a autenticar a sessão, existe somente um tipo de mensagem: *Auth*. Este tipo tem significados diferentes dependendo de quem a envia, do cliente para aplicação é interpretado como um pedido de autenticação, de forma inversa, é interpretado como uma resposta ao pedido de autenticação. As informações a serem utilizadas para autenticar o utilizador são indiferentes para aplicação sendo a responsabilidade do destinatário do *Auth Provider* de interpretar e validar o conteúdo, dessa forma, a mensagem de autenticação do cliente consiste no envelope com tipo de evento *Auth* e conteúdo é um conjunto de *bytes*. De forma a informar a sessão do resultado, o mesmo tipo de evento é enviado, mas tendo como conteúdo:

- *Success* - Se a autenticação teve sucesso;
- *UserID* - Identificar de utilizador;
- *Token* - Um *token* gerado pela aplicação de forma a utilizar outras *APIs* da aplicação;
- *ChannelAuthorizations* - Lista de autorizações para *channels*;
- *RPCAuthorizations* - Lista de autorizações para *RPCs*;
- *Extra* - Um dicionário de valores adicionais.

As propriedades *ChannelAuthorizations* e *RPCAuthorizations*, assim como os seus nomes indicam, são o meio disponível para fornecer permissões às sessões, além das configurações como *Public* e *Allow Anonymous*. Cada *ChannelAuthorization*, consiste em três simples propriedades, sendo o *channel* a ser permitido, se tem permissão de escrita (*Publish*) e se tem permissão de leitura (*Subscribe*). Para a propriedade *channel*, é possível utilizar *wild-cards* com os seguintes caracteres:

- * - Este permite que qualquer valor antes ou depois deste, por exemplo, com “product:*”, qualquer *channel* que comece com “product:” é atingido pela permissão.

- # - Permite um valor arbitrário até ao próximo separador com ":", por exemplo, "product:#:stock" que atinge qualquer *channel* que tenha outro valor entre o "product:" e ":stock" desde que não contenha um separador.

Para o *RPCAuthorization*, este tem somente a propriedade com o nome do método, e utiliza os mesmos *wild-cards* que o *ChannelAuthorization*.

4.7.8.4 Session State

O *Session State* é um *snapshot* do último estado de uma sessão, este contém o identificador de sessão e *hub*, *channels* subscritos e informações recebidas após autenticação. Sempre que uma alteração ocorre na sessão, como por exemplo, uma nova subscrição, o estado atual da sessão é copiado e armazenado no *Redis*, posteriormente, utilizando a chave de restauração (*RestoreKey*) é realizada uma tentativa de restaurar todas subscrições ao channels anteriores (*SessionRestore*), e toda a informação de autenticação é restaurada assim como o mesmo identificador de sessão.

4.7.8.5 RPCs

Uma funcionalidade adicional, são os *RPCs*, quando um utilizador é autenticado, este recebe uma lista de permissões para *RPCs*. Estes são representados por um valor de texto, e o seu conteúdo é indiferente para a aplicação, sendo somente exigido o seu envio como um conjunto de *bytes*. Cada *RPC* realizado é redirecionado para o *NATS* que gere o envio para o destinatário e a sua resposta.

4.7.8.6 Streams

A funcionalidade de *streams* está de momento incompleta e irá se manter desativada inicialmente, até os casos da sua utilização serem bem estabelecidos. O objetivo inicial desta funcionalidade, passa por permitir que clientes submetam eventos para uma *stream* na aplicação *NATS* ou outra como *AWS Kinesis*. Todos os eventos, são complementados com informações da sessão, como identificador de utilizador e sessão. Esta funcionalidade será utilizada futuramente para coleta de eventos para análise, assim como forma de os clientes emitirem ações que tenham realizado nas aplicações móveis e receber eventos que tenham sido acionados pelas suas ações.

4.7.8.7 Document

Para o funcionamento de um *channel* do tipo *Document*, existem 4 tipos de mensagens:

- *DocumentGet* - Pedir cópia do conteúdo atual documento se for enviado pelo cliente e resposta de for pelo lado da aplicação;
- *DocumentChange* - Realizar alterações ao documento se for enviado pelo cliente e resposta às alterações se for enviado pela aplicação;
- *DocumentUpdated* - Enviado somente pela aplicação sempre que uma alteração ao documento é realizada, contendo nova versão e alterações aplicadas;
- *DocumentInfo* - Enviado somente pela aplicação com informação atual sobre o documento como versão documento. Este é sempre enviado quando um cliente subscreve ao *channel*.

4.7.8.8 Default Channel

Num *Default* e *Document channel*, onde o *PubSub* é permitido existem somente dois tipos de mensagens: *Publish* e *Ack*. Como os nomes indicam, o primeiro representa um pedido de publicação realizado pelo cliente ou quando enviado pela aplicação, representa uma publicação que tenha ocorrido. Por fim, o *Ack* é enviado como confirmação se o *Publish* ocorreu com sucesso, e é somente enviado caso solicitado.

4.7.8.9 Notification Channel

Por fim, os Notification *channels* utilizam os tipos de mensagens:

- *NotificationNew* - Sempre que uma nova notificação é criada;
- *NotificationRead* - Recebido pelo cliente quando uma notificação é marcada como lida;
- *NotificationInfo* - Informação recebida pelo cliente com o número de notificações não lidas;
- *MarkNotificationAsRead* - Enviada pelo cliente, quando pretende marcar um conjunto de notificações como lidas.

4.8 Métricas

De forma a compreender como a aplicação está a funcionar, estão a ser coletadas algumas métricas a nível global e individualmente por cada *tenant*, estas métricas começaram por ser somente número de mensagens e *bytes* enviados e recebidos que são coletadas em forma de contadores totais e os seus valores em intervalos de 5 em 5 minutos, sendo um valor

configurável. De forma a coletar erros e analisar a origem destes, é utilizado o serviço *AWS Cloudwatch* (apêndice C) e *AWS X-Ray* (apêndice D), este também é utilizado para acompanhar informações como número de instâncias, utilização de *CPU* e *RAM*. Adicionalmente, existem mais métricas a serem coletadas a nível global utilizando *Prometheus*, além das primeiras mencionadas, temos número de subscrições, sessões ativas, tempo médio de processamento de uma publicação num *channel* e tempo médio de processamento de alterações a um documento. Estas métricas são fundamentais para avaliar a eficiência da aplicação, identificar possíveis problemas e promover melhorias na aplicação. No entanto, eventualmente surgiu o interesse de analisar mais duas métricas adicionais, que resultou na coleta de cada sessão iniciada e quando esta terminava, desta forma, pode ser observado quantas sessões foram iniciadas num período de tempo e a duração de cada, estas métricas são coletadas por cada *tenant.*

4.8.1 Dashboard

Ao longo do desenvolvimento da aplicação, foi também desenvolvido um cliente em *typescript* e um *dashboard* para testar o funcionamento da aplicação e ser capaz de analisar o seu funcionamento. Dessa forma, o *dashboard* foi desenvolvido em *typescript* (apêndice E) com a biblioteca de interface visual *React*. Esta ferramenta consiste em somente 4 páginas, e sendo uma ferramenta interna esta contém mais uma página para a autenticação do seu utilizador.

4.8.1.1 Contadores

A primeira parte consiste numa simples visualização dos contadores armazenados globalmente, assim como a possibilidade de ver as de um *hub*. Adicionalmente, mostra algumas informações atuais sobre o *cluster* como número de membros, *hubs*, sessões e *channels* ativos no momento, assim como pode ver visto no apêndice F, figura 41.

4.8.1.2 Topografia de *cluster*

A segunda parte, permite inspecionar a topografia do *cluster*. Isto é realizado pedido a um membro do *cluster* que pede a cada membro do *cluster* que realize uma introspeção a todos os *hubs* e seus respetivos *channels* e sessões, esta informação é depois enviada para o *dashboard* onde é transformado numa topografia. A visualização da topografia é dividida em três gráficos, no primeiro, representado no apêndice F, figura 42 e figura 43, é demonstrado todos os membros do *cluster* e o seu nome neste, adicionalmente, quando existe uma linha entre eles significa que existe uma conexão *gRPC* com *streaming* ativa entre estes. A segunda

visualização da topografia é representada no apêndice F, figura 43 e figura 44, onde é demonstrado a partir de uma origem como centro do gráfico todos os *hubs* atualmente ativos e ligado a estes os seus *channels* pertencentes. Por fim, a última visualização como representado no apêndice F, figura 45 e figura 46, é demonstrado a hierarquia de membros para *hubs*, para *channels* e sessões. Estas visualizações da topografia do *cluster* foram e são fundamentais para perceber como o *cluster* distribui a carga recebida e como se comporta na presença de falhas e modificações ao número de membros no *cluster*.

4.8.1.3 Visualização de métricas

A terceira parte consiste na visualização das primeiras métricas previamente definidas em todos *cluster* ou por *tenant*, em simples gráficos de linhas. Portanto no apêndice F, figura 47, figura 48 e figura 49 temos a evolução dos *channels*/sessões/*hubs* ativos ao longo do tempo, e nas figura 50 e figura 51 temos o número de mensagens ou *bytes* enviados em contraste com os recebidos. Por fim temos 2 gráficos que foram posteriormente adicionados permitindo saber quantas sessões foram iniciadas por dia, e qual a sua duração por dia como pode ser visto no apêndice F figura 52 e figura 53.

4.8.1.4 Ferramenta para inspecionamento

A última parte consiste numa ferramenta que permite inspecionar o funcionamento da aplicação. Esta utiliza o cliente desenvolvido, e fornece as funcionalidades deste numa interface visual, como pode ser visto no apêndice F, figura 54. Esta parte, pode ser divida em mais 3 partes. A primeira, consiste nas informações de sessão, como estado de conexão, identificador de utilizador, informações sobre *hub*, informações extra de autenticação e permissões como demonstrado no apêndice F, figura 55. A segunda parte consiste em todo o histórico da conexão, assim como visto no apêndice F, figura 56, esta regista quando a conexão inicia e todos eventos que recebe, aqui eventos como *PING/PONG* (mecanismo equivalente ao *heartbeat*) são ignorados pois são muito frequentes e pouco úteis. Em cada registo na tabela é possível inspecionar o conteúdo caso este esteja em formato de texto, caso contrário um conjunto de *bytes* é apresentado. Por fim no apêndice F, figura 57, temos todos os *channels* a que esta sessão está subscrita. Em cada aba, é demonstrado as funcionalidades de presença e *occupancy*, em *channels* com *PubSub* o histórico de publicações é demonstrado, nos de notificações de forma similar é demonstrado a lista de notificações e por fim como representado na mesma figura, temos a representação de um documento que se mantém sempre atualizado.

4.8.2 Testes

Os testes são uma parte fundamental do processo de desenvolvimento de uma aplicação. Estes ajudam a garantir que uma aplicação está a funcionar conforme o esperado, identificando problemas de desempenho, segurança e usabilidade e permitindo reduzir o tempo e o custo do processo de desenvolvimento. Neste projeto existem maioritariamente testes unitários e teste de integração.

4.8.2.1 Testes unitários

Os testes unitários são uma técnica essencial de programação que consiste em testar partes específicas e isoladas do código fonte de um programa, nomeadas de *unit test*. Grande parte do código foi desenvolvido de forma modular com o objetivo de facilitar o desenvolvimento de *unit tests*, e muito do código foi desenvolvido ao mesmo tempo que os *unit tests* para estes de forma similar a *test-driven development*. Embora não exista uma cobertura de testes a 100% do código, todo o seu código principal e lógica estão cobertos num total de 60%.

4.8.2.2 Testes de integração

Os testes de integração são utilizados para garantir que diferentes partes da aplicação funcionem corretamente juntas. Estes testes ajudam a identificar erros e falhas que podem ocorrer quando diferentes componentes são combinados, enquanto testes unitários focam-se em pequenas partes de funcionamento.

4.8.2.3 Experimentação com clientes

De forma a experimentar a aplicação com tráfego real, foi pensada uma forma gradual para expor a aplicação a utilizadores finais sem impactar caso ocorra uma falha com este, ao mesmo tempo, vão sendo coletadas métricas de forma a procurar problemas que possam estar a ocorrer. Sendo que o objetivo é testar a capacidade do sistema, foram utilizados servidores com baixas especificações, com somente 0.25 *vCPU* e 0.5 *GB* de *RAM*.

Portanto, a exposição está a ocorrer em 4 simples fases, na primeira consiste em fazer com que as aplicações se conectem ao sistema, sem utilizar nenhuma funcionalidade deste, somente manter uma sessão autenticada ativa de forma a estimar quantas conexões podemos esperar por agora, nesta primeira fase somente 2 clientes foram escolhidos como alvos.

Para a segunda fase, todos os clientes irão manter uma conexão ativa com o mesmo objetivo da anterior, e manter um *channel* ativo por sessão.

Na terceira fase, será utilizada de forma a experimentar uma possível primeira funcionalidade, em que cada sessão irá se subscrever ao item que estejam a ver utilizando a capacidade de *occupancy*. Por fim, na última fase será aumentada a quantidade de dados enviados pelas sessões, essa informação poderá ser de novas funcionalidades ou somente dados para teste.

4.8.2.4 Resultados

Antes de expor o sistema às aplicações móveis, foi feito um teste de quantas conexões um *cluster* com 3 membros seria capaz de suportar, utilizando as especificações previamente mencionadas, e o comportamento do *cluster* ao adicionar ou remover membros. Neste teste, cada membro do *cluster* foi capaz de suportar aproximadamente 15 mil conexões num total de aproximadamente 45 mil conexões, não sendo capaz de ter mais conexões devido ao limite de *RAM* nos servidores. Utilizando servidores com maior capacidade o *cluster* é capaz de aumentar a quantidade de conexões com o mesmo número de membros, no entanto, o objetivo é ver a capacidade de distribuição e tolerância a falhas. Estes valores são muito superiores aos que eram esperados no sistema anterior, além de ser capaz de suportar falha dos membros sem indisponibilizar o serviço.

O ajuste do *cluster* ao adicionar ou remover membros era relativamente rápido, demorando aproximadamente 300ms, visto que este notifica os outros membros de que está ativo ou que vai deixar de estar. Nos casos de falha, por impossibilidade de um membro notificar os outros de que vai deixar de estar ativo, o *cluster* demorou aproximadamente 700ms. O motivo desta duração deve-se principalmente à deteção do estado de atividade do membro. Naturalmente, em *clusters* com maior número de membros estes ajustes vão demorar mais, mas é um comportamento já documentado e não é problemático. É possível fazer alguns ajustes nas configurações da implementação de *gossip*, no entanto, não há necessidade para alterar as atuais.

Quanto às fases de teste com tráfego real, não foi possível obter todos os resultados antes da produção deste documento. Visto que o projeto ainda se encontra na segunda fase de testes, as análises de resultados são bastante limitadas.

Nas primeiras duas fases de testes com tráfego real, foi observado que o número de sessões em simultâneo era consideravelmente inferior ao esperado, no entanto, é possível verificar que o número de sessões é distribuído ao longo do dia, com picos relativamente pequenos.

Portanto, de forma a melhor avaliar as sessões ao longo do tempo, foi implementado a coleta de sessões e suas durações. Com estas métricas adicionais, assim como previsto, podemos concluir que grande parte das sessões são de curta duração, sendo a média de aproximadamente de 2 minutos. No apêndice F, figura 29, podemos observar a distribuição de sessões ao longo do dia, esta visualização acumula as sessões dos *tenants* em fase de teste. Assim como pode ser observado, a única altura do dia onde o número de sessões é mais elevado é entre as 21 e 22 e 9 e 11 horas, nas outras alturas do dia o número de sessões é relativamente baixo. Ao mesmo tempo, a duração de sessões é maior entre as 5 e 6 horas, assim como pode ser observado no apêndice F, figura 30

Tirar conclusões desta informação é somente relevante quando é analisado tendo em conta o *tenant*, os seus produtos e campanhas, no entanto, com esta informação podemos analisar a carga de pedidos na aplicação, quando esta é mais esperada e utilizar esta informação de forma a preparar o *cluster* para a carga esperada, embora este processo não será tido em conta por agora.

4.8.3 Clientes

De forma a integrar a aplicação com dispositivos móveis e aplicações *Web*, a implementação de clientes teve de ser criada. Por agora, as únicas plataformas alvo são *Android*, *IOS* e *Web*, sendo as linguagens respetivas para cada plataforma, *Kotlin*, *Swift* e *javascript*. Visto protocolo utilizar *protocol buffers*, utilizando as suas ferramentas foi possível gerar todo as mensagens assim como a sua forma de serialização para cada plataforma, sendo somente necessário implementar a gestão da conexão e sessão e a informação recebida para cada tipo de *channel* sendo somente o *channel* do tipo *document* o mais complexo. Para as plataformas móveis foi utilizado o *Kotlin Multiplatform Mobile* (*KMM*) que permite criar somente uma única implementação para as plataformas *Android* e *IOS*. Por fim, o cliente para *Web* em *javascript* foi desenvolvido em conjunto com o *dashboard*, e é utilizado na ferramenta de inspecionamento. Existem alguns pontos importantes a serem mencionados no lado das implementações dos clientes, sendo o protocolo e gestão de sessão.

4.8.4 Protocolo

Assim como previamente mencionado, as mensagens foram definidas com *protocol buffers*, além de permitir gerar código para várias linguagens de programação, este permite também implementar novas funcionalidades no protocolo mantendo compatibilidade com versões anteriores, sendo necessário que o cliente saiba que versão do protocolo este utiliza. Em geral

novas versões somente irão adicionar novas propriedades ou tipos de mensagens de forma a tentar manter o máximo de compatibilidade entre protocolos.

4.8.5 Gestão de Sessão

Sendo alguns dos alvos as plataformas móveis, é necessário que a gestão de conexão seja capaz de lidar com desconexões abruptas e ser capaz de restabelecer a conexão de forma a manter a sessão. Portanto, de forma a restabelecer a sessão anterior existem os mecanismos de *SessionRestore* e *SessionRestored* com este exato objetivo. No entanto, mensagens que tenham sido enviadas durante o período de desconexão são perdidas pelo cliente. De forma a tentar recuperar essas mensagens, existe a funcionalidades previamente mencionada de *live history*, onde o cliente pode requisitar mensagens enviadas após uma *timestamp*. De lembrar, que embora estas funcionalidades existam, estas funcionam na forma de *best effort* não dando fortes garantias, portanto, nestes casos o cliente deve ser capaz de ser o próprio a autenticar a sessão de subscrever de volta aos *channels* e enviar a última *timestamp* recebida de cada, para que estes tentem enviar as mensagens perdidas. A implementação também gere mensagens a serem enviadas, pondo-as numa fila de envio, de forma que mensagens a serem enviadas pelo cliente não sejam perdidas. Por fim, caso um servidor falhe, todas conexões a este têm que ser restabelecidas, caso todas sejam feitas relativamente ao mesmo tempo, poderá ocorrer um *thundering herd*, ou seja, um aumento repentino de novas conexões que podem causar que outros servidores falharem repetindo o mesmo problema e com mais conexões. De forma evitar um *thundering herd* um *backoff* exponencial é implementado para realizar a reconexão, começando com uma tentativa instantânea seguida de um intervalo de tempo a multiplicar pelo número de tentativas.

4.8.6 Escalabilidade futura

Sendo um dos pontos fundamentais deste projeto, a sua capacidade de escalar horizontalmente, foram pré-definidas formas de aumentar a escalabilidade do sistema, caso eventualmente seja necessário. Em primeiro lugar temos de identificar os pontos em que o desempenho do sistema em geral pode ser impactado.

4.8.6.1 Problemas

Como primeiro ponto identificado, temos o número de conexões de clientes a uma aplicação. Cada conexão exige a gestão de *PING/PONG*, o envio de mensagens e o processamento de mensagens recebidas pelo cliente. Esta gestão pelo que foi observado

localmente, é a primeira parte a afetar o desempenho. Por exemplo, uma mensagem de *channel* a ser enviada para 50 sessões exige que esta seja copiada 50 vezes para *sockets* no sistema operativo, que apresenta um custo a nível de processamento e de memória, incluindo que cada servidor tem um máximo de conexões que pode manter abertas ao mesmo tempo.

Como segundo ponto, temos o número de conexões ativas entre vários membros do *cluster*, imaginando um *cluster* de 20 membros, é possível que um membro tenha de manter uma conexão ativa a vários outros devido à distribuição dos *channels* entre todo o *cluster*.

Por fim, devido ao processamento de mensagens em todos os tipos de *channels* atuais ser realizado serialmente, um *channel* com muito tráfego poderá não conseguir acompanhar a quantidade de mensagens. Neste último ponto, pelo menos por agora, não existe uma solução definida embora seja possível simplesmente retirar o processamento serial de cada mensagem e distribuir entre vários *threads*, o que exige que a ordem das mensagens não seja mantida.

4.8.6.2 Possíveis soluções

A solução já pré-definida consiste em atribuir diferentes cargos aos membros do *cluster*, idealmente de forma dinâmica. Estes cargos seriam *Edge e Core*, neste momento estes cargos já estão definidos, no entanto, de forma estática nas configurações iniciais da aplicação. Um membro com o cargo de *Core* irá funcionar normalmente como se não houvesse cargos, enquanto um membro *Edge*, não será contabilizado como um membro de distribuição de *channels*, ou seja, vai somente receber conexões de clientes e se conectar aos membros *Core* quando precisa de se subscrever.

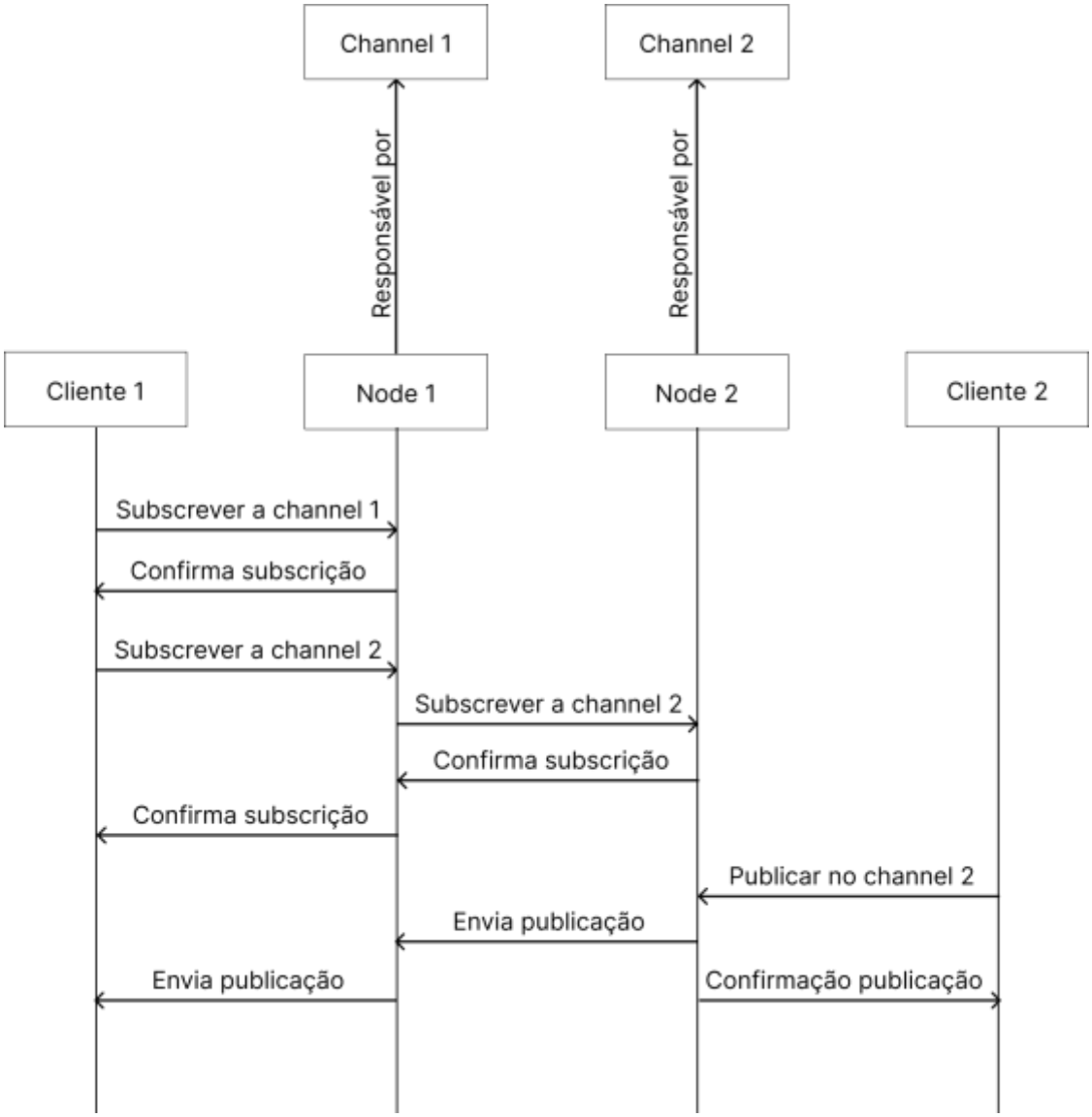
Desta forma, os membros com o cargo *Core* irão se focar principalmente com o processamento de mensagens, enquanto os outros gerem somente conexões, esta solução permite resolver tanto o primeiro como o segundo ponto. Para o primeiro ponto, podemos ter mais membros com cargo *Edge* sendo que não afetam a redistribuição do *cluster*, e para o segundo ponto, podendo os membros com cargo *Core* que serão menos, o número de conexões entre os membros do *cluster* pode ser reduzido significativamente.

4.9 Diagramas

De forma a demonstrar um segmento da aplicação, temos a figura 16, nesta temos a exemplificação dos passos ocorridos quando um cliente se subscreve a um *channel* que pertence ao *Node* em que este está conectado e quando o mesmo cliente se subscreve a um *channel* a que pertence a outro *Node*. Para o processo inverso de remover subscrição a processo é o mesmo, sendo a única diferença o conteúdo enviado nas mensagens, e a publicação feita

pelo *Cliente 2* não seria enviada do *Node 2* para o *Node 1*. As mensagens enviadas entres os clientes e *Nodes* podem ser observadas no apêndice G onde estão as mensagens do protocolo relevantes para este exemplo.

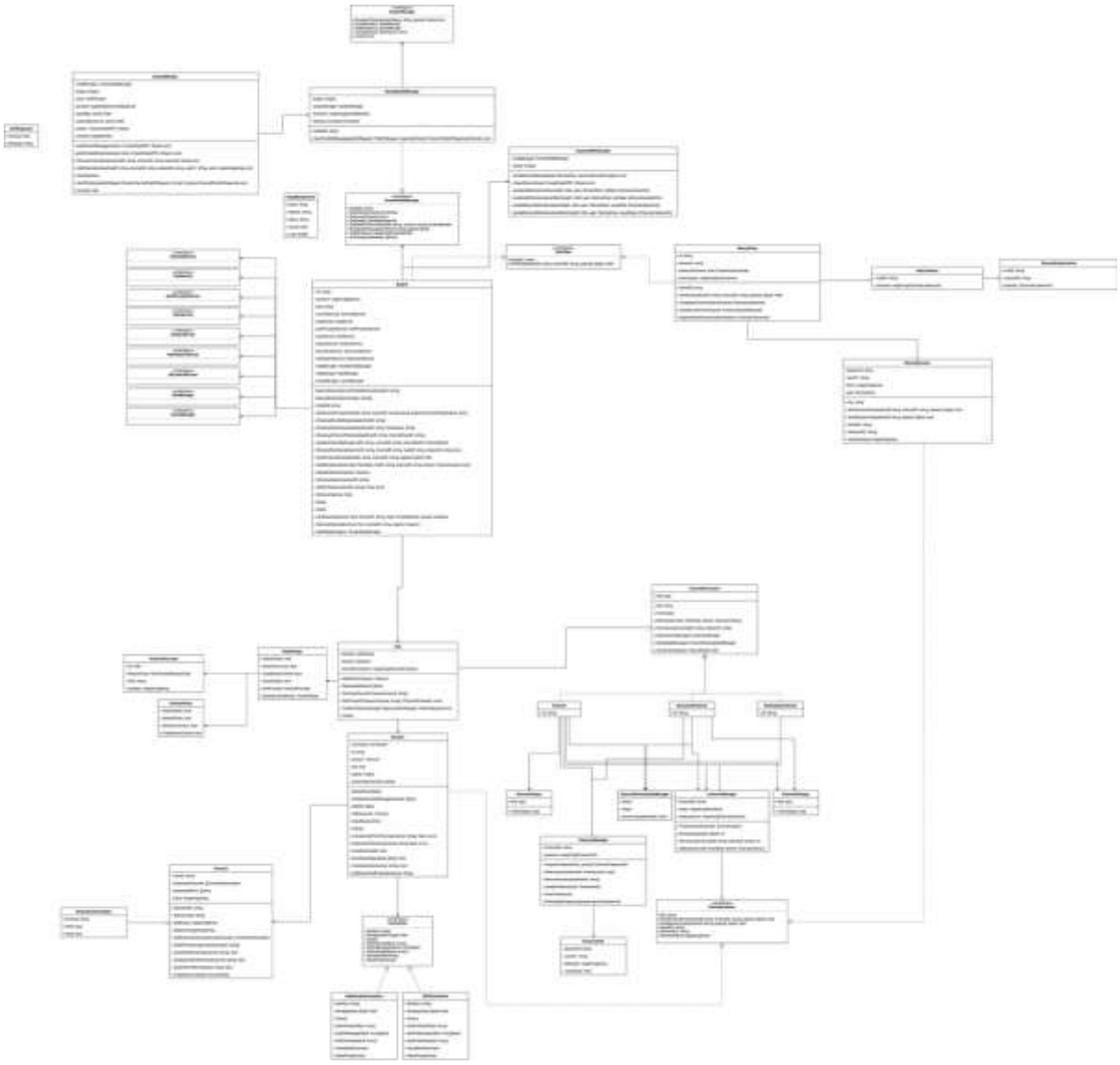
Figura 16 - Flow de subscrição do cluster



Fonte: Própria

De a ter uma visualização do sistema temos o seguinte diagrama de classes na figura 17 que se encontra incompleta e ilegível por motivos de privacidade empresarial. Este diagrama mostra apenas as classes que representam o funcionamento principal do sistema, as configurações da aplicação, armazenamento e definição de *APIs* ou não estão representadas apresentam uma classe vazia.

Figura 17 - Diagrama de classes



Fonte: Própria

De forma a tornar este diagrama legível, iremos focar nas classes e interfaces principais, o *Engine*, o *ClusterNodeManager , o ChannelProcessor*, o *Hub*, a *Session* e o *ChannelListener*.

4.9.1 Engine

O *Engine* é o elemento que agrega todos os componentes do sistema, este apresenta dois métodos *Start* e *Stop* que iniciam e terminam a aplicação. Esta classe é a que recebe transforma pedidos recebidos por outros *Nodes* em ações, assim como é a classe utilizada quando pedidos na *API* de administração são realizados. Portanto, esta classe faz a agregação de todos os *hubs* e sessões, assim como é o elemento que serve como ponte entre o *ClusterNodeManager* e o *hub* local. Sempre que alterações são realizadas no cluster, esta classe é notificada e é

responsável por realizar o seu processo de redistribuição dos *channels*. A representação desta classe pode ser observada na figura 18.

Figura 18 - Diagrama do Engine

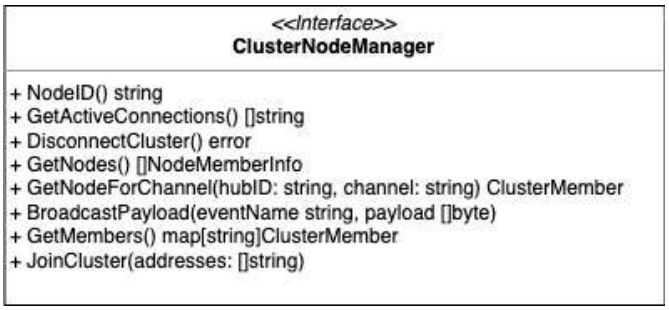
Fonte: Própria

4.9.2 ClusterNodeManager

O *ClusterNodeManager* é uma interface que representa as funcionalidades que são necessárias que os componentes que gerem o cluster sejam capazes de cumprir. Está definido como uma interface de forma permitir que sejam criadas implementações diferentes utilizados variações diferentes do protocolo *gossip*.

Para entender o funcionamento a nível interno temos a figura 24 e figura 25, ambos fazem parte do mesmo diagrama, mas estão separadas de forma a facilitar a leitura. Na primeira figura, pode ser observado o que ocorre quando uma sessão se subscreve a um *channel* que pertence ao *Node* em que esta se encontrada conectada, neste caso, o *hub* e *channel* são criados e inicializados de forma dinâmica e a sessão é adicionada como um subscritor ao *channel*. Na segunda figura, pode ser observado o processo para a subscrição a um *channel* a que o *Node* responsável por este não o mesmo que a que sessão se encontra conectada, neste caso, uma conexão e uma *stream gRPC* são criadas caso não existam, assim que o *Node* responsável receba a mensagem, este vai notificar o seu *Engine* do pedido de subscrição recebida. Este por sua vez irá criar e inicializar o *hub* e *channel* caso já não estejam e irá adicionar a sessão como um *ChannelListener*. A representação da interface pode ser observada na figura 19.

Figura 19 - Interface ClusterNodeManager

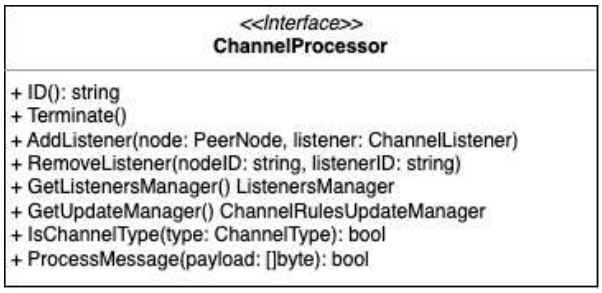


Fonte: Própria

4.9.3 ChannelProcessor

O *ChannelProcessor* é a interface que define os comportamentos do *channel*, todos os tipos de *channels* como *document*, *notification* e *default* implementam esta interface, esta recebe os pedidos de publicação de eventos e é responsável pelo envio por para as sessões subscritas, que são representadas pela classe *ChannelListener*. A representação da interface *ChannelProcessor* pode ser observada na figura 20.

Figura 20 - Interface do ChannelProcessor

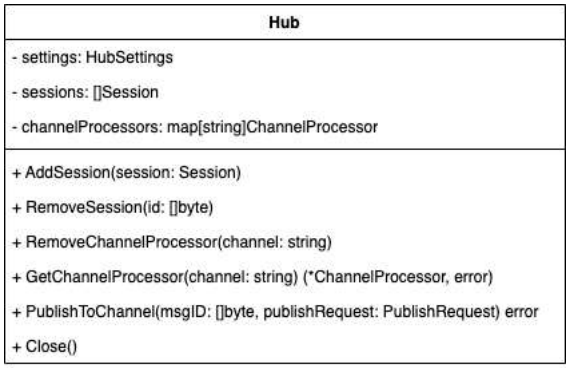


Fonte: Própria

4.9.4 Hub

O *Hub* é a classe que representa um *tenant* no sistema, por esse motivo, este agrega todos os *channels* e sessões pertencentes ao *tenant*. Sempre que é necessário inicializar ou terminar um *channel* essa operação é realizada pelo *hub*, ou seja, o *hub* é a classe que permite realizar operações a nível de um *tenant*. A representação da classe pode ser observada na figura 21.

Figura 21 - Diagrama do Hub

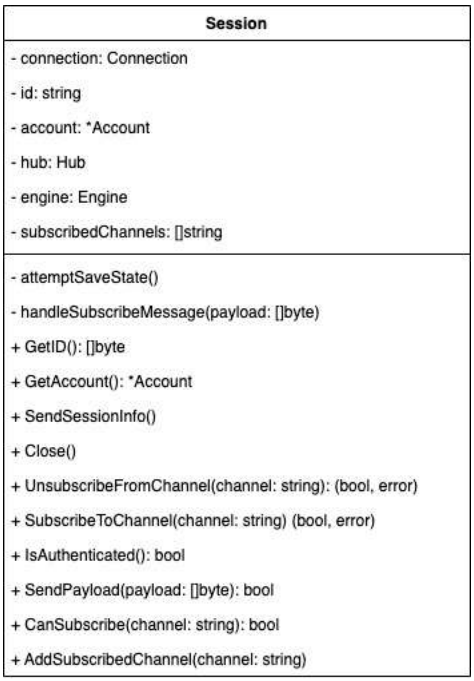


Fonte: Própria

4.9.5 Session

A *Session* ou sessão, é a classe que representa um cliente conectado. Atravéz da sessão os utilizadores podem publicar eventos e subscrever a *channels*. Assim como previamente mencionado, a sessão pertence sempre a um *hub* e pode subscrever a um *channel* com a sua representação de *ChannelListener*. A representação da classe da sessão pode ser observada na figura 22.

Figura 22 - Diagrama da Session

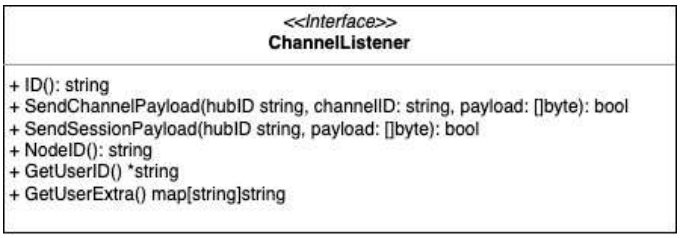


Fonte: Própria

4.9.6 ChannelListener

O *ChannelListener* é a interface que representa uma subscrição num *channel*. Sempre que uma sessão é adicionada como subscritor a um *channel* esta é adicionada uma classe que implementa esta interface. Adicionalmente, sessões remotas também são representadas por esta interface. O diagrama da interface pode ser observado na figura 23.

Figura 23 - Interface do ChannelListener



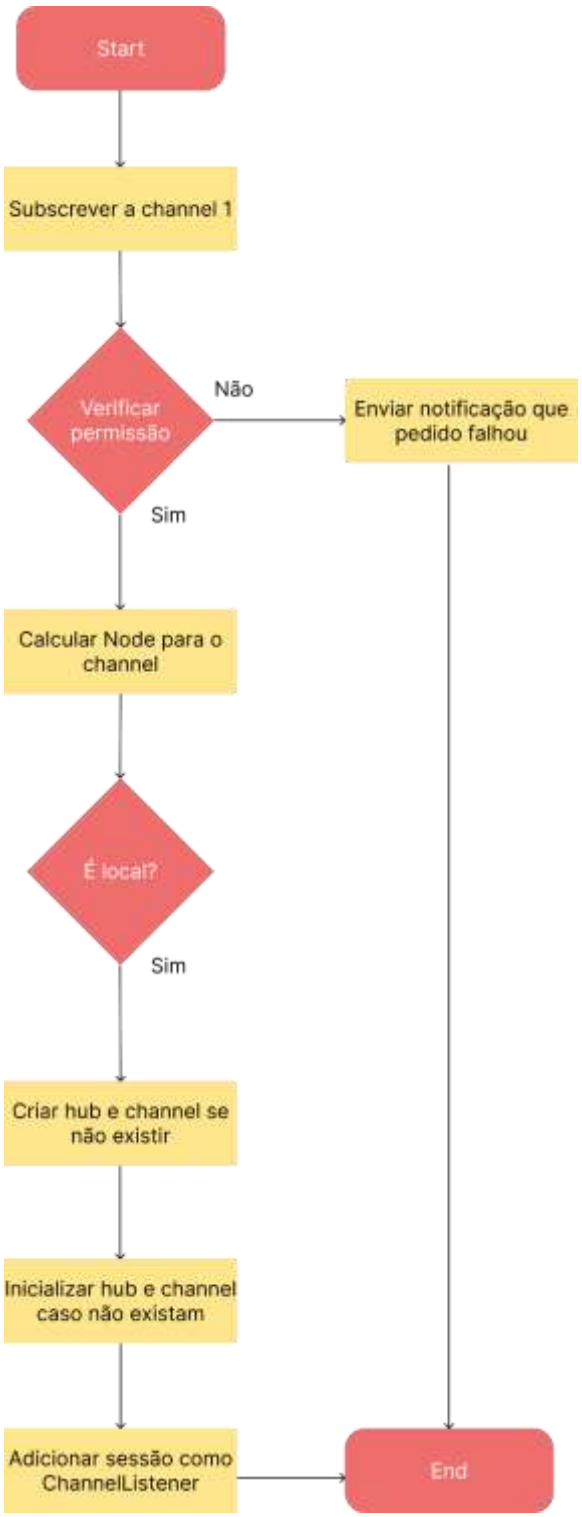
Fonte: Própria

4.9.7 Fluxograma de subscrição

Agora que foram revistas as classes e interfaces principais, podemos ver um fluxograma dos passos para a subscrição de um *channel* assim como foi representado previamente de forma mais simples. Na figura 24 e figura 25 temos um fluxograma que foi divido em duas partes, a

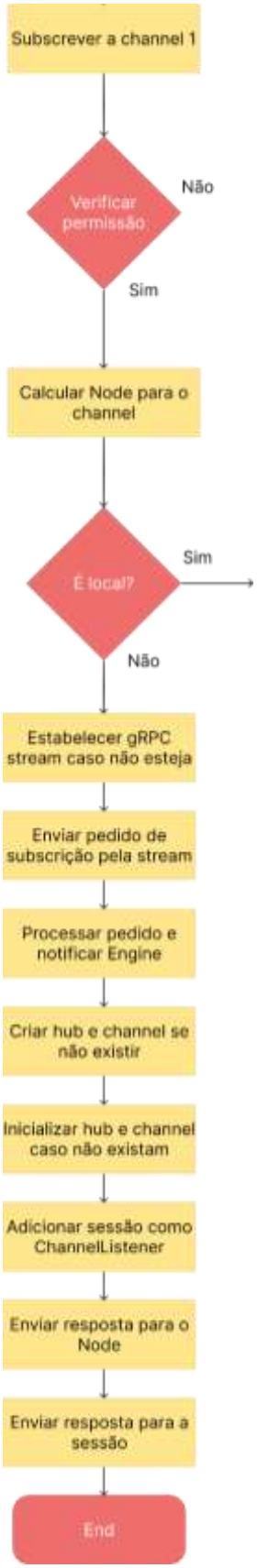
primeira representa o processo de subscrição a um *channel* quando o *Node* é responsável, ou subscrição a um *channel* local, na segunda parte é representado o processo quando o *channel* pertence a outro membro do *cluster*.

Figura 24 - Subscrição a um channel local



Fonte: Própria

Figura 25 - Subscrição a um channel remoto



Fonte: Própria

Este processo de subscrição é representado pelo segmento código presente na aplicação que está representado na figura 26. Neste segmento de código já é assumido que a sessão tem permissão para subscrever ao *channel*. Assim que esta função é invocada, esta calcula o *Node* responsável pelo *channel*, caso seja local este é criado caso não esteja e a sessão é adicionada como *ChannelListener*. Se o channel pertencer a um *Node* remoto, este faz um pedido de subscrição ao *Node*.

Figura 26 - Função de subscrever a um channel

```
func (session *Session) SubscribeToChannel(channel string) (bool, error) {

    // Calcula o Node para o channel
    node := session.engine.nodeManager.GetNodeForChannel(session.Hub.ID, channel)

    // Se o Node for local
    if !node.IsLocal() {
        // Faz o pedido ao Node para adicionar a sessão como ChannelListener
        if err := node.AddChannelListener(session.Hub.ID, channel, session.id.String(), session.GetUserID(),
session.GetUserExtra()); err != nil {
            ...
        }
    } else {
        // Adiciona a Session como channel listener
        cp, err := session.Hub.GetChannelProcessor(channel)
        ...

        cp.AddListener(session.engine, session)
    }

    // Adicionar channel á lista de channels subscritos
    session.AddSubscribedChannel(channel)
    // Guardar registo de subscrição
    session.engine.AddSubscription(session.Hub, channel, node, session)

    return true, nil
}
```

Fonte: Própria

5 DISCUSSÃO DE RESULTADOS

Assim como previamente mencionado, antes de expor o novo sistema às aplicações móveis, foi feito um teste de quantas conexões um *cluster* com 3 membros seria capaz de suportar, em servidores com somente 0.25 *vCPU* e 0.5 *GB* de *RAM*. Neste teste, cada membro do *cluster* foi capaz de suportar aproximadamente 15 mil conexões num total de aproximadamente 45 mil conexões, não sendo capaz de ter mais conexões devido ao limite de *RAM* nos servidores. Utilizando servidores com maior capacidade o *cluster* é capaz de aumentar a quantidade de conexões com o mesmo número de membros, no entanto, o objetivo era ver a capacidade de distribuição e tolerância a falhas. Estes valores obtidos são muito superiores aos 9000 atualmente esperados, além de ser capaz de suportar falha dos membros sem indisponibilizar o serviço.

Após o teste de conexões foi realizado outro teste com o objetivo de avaliar o tempo que um *cluster* demora a ajustar os channels pelos membros. Neste teste existem 3 membros no *cluster* e um quarto é adicionado normalmente e removido repentinamente de forma a simular uma falha. Após cada ajuste é coletado a partir dos *logs* de cada membro o tempo que o ajuste demorou em milissegundos, para clarificar, só é apontado a duração do ajuste e não de deteção que o membro foi adicionado ou removido.

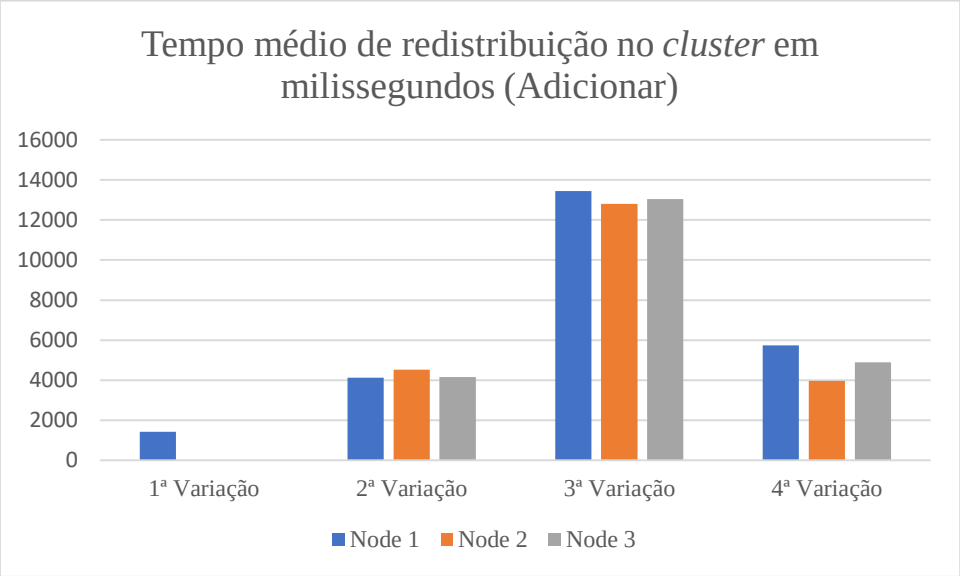
Para este teste, foram escolhidas 4 variações com 5 rondas cada, nestas variam o número de *channels* e o número de clientes, no entanto estes não ultrapassam dos 3. As 4 variações são as seguintes:

- 1ª Variação:
  - 1 cliente conectado ao *Node 1*;
  - 500 *channels*;
  - Identificador de *channels* em *UUIDs* (*Universal Unique IDentifiers*).
- 2ª Variação:
  - 3 clientes, um conectado a cada *Node*.
  - 1500 *channels*;
  - Identificador de channels em conjunto de 20 caracteres aleatórios.
- 3ª Variação:
  - 3 clientes, um conectado a cada *Node*.
  - 4500 *channels*;
  - Identificador de *channels* em conjunto de 40 caracteres aleatórios.
- 4ª Variação:

- 50 clientes distribuídos pelos *Nodes*.
- 100000 *channels*;
- Identificador de *channels* em conjunto de 40 caracteres aleatórios.

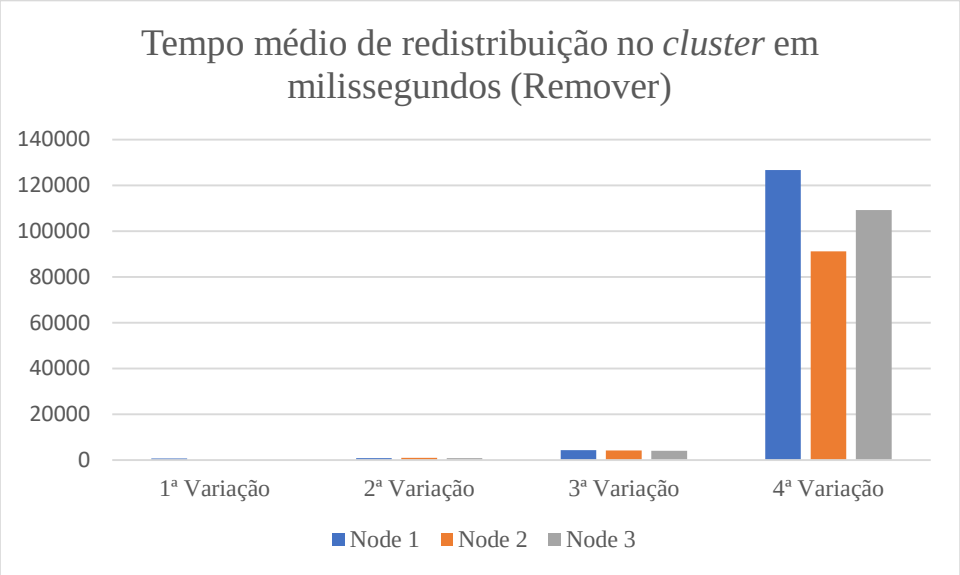
A diferença nos identificadores do *channels* deve-se ao posicionamento no *hash ring*. Na primeira variação grande parte dos identificadores *channels* eram semelhantes, o que levava que estes tivessem o mesmo *Node* como responsável. Com os identificadores gerados aleatoriamente houve uma melhor distribuição pelos *Nodes*, assim como pode ser observado nos resultados representados na figura 27 e figura 28.

Figura 27 - Tempo médio de redistribuição no cluster em milissegundos (Adicionar)



Fonte: Própria

Figura 28 - Tempo médio de redistribuição no cluster em milissegundos (Remover)



Fonte: Própria

A diferença entre de desempenho entre o adicionar ou remover um membro do *cluster* deve-se ao processo de remover um membro do cluster ser mais eficiente. Neste processo sabemos o identificador do *Node* que saiu o que permite que seja calculado de forma eficiente quais *channels* devem ser movidos, adicionalmente, numa redistribuição ao adicionar um membro é necessário notificar o membro do *cluster* previamente responsável de que não tem mais interesse nos *channels* a que estes pertenciam, este passo não acontece quando um membro é removido. O tempo de ajuste destas variações vai subindo de acordo com o número de *channels* ativos no *cluster*, sendo que quando observamos o salto da segunda para a terceira variação do teste o tempo de ajuste sobe consideravelmente, inclusive não foi possível realizar a quarta variação do teste devido ao tempo que o *cluster* fica em ajustes. Após analisar os o que leva a este considerável aumento, foi identificado que o atraso se deve à forma como a movimentação dos *channels* e das conexões *gRPC* para os novos responsáveis é realizada. Este processo é feito de forma individual, ou seja, um *channel* de cada vez. Este processo foi melhorado drasticamente, agrupando todos as alterações a serem realizadas por membro num conjunto e enviar somente uma mensagem por cada conjunto de operações, adicionalmente, este processo foi também paralelizado. Esta alteração resultou nos resultados apresentados da quarta variação.

Quanto ao tempo que leva a adicionar ou remover um *Node* ao *cluster* é de aproximadamente 16ms, para a deteção de falha do tempo é entre 120 a 500ms. De forma a obter os 16ms, foi registado o tempo em que o Node descobre os endereços dos outros membros e o tempo em

que este se juntou a pelo menos um dos membros do *cluster*. Para o tempo de deteção de falha foi comparado o tempo em que o *Node* foi terminado com o tempo que um dos *Nodes* detetou a falha, os intervalos entres estes dois tempos foram muito variados sendo os valores mais comuns entre 120 a 500ms.

Quanto às fases de teste com tráfego real, não foi possível obter todos os resultados antes da produção deste documento. Visto que o projeto ainda se encontra na segunda fase de testes, as análises de resultados são bastante limitadas.

Nas primeiras duas fases de testes com tráfego real, foi observado que o número de sessões em simultâneo era consideravelmente inferior ao esperado, no entanto, é possível verificar que o número de sessões é distribuído ao longo do dia, com picos relativamente pequenos.

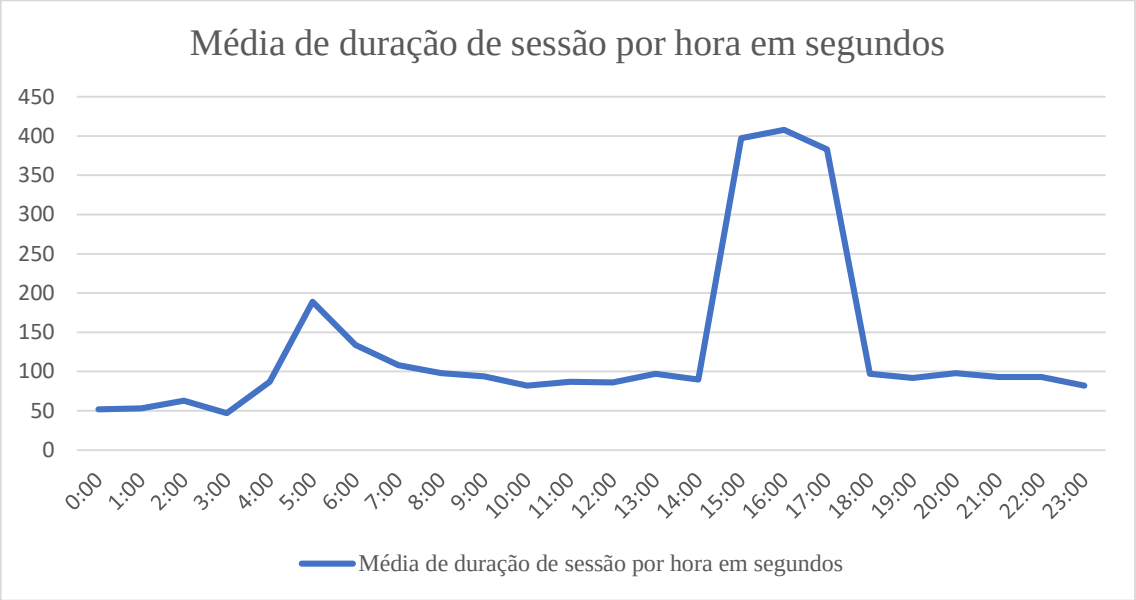
Na figura 29, podemos observar a distribuição de sessões ao longo do dia, esta visualização acumula as sessões dos *tenants* em fase de teste.

Figura 29 - Distribuição de sessões por hora

Fonte: Própria

Na figura 30, temos a representação da duração média de sessão a cada hora do dia. Assim como pode ser observado as durações das sessões são relativamente baixas, sendo assim com o apresentado as 16 horas na mesma figura.

Figura 30 – Média de duração de sessão por hora



Fonte: Própria

Até ao momento de que este documento foi desenvolvido, já foram em enviadas 12 706 780 mensagens e recebidas 12 706 690 mensagens, com um total de 173 977 sessões estabelecidas com 2802 sessões diárias em média, e com soma de duração destas de 23 757 458 segundos ou aproximadamente 6600 horas e duração média de 2 minutos.

Devido à falta de métricas coletadas pelo sistema anterior não é possível fazer uma comparação entre os resultados novo sistema com os resultados do sistema anterior.

6 CONCLUSÃO

Em conclusão, o novo sistema distribuído apresenta uma solução satisfatória para os problemas encontrados no sistema antigo, com melhorias significativas na escalabilidade, na tolerância a falhas e na monitorização. Adicionalmente, todas as funcionalidades do antigo sistema foram mantidas enquanto novas foram adicionadas, permitindo também adicionar futuras funcionalidades graças à sua arquitetura. Embora algumas funcionalidades, como *Streams* e *push notifications*, ainda não estejam completas, foram identificadas oportunidades para melhorias futuras.

O novo sistema atingiu todos os objetivos estabelecidos, incluindo escalabilidade horizontal, comunicação bidirecional entre cliente e servidor, comunicação utilizando *Pub/Sub* em tópicos, restrição de acesso a tópicos, suporte para múltiplos *tenants*, criação explícita de tópicos, rastreamento de presença de clientes em cada tópico e armazenamento de mensagens enviadas em cada tópico.

Os próximos passos para o sistema são planear o escalamento global e o *routing* inteligente, tendo em conta a latência entre o servidor e o cliente, garantindo que o sistema possa lidar com um número maior de utilizadores e volume maior de dados.

PARTE II – ARTIGO CIENTÍFICO

AppSockets

Tiago Marques Soares Lima
Estudante do 3ºano da Licenciatura em Engenharia Informática
do ISTEC Porto

Resumo: Este artigo descreve uma aplicação para um sistema de comunicação em *soft real-time*, com o objetivo de substituir um sistema anteriormente utilizado. Através do protocolo *gossip*, *hash ring* e *gRPC*, foi criada uma aplicação distribuída que é capaz de escalar horizontalmente, permitindo a substituição do sistema anterior utilizado na empresa *NAPPS*, enquanto matem todas as suas funcionalidades.

Palavras-chave: *Soft Real-Time*, Publicar/Subscrever, *WebSockets*, *Message Broker*

Abstract: *This article describes an application for soft real-time communication system, with the purpose of replacing a previously used system. Using the protocol gossip, hash ring, and gRPC technologies, a horizontally scalable distributed application was created, which is capable of replacing the previous system at NAPPS while maintaining all of its functionalities.*

Keywords: *Soft Real-Time, Publish/Subscribe, WebSockets, Message Broker.*

I. Introdução

Neste artigo é explicado a criação de infraestrutura para o envio de informação em *soft real-time* entre clientes e servidores, e ao mesmo tempo substituir um sistema com objetivos similares, mantendo o máximo de compatibilidade possível de forma a facilitar a migração para o novo sistema.

A. Motivação

A aplicação desenvolvida consiste num sistema *Publish/Subscribe* com suporte para múltiplos *tenants*, onde existem elementos que subscrevem a um tópico (*Subscribe*) e recebem todas mensagens ou eventos publicados neste mesmo tópico (*Publish*).

O sistema criado tem como propósito substituir o sistema anterior enquanto mantém todas as suas funcionalidades, adiciona novas funcionalidades e facilita a sua utilização.

As motivações para o desenvolvimento deste novo sistema foram baseadas em alguns pontos principais, sendo estes:

- O sistema a ser substituído não ser horizontalmente escalável;
- A não existência de ferramentas de monitorização e deteção de erros;
- Arquitetura não preparada para novas funcionalidades;
- Falta de testes no projeto.

O sistema a ser substituído, foi desenvolvido de forma rápida, e durante o seu desenvolvimento não existia a necessidade de que este fosse horizontalmente escalável, e a adaptação seria complicada exigindo modificar grande parte do seu funcionamento. Inicialmente, este sistema foi projetado para ser utilizado maioritariamente por *dashboards* e *backoffices* como subscritores enquanto alguns eventos eram emitidos por outros servidores. No entanto, novas funcionalidades a serem planeadas

necessitam que a utilização deste sistema seja ampliada para a aplicações móveis, onde existe um valor muito mais elevado de conexões a serem realizadas, de forma a quantificar a diferença de conexões esperadas, no sistema a ser substituído era somente esperado ter no máximo 50 conexões diárias, um valor muito baixo, enquanto o valor esperado para os utilizadores atuais é de aproximadamente 9000 conexões, um valor muito superior. Adicionalmente, sempre que se adquira um novo cliente, é esperado que este valor suba entre algumas centenas a alguns milhares (aproximadamente entre 600 e 2000), sendo que o novo sistema tem de ser capaz de capaz de suportar este aumento de utilizadores. Outro ponto relacionado com a necessidade de escalar horizontalmente, consiste em permitir que o sistema seja tolerante a falhas, algo que não é possível se somente um servidor puder ser executado ao mesmo tempo. O motivo pelo qual o sistema não é horizontalmente escalável, deve-se ao funcionamento geral de um sistema de comunicação *PubSub*, onde independentemente do servidor o cliente está conectado, este tem de receber eventos que podem ser enviados noutros servidores.

A inexistência de ferramentas de monitorização e de deteção erros dificulta a manutenção do sistema, no entanto, sendo que nenhuma funcionalidade em que este era utilizada era considerada crítica, não houve nenhum incentivo para desenvolver estas, no entanto, sendo que este sistema passou a ser utilizado por clientes finais, é importante ser capaz de identificar os erros o mais rápido possível, assim como ser capaz de monitorizar a sua utilização de forma a planear o melhor possível o escalamento automático.

II. Objetivos

Tendo sido explicado os problemas que levaram a desenvolver um novo sistema, é necessário definir os objetivos a serem cumpridos pelo novo sistema, lembrando que o novo sistema vai substituir um existente, é necessário que este seja capaz de suportar os casos de uso atuais, assim como criar o máximo de compatibilidade possível. Portanto, sendo os requerimentos do novo sistema similares com o anterior em produção, é necessário analisar como o atual funciona e ver que problemas apresenta. Os principais pontos a ter em conta no projeto são:

• Comunicação bidirecional entre cliente e servidor através *WebSockets*;
• Comunicação usando *Pub/Sub* (publicar e subscrever) em tópicos (nomeados de *channels*);
• Restringir o acesso a tópicos de acordo com as autorizações;
• Suporte para múltiplos *tenants*, existindo configurações por cada *tenant*;
• Criação explícita de tópicos e com configurações por cada;
• Rastreamento da presença dos clientes em cada tópico;
• Armazenamento das mensagens enviadas em cada tópico.

III. Estado da Arte

Nesta parte vai ser mencionado técnicas utilizadas para enviar informação em tempo real tem evoluído, protocolos que tenham vindo a ser criados e qual foi o escolhido para este projeto.
Adicionalmente, são selecionados projetos de código aberto e serviços comerciais que podem potencialmente ser utilizados de forma a tentar a cumprir os objetivos deste projeto.

A. Evolução de comunicação em tempo real

Comunicação em tempo real não é um tópico novo e está presente em várias aplicações, principalmente em aplicações de mensagens, no entanto, em aplicações *web* nem sempre existiu uma forma de criar uma ligação bidirecional entre cliente e servidor. Sendo necessário que aplicações *web* tenham a possibilidade de realizar uma comunicação com os servidores primeiro é necessário conhecer as opções existentes e como estas foram evoluindo.

Inicialmente, em aplicações *Web* não existia a possibilidade de criar ligações bidirecionais com servidores utilizando as *APIs* fornecidas pelos *browsers*, de forma a resolver esta limitação, em 2011 um novo protocolo foi padronizado *Fette* e *Melnikov* [1] como *RFC 6455*, este protocolo ficou conhecido como *WebSockets* e é atualmente a forma padrão de comunicação bidirecional com servidores em aplicações *Web*. Em outras aplicações não *web*, estas limitações não existiam, portanto cabia a cada desenvolvedor utilizar a sua implementação ou reutilizar uma existente.

Antes da criação do protocolo *WebSockets*, a técnica *long polling* era uma forma comum de simular comunicação bidirecional, assim como mencionado pela *Internet Engineering Task Force* [1], "web applications that need bidirectional communication between a client and a server [...] has required an abuse of HTTP to poll the server for updates while sending upstream notifications as distinct HTTP calls" (*The WebSocket Protocol)* (capítulo 1.1, 1º parágrafo), visto que os pedidos *HTTP* funcionam como *request-reply* (pergunta-resposta) de forma unidirecional (cliente para servidor), não existia forma de um servidor notificar o utilizador que um evento tenha acontecido no momento, ou seja, uma aplicação cliente teria que periodicamente realizar um pedido *HTTP* ao servidor de forma a verificar que novos eventos tenham ocorrido. Tendo como exemplo uma aplicação de chat, onde existem largos períodos sem atividade, é possível que grande parte destes pedidos não tenham informação nova desperdiçando recursos, ou então, caso o período entre pedidos seja longo é possível que demore demasiado tempo para receber nova informação. Utilizando o mesmo exemplo, numa conversa entre duas pessoas e com intervalo entre pedidos de 5 segundos, uma mensagem pode demorar até esse mesmo intervalo só para ser recebida pela outra pessoa.

De forma a evitar a quantidade de pedidos realizados e a reduzir o tempo que demora a receber informação, o servidor artificialmente demora mais tempo para enviar uma resposta, esperando que exista nova informação ou que tempo limite de conexão tenha sido atingido. Esta parte é a origem do nome *Long* na técnica *Polling*. Desta forma, o tempo de atraso a receber a mensagem seria no máximo o tempo de receber a última resposta mais o tempo de iniciar um novo pedido, algo que poderia demorar segundos que passou para milissegundos, além de reduzir consideravelmente a quantidade de pedidos a serem feitos.

Com a criação do protocolo *WebSockets*, a técnica *long polling* deixou de ser usada em novos projetos e serve como alternativa caso uma conexão *WebSocket* não seja possível. No entanto, embora *WebSockets* seja o padrão existem outras opções para permitir que o servidor comunique com o cliente tais como: *Server-Sent Events*, *Web Push* e *HTTP Streaming*.

Server-Sent Events ou *SSE*, assim como definido por *Roome* e *Yang* [2] no *RFC 8895*, permite ao servidor enviar informação para o cliente por *HTTP* pela duração da conexão, ao contrário do protocolo *WebSockets*, este somente permite uma comunicação unidirecional de servidor para cliente, e não suporta o envio de informação em formato binário. Visto que este protocolo somente permite o envio de informação de servidor para cliente, pedidos adicionais têm de ser feitos caso o

cliente precise de enviar informação para o servidor.

O *Web Push,* conforme definido por *Thomson* e *Damaggio* [3] no *RFC 8030*, torna possível o envio de informação para o cliente, no entanto, este costuma ser utilizado para o envio de notificações e não de dados em geral, sendo as mensagens enviadas acompanhadas por título, conteúdo, e exigem que os clientes aceitem uma permissão para receber esta informação. Embora esta opção não seja adequada para envio de informação em tempo real, esta pode servir como alternativa para o envio de informação quando é necessário que a informação seja recebida mesmo que o cliente não esteja ligado a um dos servidores.

O *HTTP Streaming* é relativamente similar ao *Server-Sent Events*, este também permite o envio de informação para o cliente de forma unidirecional. Este funciona enviando informação sem tamanho definido, pondo a aplicação cliente constantemente à espera dos próximos dados até a conclusão do pedido HTTP.

Tendo revisto os meios de comunicação disponíveis, o protocolo *WebSockets* aparenta ser a melhor opção, principalmente por ser o protocolo padrão na indústria e pela sua capacidade de comunicação bidirecional, no entanto, outros protocolos poderão ser implementados quando a bidirecionalidade não for necessária, preferencialmente utilizando *Server-Sent Events*.

Escolhido o protocolo *WebSockets*, convém conhecer o seu funcionamento, assim como mencionado previamente, este permite comunicação bidirecional entre cliente e servidor, esta é estabelecida utilizando *HTTP* inicialmente que após um *handshake* é estabelecida. Este protocolo é fundamentalmente dividido em duas partes: o *handshake* e a transferência de dados.

No *handshake,* o pedido é realizado pelo cliente enviando a intenção de transformar a conexão unidirecional em uma bidirecional (com o nome de *Upgrade* no protocolo) ao qual o servidor deverá responder que está a trocar o protocolo, após esta parte a conexão é considerada estabelecida. Na transferência de dados, é usado o conceito de mensagens, sendo cada composta por um ou mais *frames*. Cada *frame* tem um tipo associado, tendo cada *frame* pertencente à mesma mensagem o mesmo tipo. De forma geral, existem 3 tipos de dados, sendo textual, binário e de controle. No tipo textual a informação é interpretada como *UTF-8* enquanto no tipo binário a interpretação é deixada à responsabilidade da aplicação, para o controle, que não tem como objetivo transferir dados da aplicação, são usados como sinalização da conexão, como por exemplo *PING*, *PONG* e *CLOSE*. Estes últimos *PING* e *PONG* tem como propósito verificar se a conexão ainda se encontra ativa, principalmente quando a aplicação envolve pouco tráfego. O transporte de mensagens numa conexão com protocolo *WebSocket* é similar a uma conexão *TCP*, este apenas junta um mecanismo de *framing* que reduz essa responsabilidade na aplicação, quanto ao formato dos *frames* não será mencionado tendo em conta que não faz parte do objetivo deste documento.

B. Soluções existentes

Tendo em conta o protocolo escolhido e os pontos a serem considerados, foi realizada pesquisa sobre soluções já existentes que suportam os pontos definidos e ao mesmo tempo tentar perceber de que forma estas soluções estruturam as soluções e o que estas permitem. Estas soluções incluem tanto projetos e bibliotecas de código aberto como serviços, as principais soluções encontradas são as seguintes.

- Código aberto*:*
    - *Centrifugo;*
    - *Mercure;*
    - *Phoenix;*
    - *VerneMQ;*
    - *Emitter;*

    - *HiveMQ;*
    - *EMQX;*
    - *SocketCluster;*
    - *Soketi;*
    - *Signal-R;*
- Serviços:
    - *Ably;*
    - *PubNub;*
    - *Pusher;*
    - Fanout.

Deste conjunto existem algumas opções que funcionam como um *broker* de mensagens, utilizando protocolos já existentes como *MQTT*, deste conjunto temos os seguintes:

Centrifugo [4] é uma aplicação que serve como um b*roker* de mensagens. Esta aplicação suporta a distribuição de mensagens com os protocolos *WebSockets* e *gRPC* e com o envio de mensagens por pedido *HTTP*. É possível de escalar horizontalmente utilizando através da utilização de um dos *engines* suportados pela aplicação. De forma a permitir que os clientes possam enviar mensagens, estes precisam de uma autorização extra criada por servidores, ou que estes sirvam como intermediários para o envio de mensagens.

O *Mercure* [5] é um *broker* de mensagens, com distribuição de mensagens utilizando *SSE* (unidirecional) e com o envio de mensagens por pedido *HTTP*. A possibilidade de escalar horizontalmente exige o uso de um serviço oferecido pelos desenvolvedores para a gestão da infraestrutura. Sendo o protocolo de comunicação principal *SSE* este remove a possibilidade de comunicação bidirecional, para que os clientes possam enviar mensagens, precisam de uma autorização extra criada por servidores, ou que estes sirvam como intermediários para o envio de mensagens.

O *Phoenix Framework* [6] é um *framework* para a linguagem de programação *Elixir*, com suporte para comunicação em tempo real e escalável horizontalmente. Sendo desenvolvido em *Elixir* permite a utilização da *Erlang VM*, desenvolvida com suporte para tolerância a falhas e maioritariamente utilizada em sistemas de telecomunicações tornando uma excelente escolha. O protocolo de comunicação é utilizado é *WebSockets* e tem suporte para praticamente todos os outros protocolos sendo *WebSockets* o principal. Infelizmente, *Elixir* ou *Erlang* são linguagens ao qual não existe conhecimento interno para a sua utilização.

VerneMQ [7], *HiveMQ* [8], *Emitter* [9] e *EMQ* [10] são tecnologias são baseadas no protocolo *MQTT*, embora com algumas diferenças nas suas implementações, todas estas oferecem possibilidade de escalar horizontalmente. A utilização do protocolo *MQTT* permite que a comunicação seja feita diretamente por *TCP* ou *WebSockets*. O protocolo *MQTT* tem como meio de comunicação principal *Pub/Sub*, no entanto, algumas funcionalidades extras podem a vir ser necessárias, algo que podem ser implementadas utilizando tópicos no *MQTT*.

SocketCluster [11] é uma biblioteca de *javascript* que permite a comunicação no formato de *Pub/Sub* e é capaz de escalar horizontalmente. Infelizmente a documentação não é extensiva, principalmente quanto ao subprotocolo. Adicionalmente, esta opção tem como objetivo primário servir como processador direto das mensagens recebidas, enquanto o objetivo pretendido é somente a distribuição, mas é possível adaptar para o caso necessário.

Soketi [12] é um servidor de *WebSockets* compatível com o subprotocolo *Pusher v7*, permitindo que clientes desenvolvidos para esta plataforma possam ser reutilizados, adicionalmente, é capaz de escalar horizontalmente através da aplicação Redis.

Signal-R [13] é uma biblioteca criada pela *Microsoft* que oferece a possibilidade

de comunicação em tempo real com clientes, esta biblioteca funciona somente em servidores desenvolvidos em *C#* com a tecnologia *ASP.NET*. Esta opção é capaz de escalar utilizando a aplicação adicional *Redis* ou um serviço desenvolvido pela *Microsoft* disponível na *Azure Cloud*.

Desta lista de opções com código aberto a opção que mais se adequa é o *framework Phoenix*. Este é desenvolvido em *elixir* que por sua vez é executado na *Erlang VM*, a qual tem acesso a um conjunto de bibliotecas nomeadas de *OTP (Open Telecom Platform)* que facilita o desenvolvimento de aplicações distribuídas. Adicionalmente esta linguagem é utilizada por grandes plataformas como *WeChat* e *WhatsApps*, que servem como comprovativo para a sua escalabilidade. No entanto, *Elixir* ou *Erlang* são linguagens ao qual não existe conhecimento interno para a sua utilização.

Quanto à opção *Mercure*, esta não suporta o envio de mensagens bidirecionais, incluindo de clientes não autenticados, este exige que outros servidores sejam capazes de enviar mensagens pelos clientes ou que sirvam como meio de autenticação dos mesmos.

VerneMQ, *HiveMQ*, *Emitter* e *EMQ* são possíveis opções, no entanto estas ficam somente pelo protocolo *MQTT*, no entanto funcionalidades adicionais além das definidas no protocolo *MQTT*, teriam de ser desenvolvidas à parte, visto que o suporte para modificações é relativamente reduzido.

A opção *Soketi*, apresenta dois problemas, primeiro ser desenvolvida em *javascript* que por sua vez é executado em *node.js*, embora seja plataformas viáveis, este tipo de aplicação exige processamento simultâneo e paralelismo, tendo em conta que o *node.js* é executado como um processo de um único *thread*, este apresenta desvantagens quanto as outras possibilidades, adicionalmente, para utilizar eficientemente os recursos disponíveis seria necessário várias instâncias da mesma aplicação a correr em simultâneo com espaços de memória separados.

Por fim, *Signal-R* é uma boa opção para empresas que já usam *C#*, no entanto, este não é o caso, adicionalmente, de forma a escalar horizontalmente a aplicação *Redis* pode ser utilizada, mas o principal método é com um serviço desenvolvido pela *Microsoft* disponível na *Azure Cloud*, algo que também não é usado internamente.

Quanto aos serviços, a maior parte destes oferecem uma plataforma para a comunicação em tempo real, com suporte com vários protocolos e com escalabilidade gerida, no entanto, grande parte destes tem limitações no número de conexões.

Ably [14] é uma plataforma de mensagens *Pub/Sub* com garantia de envio, ordem de envio, e com suporte para vários protocolos tais como *MQTT, STOMP, AMQP, PUSHER* e *PubNub*. Permite conexões com os protocolos *WebSockets*, *SSE* e o envio de mensagens por *HTTP*. Adicionalmente, permite o rastreamento da presença dos clientes, envio de notificações *push*, oferece um histórico de mensagens e com suporte para restaurar desconexões abruptas.

PubNub Inc [15] é uma plataforma de mensagens *Pub/Sub* sem garantia de envio ou ordem de envio, os protocolos utilizados não são especificados, no entanto, segundo os exemplos apresentados utilizam a técnica *long-polling*. Esta plataforma também permite o rastreamento da presença dos clientes, envio de notificações *push* e processamento de mensagens enviadas.

Pusher Ltd [16] é uma plataforma similar às anteriores, funciona igualmente com mensagens *Pub/Sub* mas sem garantia de ordem e envio. Esta utiliza conexões com o protocolo *WebSockets* e sub-protocolo *Pusher*, um protocolo proprietário. Assim como as opções anteriores também permite o rastreamento

da presença dos clientes. Algumas funcionalidades que não oferecem são um histórico de mensagens, recuperação de mensagens perdidas. Notificações *push* são possíveis, mas fazem parte de um serviço à parte oferecido pela mesma empresa.

Fanout [17] é uma que opção oferece tanto uma versão com código aberto quanto um serviço. A opção de código aberto serve como um intermediário entre outros serviços onde estes podem enviar atualizações para serem distribuídas pelos clientes, esta opção não é horizontalmente escalável sem adaptação dos serviços para o envio de mensagens utilizando um protocolo de comunicação *ZeroMQ* ou então publicando para todas as instâncias. A versão de serviço, oferece mais funcionalidades, como organização de *channels* (equivalente a um tópico) por *realms* (um elemento que agrupa *channels*). Ambas opções permitem conexões com os protocolos *WebSocket*, *SSE* e *long-polling*. Ao contrário das opções anteriores, o rastreamento da presença de clientes, envio de notificações *push* não suportadas, adicionalmente o suporte para ordem e garantia de envio das mensagens é parcial.

Estas quatro opções, são plataformas que oferecem uma maior abstração aos sistemas *Pub/Sub*, estas oferecem funcionalidades tipicamente não existentes em um *message broker* tais como o rastreamento de presenças, envio de notificações *push* e histórico de mensagens. Desenvolver estas funcionalidades em algumas das opções apresentadas que não as oferecem necessitam modificações no projeto em si, algo que iria exigir familiaridade com o funcionamento interno destes. Quanto às plataformas apresentadas, nomeadamente *Ably*, *PubNub*, *Pusher* e *Fanout*, as que mais cumprem os pontos a ter em consideração são *Ably*, *Pusher* e *PubNub* na ordem que melhor cumprem. Embora estas opções não tenham integração com a aplicação *NATS*, seria possível adaptar para o que a plataforma oferece ou então desenvolver uma ferramenta adicional que se realiza a conversão.

A plataforma *Pusher* quando falamos de meios de comunicação e funcionamento dos mesmos, cumpre os requisitos, incluindo o rastreamento de presença através de tópicos especializados para o caso, tópicos públicos e privados utilizando um prefixo no seu nome. No entanto, nenhum dos tipos de tópicos tem a capacidade de armazenar um histórico de mensagens. Outro problema comum em plataformas, que ocorre neste caso é o número de conexões, cada loja tem a sua aplicação e o seu conjunto de clientes, e a empresa tem de estar preparada para uma elevada quantidade de conexões em simultâneo, no caso do *Pusher* o plano maior listado oferece no máximo 30 mil conexões, exigindo além disso negociar com a empresa.

A plataforma *PubNub*, não estabelece conexões utilizando o protocolo *WebSocket*, em vez disso utiliza pedidos *HTTP* e uma espécie de *long-polling* o custo de performance e energia para as aplicações acabar por ser mais elevado, e não sendo uma conexão bidirecional este não permite o envio bidirecional de mensagens, no entanto, todas outras funcionalidades estão presentes.

Por fim, *Ably* é a plataforma que melhor cumpre os pontos previamente mencionados, esta permite conexões por *WebSockets* e outros protocolos, rastreamento de presenças, garantia na ordem e entrega de mensagens, armazenamento opcional das mensagens, e agrupamento de tópicos permitindo um conjunto de tópicos ter a mesma configuração. No entanto, assim como no *Pusher* o limite de conexões se mantém.

C. Solução personalizada

Após todas estas possibilidade terem sido analisadas, foi decidido desenvolver um novo sistema em vez de reutilizar as

opções mencionadas pelos seguintes motivos:

- Extensibilidade;
- Limites da *API*;
- Adaptação ao caso de uso;
- Imprevisão de custo;
- Conhecimento existente na empresa.

Muitos destes serviços oferecem sistemas simples de *PubSub*, no entanto, pouca personalização além disso, sendo que caso seja necessário funcionalidades além das oferecidas em conjunto com o sistema *PubSub*, é necessário as desenvolver num sistema separado. Por exemplo, um sistema de presença em conjunto com meta dados sobre todos utilizadores subscritos num tópico é uma funcionalidade que pode estar embutida num tópico, mas desenvolver um sistema só para esta funcionalidade não é prático.

Dentro de todos serviços apresentados, o que mais se destacou por ser o mais próximo de atender a todos requisitos é o serviço *Ably*, no entanto, assim como os serviços em geral apresenta limites na utilização da sua *API*, como por exemplo, limites de eventos num tópico por segundo e máximo de utilizadores subscritos num *channel*. Adicionalmente, sendo o número de conexões um valor que flutua bastante, assim como o número de eventos enviados para tópicos, prever os custos dos serviços torna-se difícil e sem forma de implementar um teto máximo

Quanto às opções de código aberto, muitas destas não cumprem os requisitos necessários, sendo necessário adaptar os projetos e ter o custo extra de manutenção de manter o projeto atualizado com novas funcionalidades implementadas no código base. De todas as opções, a que melhor cumpre os requisitos necessários é o *framework Phoenix*, utilizando a tecnologia presente na *Erlang VM* este permite criar um sistema distribuído, e adicionalmente o *framework Phoenix* permite customizar o funcionamento dos tópicos. No entanto, este *framework* utiliza as linguagens *Elixir* e *Erlang*, que são linguagem ao qual não existe conhecimento interno para sua utilização.

Tendo esta informação em conta, a criação de um novo sistema foi o caminho decidido de forma reutilizar o conhecimento existente da linguagem *Go* [18] e ferramentas já utilizadas internamente como a aplicação *NATS*.

IV. Metodologia

Para levantamento de requisitos, foi usado como base o sistema já presente em produção, visto que grande parte das suas funcionalidades são necessárias por outros serviços dentro da empresa NAPPS. Estando a substituir um sistema em utilização internamente, já existe um conhecimento prévio de problemas que existiam, ou melhorias desejadas. Portanto, utilizando o *feedback* dos utilizadores do sistema, em conjunto com funcionalidades futuras previstas, foi realizado um *brainstorming* onde se definiu o que o projeto precisava, assim como vai ser visto ao longo deste documento.

A. Tarefas

O projeto está dividido em várias tarefas, algumas das tarefas vão envolver vários pontos que serão descobertos ao longo da fase de pesquisa e possivelmente em adaptações a novas funcionalidades. As tarefas definidas até ao momento são:

- Tarefa 1 – Pesquisa de possíveis soluções existentes e avaliação das mesmas;
- Tarefa 2 – Pesquisa do funcionamento das atuais soluções;
- Tarefa 3 – Elaborar funcionamento do projeto;
- Tarefa 4 – Avaliar possíveis problemas de migração para novo projeto;
- Tarefa 5 – Desenvolvimento de protótipo;

- Tarefa 6 – Teste de protótipo e avaliar possíveis problemas;
- Tarefa 7 – Corrigir possíveis problemas ou adaptar para possíveis utilizações;
- Tarefa 8 – Teste em Cloud (AWS);
- Tarefa 9 – Criação de testes para cobrir lógica de projeto;
- Tarefa 10 – Implementação em produção em fase de teste.

Após mencionadas as tarefas para realização, passo a elaborar o que cada constitui.

Na tarefa 1, é realizada uma pesquisa por possíveis soluções comerciais ou de código aberto e análise rápida se estas podem cobrir os casos de utilização atual, na tarefa 2 após a eliminação de soluções que não se adaptam aos casos de utilização, iremos verificar mais profundamente o seu funcionamento, e como se comportaria em funcionalidades planeadas e custos para as mesmas. Utilizando o conhecimento do funcionamento obtido pelas tarefas 1 e 2, é elaborado um plano geral com todas as funcionalidades necessárias e o seu funcionamento interno, após esta será elaborado uma análise de problemas que possam existir ao realizar a migração do projeto anterior para o atual, quanto menor o custo de migração menor será o tempo para introduzir em produção e atualização de sistemas em produção, e esta etapa será a tarefa 4.
Após ter sido realizada uma análise do funcionamento e tendo sido verificado possíveis partes problemáticas, é realizado o desenvolvimento de um protótipo do projeto como tarefa 5, os testes mais manuais serão realizados e serão avaliados possíveis problemas que tenham ocorrido, este passo corresponde à tarefa 6, para tarefa 7, será a correção dos erros que tenham sido encontrados e adaptação para funcionalidades que tenham surgido ou adaptação das atuais.
Por fim, o funcionamento será testado na *cloud AWS* e o desenvolvimento de testes e ferramentas de análise para ser possível inspecionar os funcionamento e erros que ocorram com o projeto em funcionamento na *cloud*, e como última etapa o projeto será posto em produção, mas em fase de teste com tráfego real, mas em componentes que não sejam críticos, estas três partes serão as tarefas 8, 9 e 10.

V. Desenvolvimento

Neste capítulo são apresentadas as decisões que foram tomadas inicialmente, nomeadamente a estrutura inicial e a utilização da aplicação *NATS*. Também é apresentado os motivos que levaram a desconsiderar a aplicação *NATS*, assim como a alternativa que foi implementada e por fim as funcionalidades existentes na aplicação.
Portanto seguindo o sistema anterior, existe somente um servidor onde todos os clientes estão conectados. Caso este servidor falhe, os clientes ficam sem forma de utilizar o serviço. De forma a evitar que isso aconteça, é necessário adicionar mais servidores, assim caso um falhe existem outros que podem receber as conexões. A isto nomeamos de ser horizontalmente escalável, caso um servidor falhe ou não seja capaz de aguentar o número de clientes atual, existem outros servidores para receber estes clientes.
No entanto, quando falamos num sistema *PubSub* é necessário que quando um evento é publicado num tópico, este tem de ser transmitido para todos os clientes subscritos neste mesmo tópico, independentemente a qual servidor estes estão conectados. De forma a resolvermos este problema, tinha sido inicialmente planeado a utilização da aplicação *NATS*, para realizar a intercomunicação entre os servidores, desta forma, sempre que um evento é publicado pelo cliente, é transmitido pelo *NATS* para todos os servidores interessados no tópico.
Com esta solução, foi observado um problema com a utilização do *NATS*, sendo este a ineficiência introduzida na passagem de um evento, principalmente quando é

aumentado o número de servidores em funcionamento.
Na figura 31, existem 3 servidores e 3 instâncias da aplicação *NATS*. Existe um número mais elevado de servidores de forma a ter redundância em caso de falhas e de forma a ser capaz de receber um maior número de conexões. Nesta figura temos o Cliente 1 que publica um evento que tem de chegar aos Servidores 2 e 3, para isso, assim que o Servidor 1 receba o evento publicado pelo Cliente 1, este tem de enviar o evento para o *NATS* 1, que por sua vez envia para o *NATS* 2 e 3 que por fim enviam aos Servidores 2 e 3. Portanto, foi necessário que o evento fosse passado 5 vezes por rede, relembrando que cada passagem exige a codificação da mensagem por quem envia e descodificação por quem recebe.

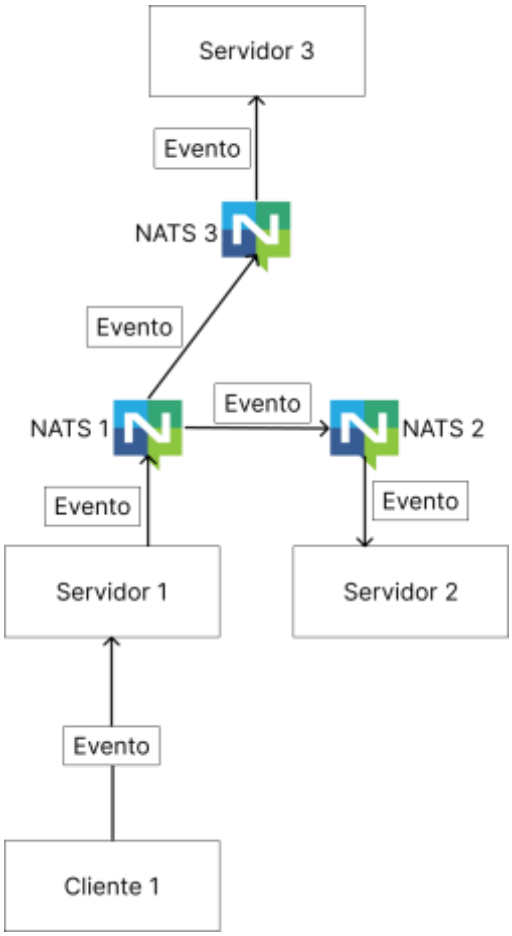


Figura 31 - Exemplificação da ineficiência da intercomunicação com a aplicação NATS

De forma a evitar esta ineficiência, podemos ter os servidores a comunicar entre si em vez de utilizar a aplicação *NATS* como intermediário. Para isso, é necessário implementar algo que seja capaz de substituir a utilização da aplicação *NATS*, ou seja, é necessário resolver os 3 seguintes pontos:
• Consenso;
• Intercomunicação;
• Distribuição.

O consenso consiste em ter conhecimento de quais servidores estão ativos. A utilização da aplicação *NATS* evitava esta necessidade, afinal os eventos eram publicados no *NATS* e este iria distribuir o evento por quem está interessado. A intercomunicação consiste no envio de informações e eventos entre os servidores. A distribuição consiste em como os tópicos ou *channels* são distribuídos pelos servidores.

A. Consenso

O consenso é um dos problemas fundamentais em sistemas distribuídos, este exige que múltiplos membros concordem em um conjunto de valores mesmo na presença de falhas. Um protocolo de consenso que seja capaz de tolerar falhas deve cumprir as seguintes propriedades: terminação, sendo que eventualmente todos membros concordam com um valor; integridade, caso os membros proponham o mesmo valor, então outros devem decidir no mesmo valor; concordância, todos membros devem concordar no mesmo valor. Algoritmos de consenso tendem a confirmar um valor quando a maioria dos membros do *cluster* esteja disponível, por exemplo, um *cluster* de 5 membros pode continuar a operar com a falha de dois membros, no entanto, caso mais que dois falhem estes deixam de conseguir alterar os valores e somente retornam os valores previamente acordados.
Portanto, o consenso em sistemas distribuídos é um processo em que vários membros de um sistema distribuído trabalham em conjunto para tomar uma

decisão em comum. Este processo é necessário quando há vários componentes no sistema e é preciso chegar a um acordo sobre qual ação deve ser tomada. Por exemplo, num *cluster*, é necessário que todos os membros saibam qual membro é responsável por determinada tarefa ou quais dados estão disponíveis em cada membro. Para alcançar o consenso, os sistemas distribuídos utilizam algoritmos de consenso, como o algoritmo *Paxos* ou o algoritmo *Raft*, que são projetados de forma a garantir que todos os membros no sistema tenham a mesma visão dos dados e das ações a serem tomadas. Estes algoritmos permitem que os membros elejam um líder ou coordenador que tomará as decisões, enquanto os outros membros seguirão as instruções do líder. O consenso em sistemas distribuídos é fundamental para garantir a consistência e a integridade dos dados em todo o sistema. Neste são considerados os protocolos *Raft* e *Gossip* como protocolos de consenso.

B. Raft

Raft é um algoritmo de consenso com propósito de ser simples de compreender, este é equivalente ao algoritmo *Paxos* a nível de tolerância de falhas e desempenho. Este atinge o consenso através de um e somente um líder eleito. Neste protocolo, cada membro tem o cargo de líder ou seguidor e pode ser um candidato caso um líder não exista. O membro com o cargo de líder tem a responsabilidade de replicar *logs* para os seguidores, adicionalmente, este regularmente informa os seus seguidores da sua existência através do envio de um *heartbeat*. Cada seguidor tem um ciclo de intervalos de tempo em qual espera receber um *heartbeat* do líder que é reiniciado sempre que o receba, no entanto, caso o intervalo de tempo termine sem o receber, então, o seguidor muda o seu cargo para candidato e começa uma eleição para um novo líder. Portanto, o protocolo *Raft* está dividido fundamentalmente em duas partes: eleição de líder e replicação de *logs*.

Quando o algoritmo inicializa ou um líder falha, um novo líder tem de ser eleito. Neste caso, é iniciado um novo termo no *cluster*. Um termo é um período arbitrário no *cluster* para o qual um novo líder precisa ser eleito, cada termo começa com a eleição de um novo líder. A eleição de um líder é iniciada por um membro candidato, este aumenta o contador de termo, vota em si mesmo como novo líder e envia uma mensagem para todos os outros membros a pedir o seu voto. Cada membro só pode votar uma vez por cada termo, e estes votam a favor do primeiro pedido de voto que receberam. Caso um candidato receba uma mensagem de outro membro com um contador de termo superior então este é automaticamente desqualificado e muda o seu cargo de volta para seguidor. Caso um membro receba a maioria de votos então este torna-se o novo líder, caso exista um empate de votos então um novo termo é começado e o processo é repetido, adicionalmente, de forma a evitar ciclos de empate de votos, cada membro escolhe um intervalo de tempo aleatório, com valores reduzidos, antes de voltar a tentar a nova eleição. Quanto à segunda parte, a replicação de *logs*, esta é a responsabilidade do líder, este recebe pedidos de clientes, sendo que cada pedido consiste num comando a ser executado e replicado por todos membros do *cluster*. Após o comando seja adicionado à lista de *logs* do líder, este envia este comando para todos os seguidores. Caso os seguidores não estejam disponíveis, o líder volta a tentar enviar o comando por vezes indefinidas até que o *log* seja eventualmente adicionado à lista dos seguidores. Assim que o líder recebe a confirmação, de metade ou mais dos seus seguidores, que o comando foi replicado, este aplica o comando ao seu estado local e o pedido é considerado como aplicado. Este protocolo é utilizado quando é necessário que exista uma forte consistência de informações no *cluster*, sendo permitido apenas ao líder realizar alterações, um exemplo comum de utilização deste protocolo pode ser encontrado nas bases de

dados *CockroachDB, MongoDB, Neo4j, TiDB* e *YugabyteDB*. Sendo que somente um membro do *cluster* é capaz de realizar alterações, a capacidade do *cluster* é limitada pela capacidade do líder. De forma a resolver este problema, é utilizado o *Multi-Raft*, este utiliza múltiplos grupos tendo cada um o seu líder e gerindo uma secção da informação. No caso de uma base de dados, podemos ter um grupo por cada tabela e aplicar alterações a grupos separados aumentando a quantidade de alterações possíveis e distribuindo a carga entre mais membros. Adicionalmente, caso somente seja necessário a consulta de informações, esta pode ser realizada a qualquer seguidor, com o risco de receber informação desatualizada ou então realizar a consulta ao líder para ter a garantia de ter a última informação.

C. Gossip

O protocolo *gossip* ou protocolo epidémico consiste em um procedimento de comunicação *peer-to-peer* que assimila a forma como as epidemias ou rumores se espalham, neste protocolo cada membro de grupo periodicamente troca informação com outros membros sobre o seu próprio estado e sobre o estado de outros membros. Este protocolo permite que um sistema distribuído tenha a garantia que a informação é eventualmente distribuída por todos os membros do grupo sem precisar de um sistema centralizado a coordenar esse aspeto. Visto não precisar de um sistema centralizado este protocolo é dos mais robustos e escaláveis para consistência eventual dos membros do *cluster*, deteção de falhas e permite o envio de informações adicionais durante as trocas de informação.

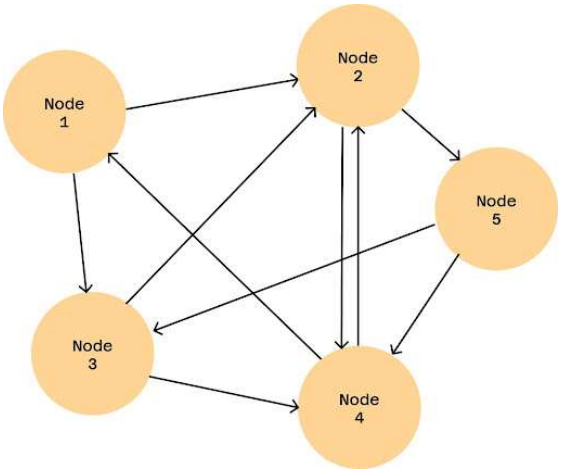


Figura 32 - Exemplo de cluster a utilizar o protocolo gossip

Na figura 32 podemos ver um exemplo de um *cluster* com 5 membros, neste exemplo cada membro comunica somente com outros 2 membros. De forma a propagar uma informação entre todos os membros seriam necessários 3 ciclos, sendo cada ciclo uma troca de informação entre os membros após cada intervalo de tempo definido no *cluster*. Uma exemplificação da propagação com origem no *Node 1* pode ser observada na figura 33.

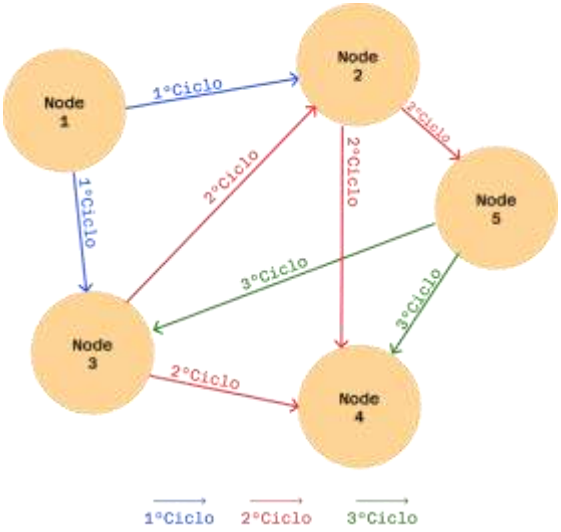


Figura 33 - Exemplo de propagação

No caso de um *cluster* com 40 membros e cada membro comunique somente com outros 4 membros seriam necessários somente 4 ciclos. O artigo "Epidemic

Algorithms for Replicated Database Maintenance" [19] descreve algoritmos de replicação de bases de dados que usam a propagação de informações entre membros de um sistema distribuído, este apresenta uma fórmula para estimar o tempo de convergência do algoritmo de propagação de informações com base no número de membros do sistema e na taxa de propagação de informações. Essa fórmula é T = O($log(N) / p$), onde N é o número de membros do sistema e p é a taxa de propagação de informações e O($log(N)$) o número de ciclos necessários para que a informação seja propagada por todo o sistema. Portanto, de forma a calcular aproximadamente quantos ciclos são necessários para a propagação de uma informação, iremos nos focar apenas na parte O(log(N)), desta forma, iremos usando o seguinte cálculo $C = log_P(N)$ onde c é o número de ciclos. Portanto, com 40 membros e propagação de 4 temos $log_4(40) = 2.66$, ou seja, aproximadamente 3 ciclos, no caso de um *cluster* com 5 membros e propagação de 2 temos $log_2(5) = 2.32$ que também são aproximadamente 3 ciclos.

Vendo o protocolo de alto nível, num *cluster*, cada membro mantém uma lista de um subconjunto dos membros a que tem conhecimento, os seus endereços e alguns dados adicionais (metadata), e periodicamente, cada membro atualiza na sua lista de "vizinhos" os contadores de *heartbeat* de acordo com os dados emitidos por outros membros e envia a informação atualizada para alguns dos membros. Assim que um membro tenha recebido uma das mensagens, esta junta a lista na mensagem com a sua lista e mantém os dados com o contador de *heartbeat* mais elevado no caso de colisões. Assim sendo, enquanto o valor do contador for subindo para um membro é garantido que este esteja *healthy* (ativo e sem problemas) e é considerado unhealthy (desativo ou com problemas) caso o contador de *heartbeat* não seja aumentado durante um intervalo de tempo. Adicionalmente, durante a troca de informações entre membros é possível enviar informações extra como por exemplo, carga média e memória livre para que outros membros possam utilizar essa informação para balancear a carga entre membros. Outra forma de explicar o protocolo *gossip* é comparando com a disseminação de rumores numa comunidade. Assim como no protocolo *gossip*, um rumor começa com uma pessoa que o compartilha com alguns amigos próximos. Esses amigos, por sua vez, compartilham o rumor com outros amigos, e assim sucessivamente. Conforme o rumor se espalha, este pode ser confirmado, negado ou até mesmo modificado por diferentes pessoas ao longo do caminho. O resultado é uma ampla disseminação de informações pela comunidade, com a possibilidade de chegar a um consenso ou opinião comum. Da mesma forma, o protocolo *gossip* permite a disseminação de informações em sistemas distribuídos, onde diferentes membros compartilham e modificam informações entre si até chegarem a um consenso ou estado comum. No caso deste projeto, o protocolo *gossip* é baseado em "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol" [20] com algumas modificações. A implementação foi criada pela empresa *Hashicorp* e foi nomeada de *Serf*. Explicando de forma breve e incompleta, um membro começa por se juntar a *cluster* já existente ou cria um novo, caso esteja-se a juntar, é realizada uma sincronização completa com um membro já existente do *cluster* utilizado o protocolo *TCP* e depois começa a realizar trocas de informação assim como referido previamente. Neste caso, a comunicação utilizada para troca de informações utiliza o protocolo *UDP* com o número de propagação de intervalo configurável. Nesta implementação é apenas enviadas alterações de informação com o protocolo *UDP*. Mesmo após um membro se juntar ao grupo algumas sincronizações completas ocorrem com outro membro aleatório

utilizando o protocolo *TCP*, no entanto, estas ocorrem com menor frequência, o intervalo destas transmissões também pode ser configurado ou desativado. De forma a detetar uma falha, um pedido de verificação é enviado aleatoriamente num intervalo de tempo configurável, caso o destinatário falhe a responder dentro de um prazo de tempo razoável então um pedido de verificação é enviado indiretamente. Um pedido de verificação indireto passa por pedir a um número configurável de membros para realizarem um pedido de verificação ao membro, isto permite perceber se um membro não está acessível por problemas que estejam a ocorrer na rede. Caso ambas tentativas falhem, então o membro é marcado como suspeito e estas informações são enviadas para todo *cluster* utilizado o mesmo mecanismo de propagação. Por fim, caso o membro suspeito não responda à suspeita num intervalo de tempo configurável então o membro é considerado como morto, e novamente esta informação é propagada pelo *cluster*. Outra funcionalidade desta implementação passa por permitir o envio de eventos e consultas utilizando o mecanismo de propagação, algo que pode ser utilizado, por exemplo, quando a configuração do *cluster* muda e é necessário que esta alteração seja propagada por todos os membros.

D. Escolha de protocolo de consenso

Tendo revisto as opções *gossip* e *Raft*, a opção escolhida para ser utilizada neste projeto passa pelo *gossip*. Tendo como objetivo que todos membros concordem com quais membros estão ativos, o algoritmo *Raft* oferece mais funcionalidades do que as necessárias e mais restrições do que a opção *gossip*, adicionalmente, não sendo necessário armazenar informação ou sendo exigido uma forte consistência de informação é preferível a utilização do protocolo *gossip* sendo este mais eficiente no consumo de recursos de processamento e de rede e permite uma quantidade mais elevada de membros sendo que o algoritmo *Raft* tem o seu melhor desempenho num *cluster* com 3 a 9 membros enquanto em *gossip* um número muito mais elevado é possível, por exemplo, em *gossip* um *cluster* com 100 membros e propagação de 4 leva aproximadamente 3 ciclos a propagar a informação.

A utilização de ambos protocolos em simultâneo também é possível, utilizando o protocolo *gossip* de forma a manter uma lista de membros ativos, e utilizar o algoritmo *Raft* apenas para gerir a consistência de informação, no entanto, como previamente mencionado, a utilização do *Raft* limite consideravelmente o número de membros a serem utilizados num *cluster*.

Utilizando o *gossip*, é possível manter uma consistência eventual dos membros presentes no *cluster*, e esta informação é somente utilizada de forma a realizar intercomunicação entre os membros do *cluster*. Não tendo o protocolo *gossip* como objetivo de enviar informação de forma rápida, será antes utilizada a informação que este gere para utilizar outro método de envio de informação para o resto dos dados aplicacionais, adicionalmente, também é necessário organizar os membros de forma a evitar e reduzir o número de vezes que uma mensagem tem de ser transmitida.

E. Intercomunicação

De forma a realizar a intercomunicação entre membros existem várias possibilidades, no entanto, as mais utilizadas são *Apache Thrift*, *gRPC* ou então usar diretamente uma conexão *TCP* e gerir diretamente o envio de dados. De forma a simplificar e reutilizar conhecimento já existente na empresa, o método de comunicação escolhido é o *gRPC*.

O *gRPC* é um *framework* de comunicação remota de alta performance, este permite que aplicativos clientes e servidores troquem dados entre si de maneira rápida, confiável e eficiente, utilizando protocolos de comunicação padronizados e uma interface de programação simples e fácil de utilizar. O *gRPC* é baseado no protocolo *HTTP/2*, o que significa que este suporta funcionalidades avançadas, como *streaming* bidirecional e unidirecional, compressão de dados e multiplexação de pedidos. Este é frequentemente utilizado em sistemas distribuídos e em arquiteturas baseadas em microserviços para facilitar a comunicação entre diferentes componentes do sistema, outras funcionalidades deste *framework*. De forma a serializar os dados enviados, o *gRPC* utiliza *Protocol Buffers*. O *Protocol Buffers* é uma tecnologia de serialização de dados também desenvolvida pela *Google*, que permite que estruturas de dados sejam definidas em um formato de linguagem neutra e compacta. Estas estruturas são então compiladas em código fonte para várias linguagens de programação, o que permite que as aplicações cliente e servidor possam facilmente trocar dados estruturados entre si.

F. Distribuição

Tendo uma forma de saber quais membros estão presentes no *cluster* e forma de comunicação entre cada membro, é necessário estabelecer a forma como estes serão organizados. De forma a aumentar e evitar problemas de desempenho, o *cluster* não terá nenhum membro central que terá toda a responsabilidade ou que irá atribuir responsabilidades, em vez disso, cada membro vai ser responsável por um conjunto da carga a ser processada, e a designação de qual membro tem qual responsabilidade vai ser definida através do hash ring. Sendo um *channel* a parte onde irá ocorrer quase todo o processamento da aplicação, o nome deste em conjunto o nome do *hub* vão ser utilizados como chave para distribuição.

G. Hash Ring

Portanto, de forma a evitar este problema, temos a técnica de anel de *hash*, ou *hash ring*, esta técnica forma um anel virtual (figura 34), em que cada membro é responsável por um intervalo contínuo de valores no anel.

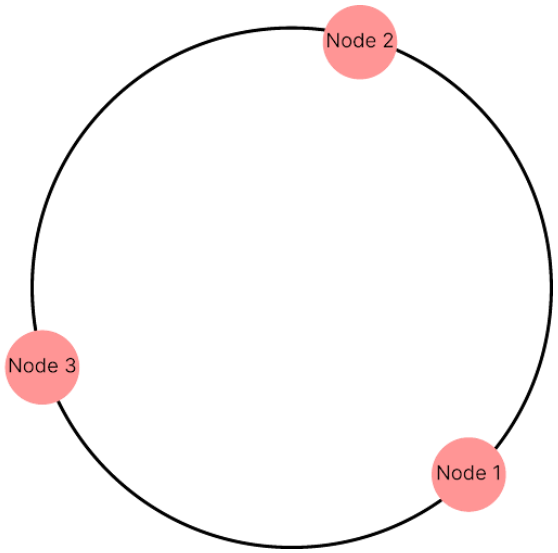


Figura 34 - Membros do cluster representados num anel virtual

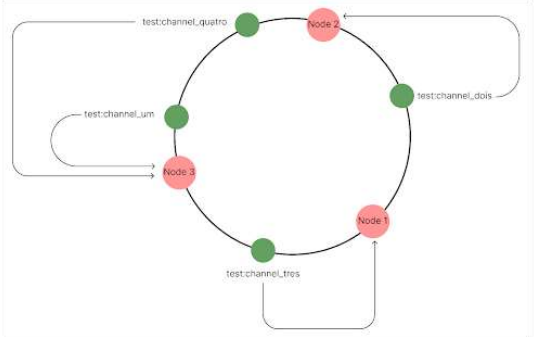


Figura 35 - Anel virtual com membros de um cluster e channels

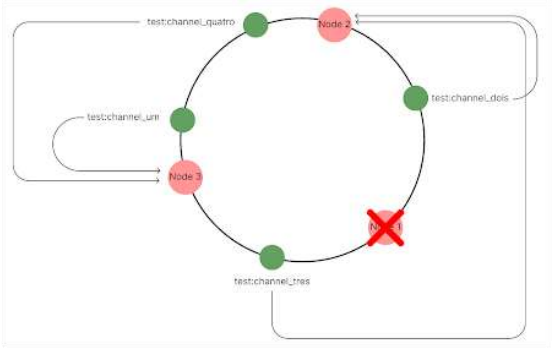


Figura 36 - Anel virtual com membros de um cluster e channels com a falha de um membro

Exemplificando os cenários anteriores podemos ver na figura 35 como a distribuição dos *channels* é representada no *hash ring*. Portanto, tendo em conta o mesmo exemplo, podemos ver na figura 36 o resultado do mesmo cenário de falha do membro "Node 1". Como pode ser visto, somente um *channel* precisa de ser rebalanceado, além de serem precisos menos rebalanceamentos, também podemos somente recalcular os *channels* a que pertenciam aquele membro, tornando esta técnica ainda mais eficiente. Imaginando a situação em que um novo membro se junta ao *cluster* com o nome "Node 4" e a sua posição no anel é calculada entre o "Node 3" e "Node 2" podemos buscar todos *channels* a que o "Node 2" é responsável e recalcular o seu responsável, mais uma vez evitando recalcular todos os *channels*. Usando esta mesma técnica, temos a possibilidade de saber quem poderá ser o próximo responsável de um certo *channel*, algo que pode ser utilizado de forma criar um sistema de redundância.

H. Novo Sistema

Tendo agora as partes fundamentais do sistema, com o consenso a ser resolvido com o protocolo gossip, a intercomunicação com o *framework gRPC* e a distribuição utilizando a junção do gossip e hash ring, é importante perceber como estes irão funcionar em conjunto.

Em primeiro lugar, temos a parte responsável por chegar ao consenso de quantos membros existem no *cluster*, esta parte assim como previamente referida é gerida pelo protocolo gossip. Portanto, sempre que um novo membro se junta ou sai do *cluster*, este irá refletir no hash ring. Lembrando, que no *hash ring* vão ser mapeados todos os identificadores dos membros do *cluster*. Assim sendo, quando precisamos de distribuir um *channel* ou localizá-lo, será calculada a localização do *channel* no *hash ring* utilizando o identificador deste. Sabendo a posição no *hash ring*, podemos facilmente calcular a qual membro o *channel* pertence. Portanto, quando um membro sai ou se junta, a sua posição será adicionada ou removida do *hash ring* e potencialmente será necessário recalcular a quais membros os *channels* pertencem.

O ponto de intercomunicação, neste sistema é introduzido quando é necessário enviar informação entre membros, por exemplo, publicar um evento num *channel*, exige que um pedido seja feito ao membro responsável por este, ou seja, uma conexão será criada ou reutilizada ao membro destino onde será enviado o pedido para publicar o evento. Assim como mencionado, esta intercomunicação será realizada com o *framework gRPC*, de forma a saber os endereços para qual a conexão será criada, será utilizado o protocolo *gossip* para descobrir esses endereços.
Portanto, o protocolo *gossip* é o elemento principal destes três pontos, este em junção com os outros dois pontos permite ter bases para a criação de um sistema distribuído. Existe ainda um ponto.

I. Inicializar

Para iniciar um *cluster* é necessário a existência de mais que uma instância da aplicação. A uma destas instâncias é indicado os endereços de rede da outra, para que uma conexão seja estabelecida. Assim que estabelecida, ambas instâncias formam

um *cluster* de acordo com o protocolo *gossip,* e novas instâncias têm de se juntar ao *cluster* utilizando o mesmo processo.
Para este processo, é necessário o conhecimento dos endereços de rede das novas instâncias da aplicação, algo que costuma ser gerido por um *service discovery*, ou descoberta de serviços. Este é um serviço utilizado em arquiteturas de sistemas distribuídos para encontrar e se conectar a serviços disponíveis numa rede, este tem um endereço de rede conhecido por todas aplicações que o usam para registarem a sua presença e publicarem informações sobre si, tornando mais fácil para outros serviços localizá-los e se comunicarem com eles. Isto permite que os sistemas distribuídos sejam mais flexíveis, escaláveis e resilientes, uma vez que os serviços podem ser facilmente adicionados ou removidos sem afetar a operação geral do sistema.
Na empresa não existe um *service discovery*, visto que o *NATS* serve como um serviço central que permite a comunicação entre serviços, evitando assim a necessidade de um *service discovery*. Portanto, para esta aplicação um *service discovery* seria útil, mas visto a inexistência de um, foram utilizados meios mais simples de forma a descobrir outras instâncias desta aplicação.
O serviço *AWS ECS*, é onde a aplicação vai ser executada, utilizando *Docker containers*. Este serviço oferece uma *API*, que permite que sejam consultadas informações sobre as instâncias em execução. Portanto, quando uma nova instância é criada, esta utiliza esta *API*, para consultar todas as placas de rede do mesmo tipo da aplicação, retornando assim os endereços de rede a que estas é atribuído, com estes endereços a aplicação realiza um pedido para se juntar ao *cluster*, que será aceite caso as credenciais da nova instância estejam corretas. Para ambientes locais de desenvolvimento, é utilizado o *UDP Broadcast*. Sendo isto uma técnica de comunicação em rede que envia mensagens de um emissor para vários dispositivos, sem que o emissor precise saber exatamente quem são esses dispositivos ou onde estes estão localizados na rede. Nesse método, o emissor envia uma mensagem de difusão (*Broadcast*) para um endereço *IP* especial, que é reconhecido por todos os dispositivos conectados na rede. Assim, todos os dispositivos conectados na rede que estão à escuta nesse endereço IP especial, podem receber a mensagem enviada pelo emissor. Quando os outros membros recebem a mensagem enviada, estes podem anunciar sua presença na rede e permitir que outros membros os descubram de maneira fácil e rápida.

IV. Resultados

Assim como previamente mencionado, antes de expor o novo sistema às aplicações móveis, foi feito um teste de quantas conexões um *cluster* com 3 membros seria capaz de suportar, em servidores com somente 0.25 *vCPU* e 0.5 *GB* de *RAM*. Neste teste, cada membro do *cluster* foi capaz de suportar aproximadamente 15 mil conexões num total de aproximadamente 45 mil conexões, não sendo capaz de ter mais conexões devido ao limite de *RAM* nos servidores. Utilizando servidores com maior capacidade o *cluster* é capaz de aumentar a quantidade de conexões com o mesmo número de membros, no entanto, o objetivo era ver a capacidade de distribuição e tolerância a falhas. Estes valores obtidos são muito superiores aos 9000 atualmente esperados, além de ser capaz de suportar falha dos membros sem indisponibilizar o serviço.
Após o teste de conexões foi realizado outro teste com o objetivo de avaliar o tempo que um *cluster* demora a ajustar os *channels* pelos membros. Neste teste existem 3 membros no *cluster* e um quarto é adicionado normalmente e removido repentinamente de forma a simular uma falha. Após cada ajuste é coletado a partir dos logs de cada membro o tempo que o ajuste demorou em milissegundos, para clarificar, só é apontado a duração do ajuste

e não de deteção que o membro foi adicionado ou removido.
Para este teste, foram escolhidas 4 variações com 5 rondas cada, nestas variam o número de *channels* e o número de clientes, no entanto estes não ultrapassam dos 3. As 4 variações são as seguintes:

1ª Variação:
- 1 cliente conectado ao *Node 1*;
- 500 *channels*;
- Identificador de *channels* em *UUIDs* (*Universal Unique IDentifiers*).

2ª Variação:
- 3 clientes, um conectado a cada *Node*.
- 1500 *channels*;
- Identificador de *channels* em conjunto de 20 caracteres aleatórios.

3ª Variação:
- 3 clientes, um conectado a cada *Node*.
- 4500 *channels*;
- Identificador de *channels* em conjunto de 40 caracteres aleatórios.

4ª Variação:
- 50 clientes distribuídos pelos *Nodes*.
- 100000 *channels*;
- Identificador de *channels* em conjunto de 40 caracteres aleatórios.

A diferença nos identificadores do *channels* deve-se ao posicionamento no hash ring. Na primeira variação grande parte dos identificadores *channels* eram semelhantes, o que levava que estes tivessem o mesmo Node como responsável. Com os identificadores gerados aleatoriamente houve uma melhor distribuição pelos Nodes, assim como pode ser observado nos resultados representados na figura 37 e figura 38.

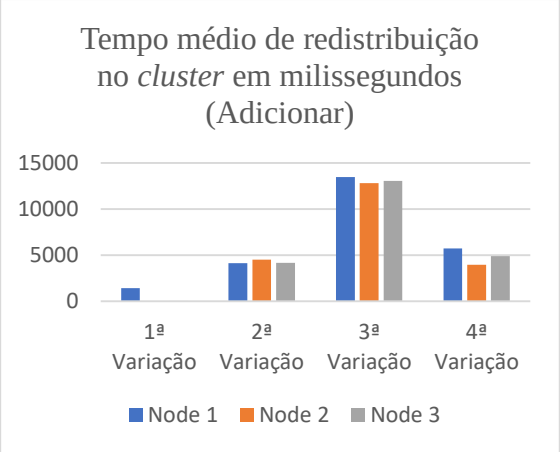


Figura 37 - Tempo médio de redistribuição no cluster em milissegundos (Adicionar)

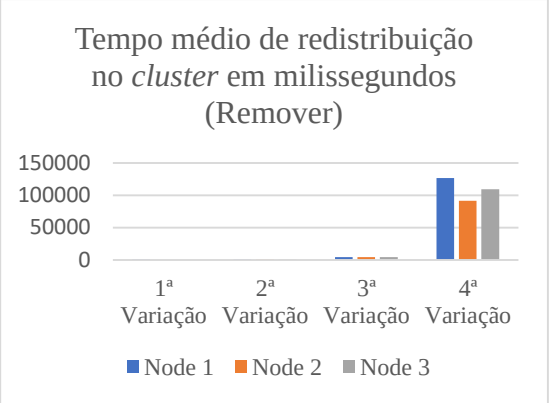


Figura 38 - Tempo médio de redistribuição no cluster em milissegundos (Remover)

A diferença entre de desempenho entre o adicionar ou remover um membro do *cluster* deve-se ao processo de remover um membro do *cluster* ser mais eficiente. Neste processo sabemos o identificador do *Node* que saiu o que permite que seja calculado de forma eficiente quais *channels* devem ser movidos, adicionalmente, numa redistribuição ao adicionar um membro é necessário notificar o membro do *cluster* previamente responsável de que não tem mais interesse nos *channels* a que estes pertenciam, este passo não acontece quando um membro é removido. O tempo de ajuste destas variações vai subindo de acordo com o número de *channels* ativos no *cluster*, sendo que quando observamos o salto da segunda para a terceira variação do teste o tempo de ajuste sobe consideravelmente, inclusive não foi possível realizar a quarta

variação do teste devido ao tempo que o *cluster* fica em ajustes. Após analisar os o que leva a este considerável aumento, foi identificado que o atraso se deve à forma como a movimentação dos *channels* e das conexões *gRPC* para os novos responsáveis é realizada. Este processo é feito de forma individual, ou seja, um *channel* de cada vez. Este processo foi melhorado drasticamente, agrupando todos as alterações a serem realizadas por membro num conjunto e enviar somente uma mensagem por cada conjunto de operações, adicionalmente, este processo foi também paralelizado. Esta alteração resultou nos resultados apresentados da quarta variação.

Quanto ao tempo que leva a adicionar ou remover um *Node* ao *cluster* é de aproximadamente 16ms, para a deteção de falha do tempo é entre 120 a 500ms. De forma a obter os 16ms, foi registado o tempo em que o *Node* descobre os endereços dos outros membros e o tempo em que este se juntou a pelo menos um dos membros do *cluster*. Para o tempo de deteção de falha foi comparado o tempo em que o *Node* foi terminado com o tempo que um dos *Nodes* detetou a falha, os intervalos entres estes dois tempos foram muito variados sendo os valores mais comuns entre 120 a 500ms.

Quanto às fases de teste com tráfego real, não foi possível obter todos os resultados antes da produção deste documento. Visto que o projeto ainda se encontra na segunda fase de testes, as análises de resultados são bastante limitadas.

Nas primeiras duas fases de testes com tráfego real, foi observado que o número de sessões em simultâneo era consideravelmente inferior ao esperado, no entanto, é possível verificar que o número de sessões é distribuído ao longo do dia, com picos relativamente pequenos.

Na figura 39 podemos observar a distribuição de sessões ao longo do dia, esta visualização acumula as sessões dos tenants em fase de teste.

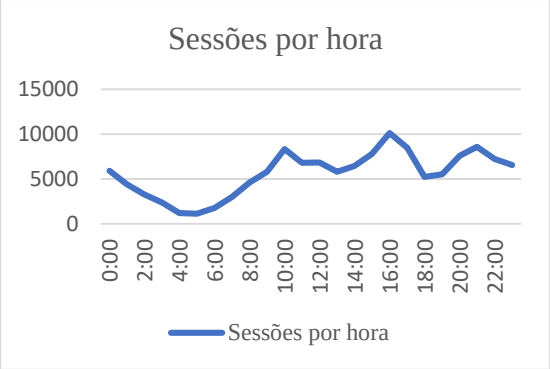


Figura 39 - Distribuição de sessões por hora

Na figura 40, temos a representação da duração média de sessão a cada hora do dia. Assim como pode ser observado as durações das sessões são relativamente baixas, sendo assim com o apresentado as 16 horas na mesma figura.

Figura 40 - Média de duração de sessão por hora

Até ao momento de que este documento foi desenvolvido, já foram em enviadas 12 706 780 mensagens e recebidas 12 706 690 mensagens, com um total de 173 977 sessões estabelecidas com 2802 sessões diárias em média, e com soma de duração destas de 23 757 458 segundos ou aproximadamente 6600 horas e duração média de 2 minutos.

V. Conclusão

Em conclusão, o novo sistema distribuído apresenta uma solução satisfatória para os problemas encontrados no sistema antigo, com melhorias significativas na escalabilidade, na tolerância a falhas e na monitorização. Adicionalmente, todas as funcionalidades do antigo sistema foram

mantidas enquanto novas foram adicionadas, permitindo também adicionar futuras funcionalidades graças à sua arquitetura. Embora algumas funcionalidades, como *Streams* e *push notifications*, ainda não estejam completas, foram identificadas oportunidades para melhorias futuras.

O novo sistema atingiu todos os objetivos estabelecidos, incluindo escalabilidade horizontal, comunicação bidirecional entre cliente e servidor, comunicação utilizando *Pub/Sub* em tópicos, restrição de acesso a tópicos, suporte para múltiplos *tenants*, criação explícita de tópicos, rastreamento de presença de clientes em cada tópico e armazenamento de mensagens enviadas em cada tópico.
Os próximos passos para o sistema são planear o escalamento global e o *routing* inteligente, tendo em conta a latência entre o servidor e o cliente, garantindo que o sistema possa lidar comum número maior de utilizadores e volume maior de dados.

VI. VI. Referências

[1] Fette, I., & Melnikov, A. (2011). *The WebSocket Protocol. RFC*. https://doi.org/10.17487/rfc6455https://datatracker.ietf.org/doc/html/rfc6455

[2] Roome, W., & Yang, Y. R. (2020, novembro 1). *Application-Layer Traffic Optimization (ALTO) Incremental Updates Using Server-Sent Events (SSE)*. IETF. https://www.rfc-editor.org/rfc/rfc8895.html

[3] Thomson, M., & Damaggio, E. (2016, dezembro 1). *Generic Event Delivery Using HTTP Push*. https://www.rfc-editor.org/rfc/rfc8030

[4] Centrifugo. (s.d.). *Scalable real-time messaging server in a language-agnostic way*. Recuperado a 13 de janeiro de 2023, de https://centrifugal.dev/

[5] Mercure. (s.d.). Recuperado a 13 de janeiro de 2023, de https://mercure.rocks/

[6] Phoenix Framework. (s.d.). Recuperado a 13 de janeiro de 2023, de https://www.phoenixframework.org/

[7] Octavo Labs AG. (s.d.). *A MQTT broker that is scalable, enterprise ready, and open source*. VerneMQ. Recuperado a 13 de janeiro de 2023, de https://vernemq.com/

[8] HiveMQ (s.d.). *Enterprise ready MQTT to move your IoT data*. HiveMQ. Recuperado a 13 de janeiro de 2023, de https://www.hivemq.com/

[9] Emitter (s.d.) *Scalable Real-Time Communication Across Devices*. Emitter.io. Recuperado a 13 de janeiro de 2023, de https://emitter.io/

[10] EMQ Technologies Inc. (s.d.). *EMQ X - MQTT Messaging Broker for IoT*. EMQ. Recuperado a 13 de janeiro de 2023, de https://www.emqx.io/.

[11] SocketCluster. (s.d.) *SocketCluster - Highly scalable pub/sub and RPC toolkit optimized for async/await*. Socketcluster.io. Recuperado em 13 de janeiro de 2023, de https://socketcluster.io/

[12] Soketi. (s.d.). Recuperado a 13 de janeiro de 2023, de https://soketi.app/

[13] Microsoft. (s.d.). *Real-time ASP.NET with SignalR | .NET*. Recuperado em 20 de junho de 2023, de https://dotnet.microsoft.com/apps/aspnet/signalr

[14] Ably. (s.d.). *The platform to power synchronized digital experiences in realtime*. Ably Realtime. Recuperado a 13 de janeiro de 2023, de https://ably.com/

[15] PubNub Inc. (2022, julho 14). *Real-time in-app chat and Communication Platform*. PubNub. Recuperado a 13 de janeiro de 2023, de https://www.pubnub.com/

[16] Pusher Ltd. (s.d.). *Powering realtime experiences for mobile and web, Leader in Realtime Technologies*. Pusher Recuperado a 13 de janeiro de 2023, de https://pusher.com/

[17] Fanout. (s.d.). *Fanout | Powering Streaming APIs*. Fanout Recuperado a

13 de janeiro de 2023, de https://fanout.io/

[18] The Go Programming Language. (s.d.). Golang.org. Recuperado a 13 de janeiro de 2023, de https://golang.org/

[19] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., & Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC '87) (pp. 1-12). Association for Computing Machinery. https://doi.org/10.1145/41840.41841

[20] Das, I., Gupta, I., & Motivala, A. (2002). SWIM: scalable weakly-consistent infection-style process group membership protocol. In Proceedings International Conference on Dependable Systems and Networks (pp. 303-312). Washington, DC, USA. doi: 10.1109/DSN.2002.1028914.

BIBLIOGRAFIA

Redis Labs. (s.d.). Redis. Recuperado a 13 de janeiro de 2023 de https://redis.io/

Fette, I., & Melnikov, A. (2011). *The WebSocket Protocol. RFC.* https://doi.org/10.17487/rfc6455

Roome, W., & Yang, Y. R. (2020, novembro 1). *Application-Layer Traffic Optimization (ALTO) Incremental Updates Using Server-Sent Events (SSE)*. IETF. https://www.rfc-editor.org/rfc/rfc8895.html

Thomson, M., & Damaggio, E. (2016, dezembro 1). *Generic Event Delivery Using HTTP Push*. https://www.rfc-editor.org/rfc/rfc8030

Centrifugo. (s.d.). *Scalable real-time messaging server in a language-agnostic way*. Recuperado a 13 de janeiro de 2023, de https://centrifugal.dev/

Mercure. (s.d.). Recuperado a 13 de janeiro de 2023, de https://mercure.rocks/

Phoenix Framework. (s.d.). Recuperado a 13 de janeiro de 2023, de https://www.phoenixframework.org/

Octavo Labs AG. (s.d.). *A MQTT broker that is scalable, enterprise ready, and open source*. VerneMQ. Recuperado a 13 de janeiro de 2023, de https://vernemq.com/

HiveMQ (s.d.) *Enterprise ready MQTT to move your IoT data*. HiveMQ. Recuperado a 13 de janeiro de 2023, de https://www.hivemq.com/

Emitter (s.d.) *Scalable Real-Time Communication Across Devices*. Emitter.io. Recuperado a 13 de janeiro de 2023, de https://emitter.io/

EMQ Technologies Inc. (s.d.). *EMQ X - MQTT Messaging Broker for IoT*. EMQ. Recuperado a 13 de janeiro de 2023, de https://www.emqx.io/.

SocketCluster. (s.d.) *SocketCluster - Highly scalable pub/sub and RPC toolkit optimized for async/await*. Socketcluster.io. Recuperado em 13 de janeiro de 2023, de https://socketcluster.io/

Soketi. (s.d.). Recuperado a 13 de janeiro de 2023, de https://soketi.app/

Microsoft. (s.d.). *Real-time ASP.NET with SignalR | .NET*. Recuperado em 20 de junho de 2023, de https://dotnet.microsoft.com/apps/aspnet/signalr

Ably. (s.d.). *The platform to power synchronized digital experiences in realtime. Ably Realtime*. Recuperado a 13 de janeiro de 2023, de https://ably.com/

PubNub Inc. (2022, julho 14). *Real-time in-app chat and Communication Platform. PubNub*. Recuperado a 13 de janeiro de 2023, de https://www.pubnub.com/

Pusher Ltd. (s.d.). *Powering realtime experiences for mobile and web. Leader in Realtime Technologies*. Pusher. Recuperado a 13 de janeiro de 2023, de https://pusher.com/

Fanout. (s.d.). *Fanout | Powering Streaming APIs*. Fanout Recuperado a 13 de janeiro de 2023, de https://fanout.io/

The Go Programming Language. (s.d.). Golang.org. Recuperado a 13 de janeiro de 2023, de https://golang.org/

NATS.io. (s.d.). Recuperado a 13 de janeiro de 2023, de https://nats.io/

Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., & Terry, D. (1987). *Epidemic algorithms for replicated database maintenance*. In Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC '87) (pp. 1-12). Association for Computing Machinery. https://doi.org/10.1145/41840.41841

Das, I., Gupta, I., & Motivala, A. (2002). *SWIM: scalable weakly-consistent infection-style process group membership protocol*. In Proceedings International Conference on Dependable Systems and Networks (pp. 303-312). Washington, DC, USA. doi: 10.1109/DSN.2002.1028914.

Google Developers. (2019). *Protocol Buffers*. Recuperado a 13 de janeiro de 2023, de https://developers.google.com/protocol-buffers

Bryan, P. C., & Nottingham, M. (2013, abril 1). *JavaScript Object Notation (JSON) Patch*. IETF. https://datatracker.ietf.org/doc/html/rfc6902

APÊNDICE A

gRPC

O *gRPC* é um *framework* de comunicação remota de alta performance desenvolvido pela *Google*. Este permite que aplicativos clientes e servidores troquem dados entre si de maneira rápida, confiável e eficiente, utilizando protocolos de comunicação padronizados e uma interface de programação simples e fácil de utilizar. O *gRPC* é baseado no protocolo *HTTP/2*, o que significa que este suporta funcionalidades avançadas, como *streaming* bidirecional e unidirecional, compressão de dados e multiplexação de chamadas. Este é frequentemente utilizado em sistemas distribuídos e em arquiteturas baseadas em microserviços para facilitar a comunicação entre diferentes componentes do sistema. Utilizando as ferramentas existentes no *framework* é possível gerar implementações de servidor e cliente para várias linguagens de programação, reduzindo o tempo de desenvolvimento e facilitando a manutenção dos servidores e clientes.

APÊNDICE B

AWS ECS

O *Amazon Elastic Container Service* (*ECS*) consiste num serviço de gestão de containers fornecido pela *Amazon Web Services* (*AWS*) que permite aos utilizadores executar e dimensionar aplicativos em containers. O *ECS* é integrado com outros serviços da *AWS*, como o *Amazon Elastic Compute Cloud* (*EC2*) e o *Amazon Elastic Load Balancing* (*ELB*), para permitir a gestão de containers em grande escala e distribuir o tráfego entre estes. Portanto, as principais funcionalidades deste serviço são:

- Gestão de *Docker containers*;
- Monitorização de *containers*;
- Integração com outras serviços *AWS*;
- Escalamento automático;
- Distribuição de carga;
- Flexibilidade de implementação;
- Inclui a mesma segurança que a *AWS* oferece nos seus serviços como *AWS Identity, Access Management* (*IAM*) e *Amazon Virtual Private Cloud* (*VPC*).

APÊNDICE C

AWS Cloudwatch

O *Amazon CloudWatch* é um serviço de monitorização e análise de *logs* fornecido pela *Amazon Web Services* (*AWS*) que permite monitorizar recursos e aplicações em tempo real. Com o *CloudWatch*, os utilizadores podem coletar e rastrear métricas, coletar e monitorar *logs*, definir alarmes e automatizar ações com base em eventos. Este serviço é amplamente utilizado de forma a monitorar a saúde de aplicativos, identificar e resolver problemas de desempenho e otimizar o uso de recursos na *AWS*.

APÊNDICE D

AWS X-Ray

O *AWS X-Ray* é um serviço de rastreamento de transações fornecido pela *Amazon Web Services* (*AWS*) que permite aos utilizadores analisar e depurar aplicativos distribuídos. Utilizando o *X-Ray*, os utilizadores podem rastrear o fluxo de solicitações através de seus aplicativos e identificar pontos de baixo desempenho ou falhas de desempenho em sistemas complexos. Este serviço fornece uma visão abrangente do desempenho dos aplicativos, permitindo que os utilizadores identifiquem e resolvam problemas de desempenho. De forma coletar as transações é utilizado o *OpenTelemetry*. Este é um projeto de código aberto que tem como objetivo fornecer uma maneira padrão e flexível de instrumentar aplicativos para coletar dados de telemetria, como rastreamento, métricas e *logs*.

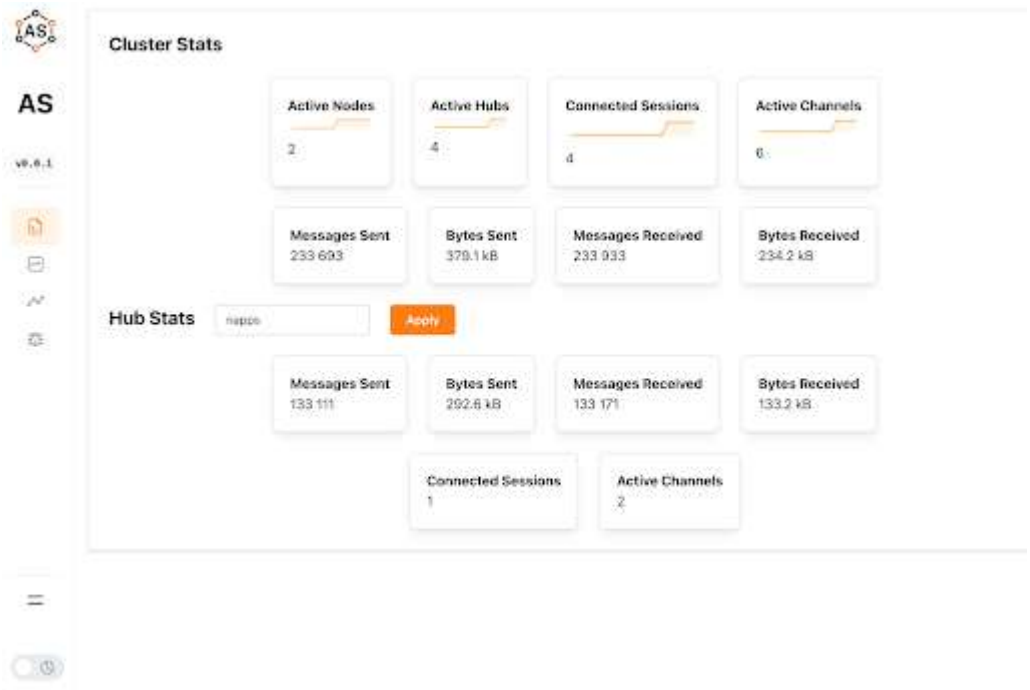
APÊNDICE E

Typescript

TypeScript é uma linguagem de programação de código aberto desenvolvida pela *Microsoft* que adiciona recursos opcionais de tipagem estática ao *JavaScript*. Este é projetado para ser um *superset* do *JavaScript*, o que significa que todo o código *JavaScript* é válido em *TypeScript* e os seus utilizadores podem gradualmente adicionar recursos de tipagem estática para obter mais segurança e facilidade de manutenção nos seus projetos.

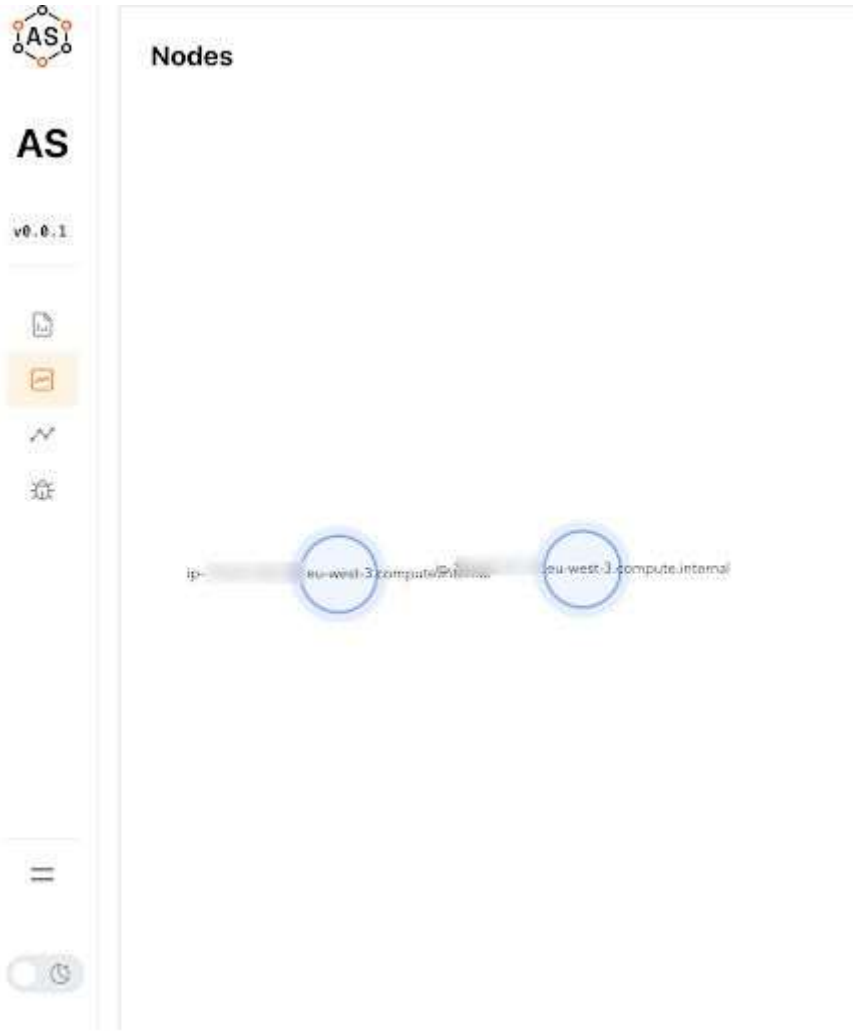
APÊNDICE F

Figura 41- Página inicial do dashboard



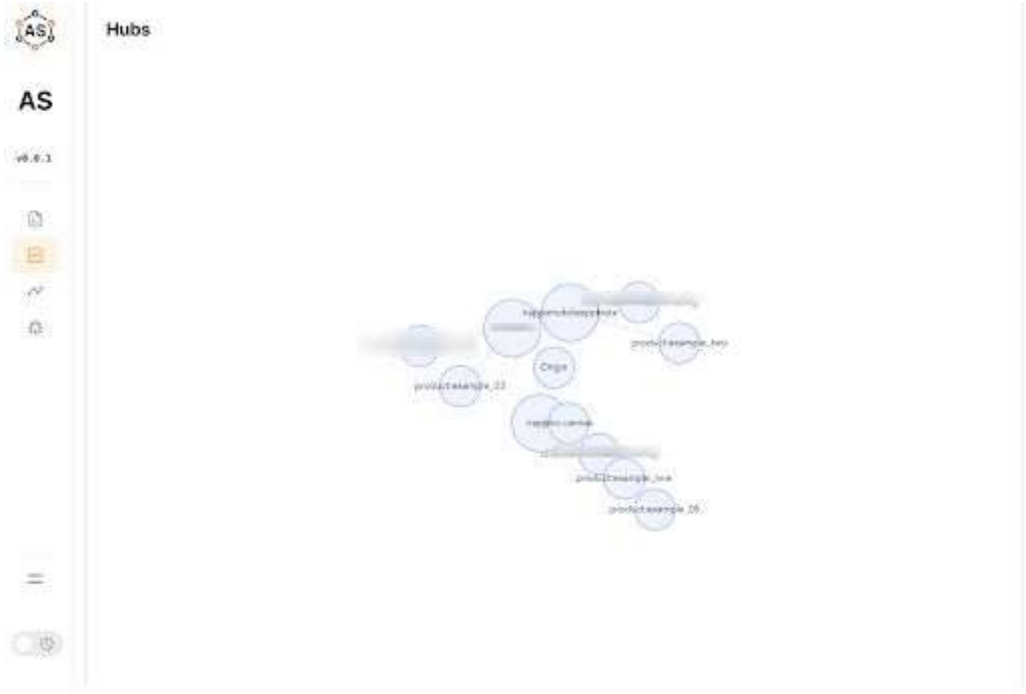
Fonte: Própria

Figura 42- Página de topografia do dashboard, parte 1



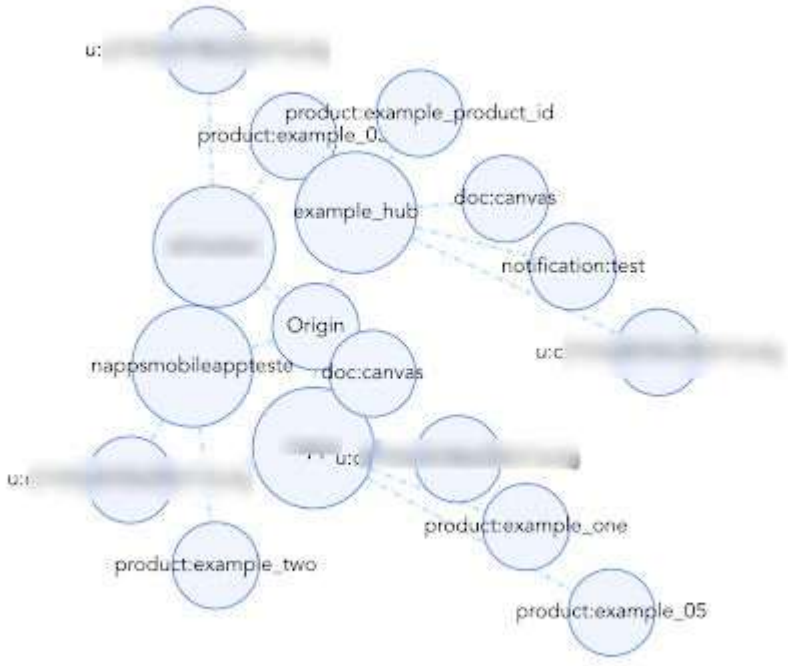
Fonte: Própria

Figura 43- Página de topografia do dashboard, parte 2



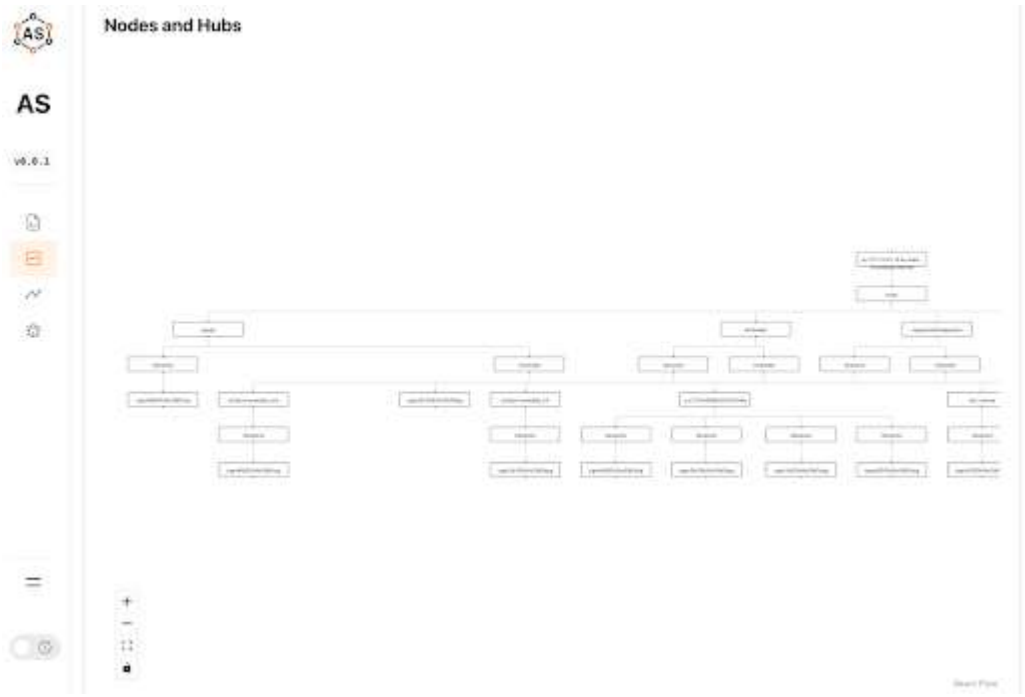
Fonte: Própria

Figura 44- Página de topografia parte 2 ampliada



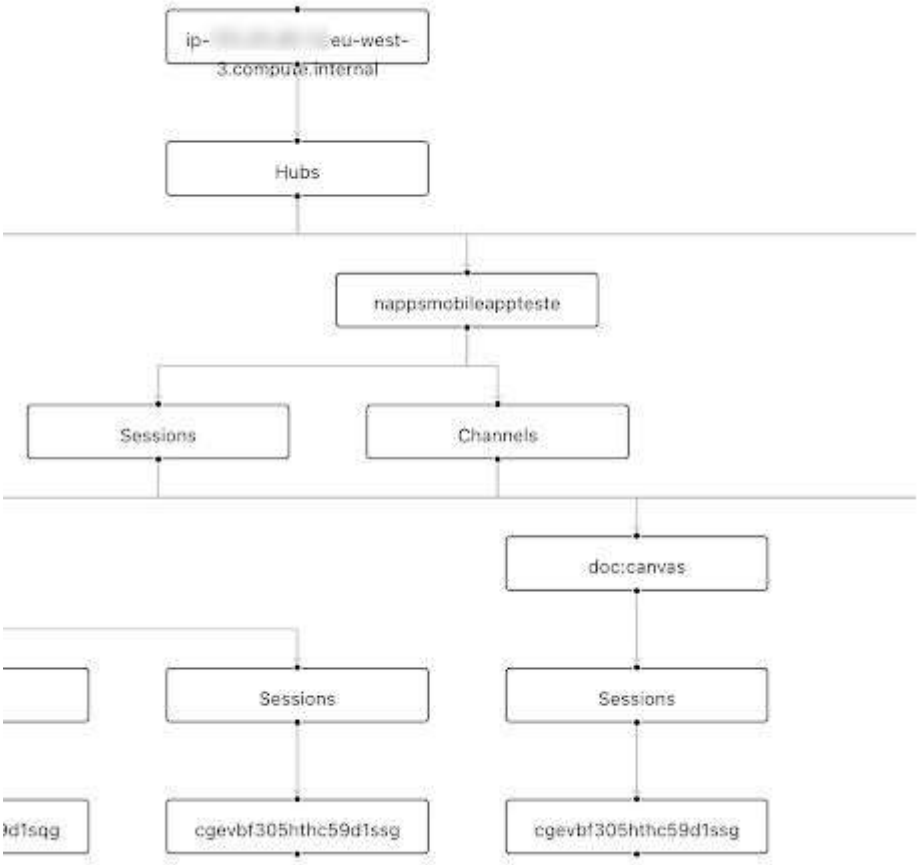
Fonte: Própria

Figura 45- Página de topografia do dashboard, parte 3



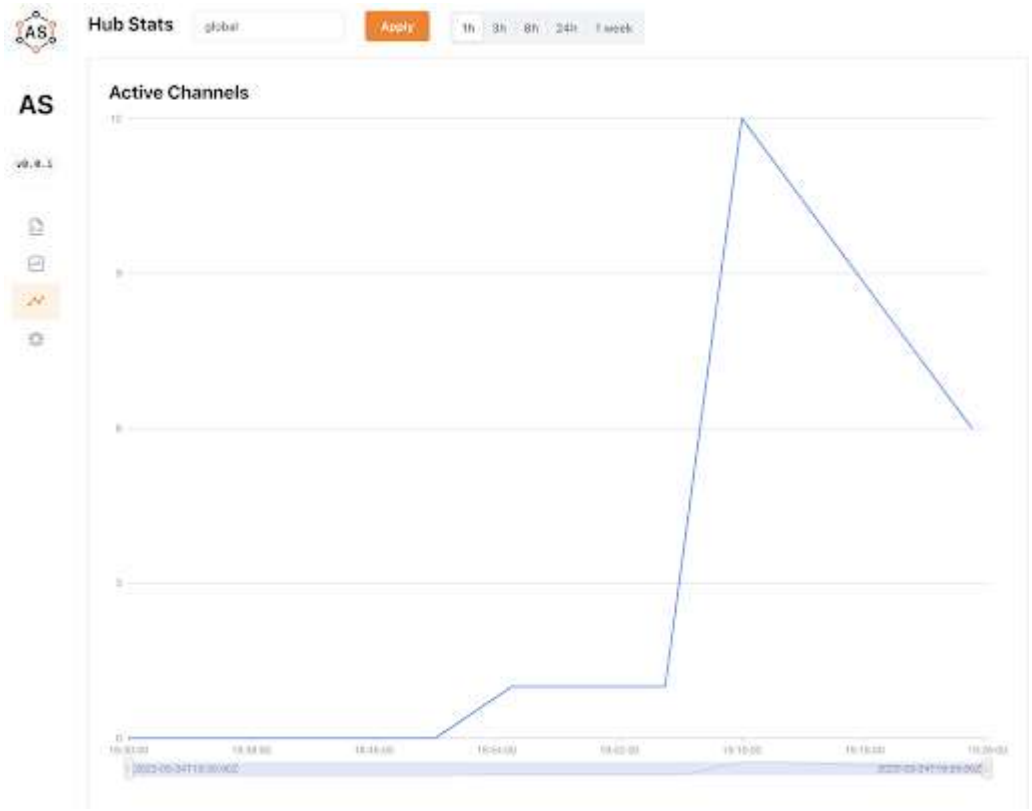
Fonte: Própria

Figura 46 - Página de topografia parte 3 ampliada



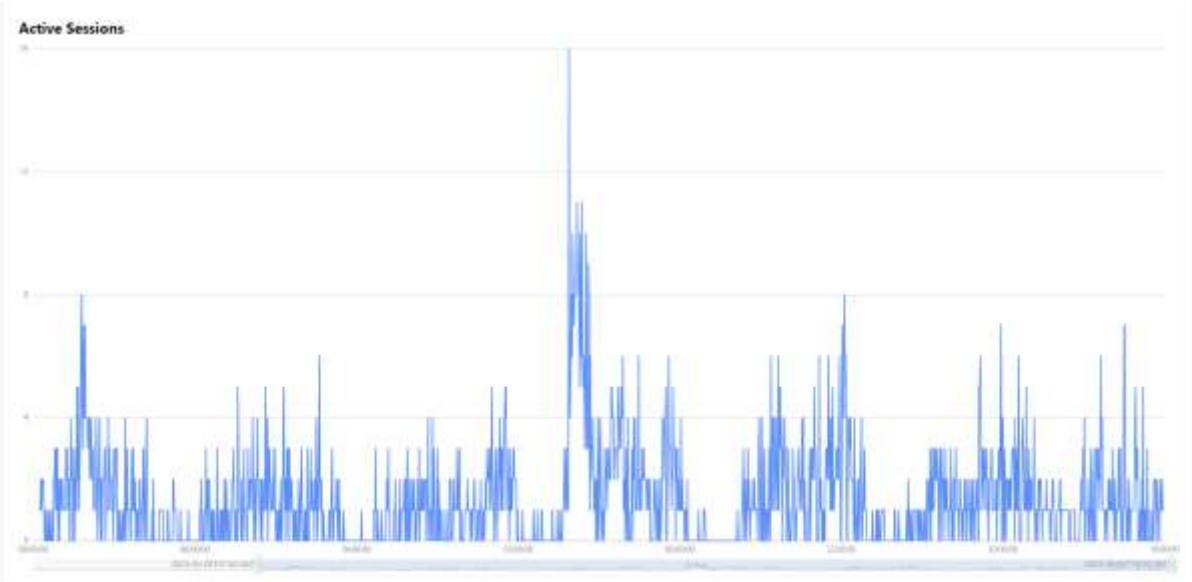
Fonte: Própria

Figura 47- Página de métricas do dashboard, channels ativos



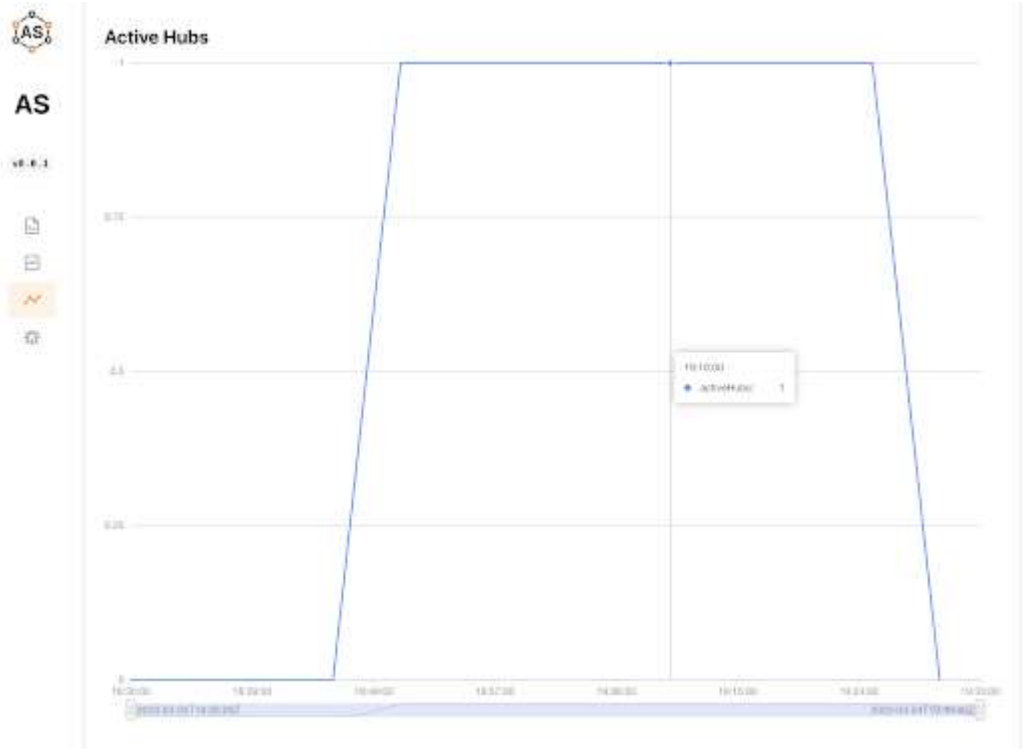
Fonte: Própria

Figura 48- Página de métricas do dashboard, sessões ativas



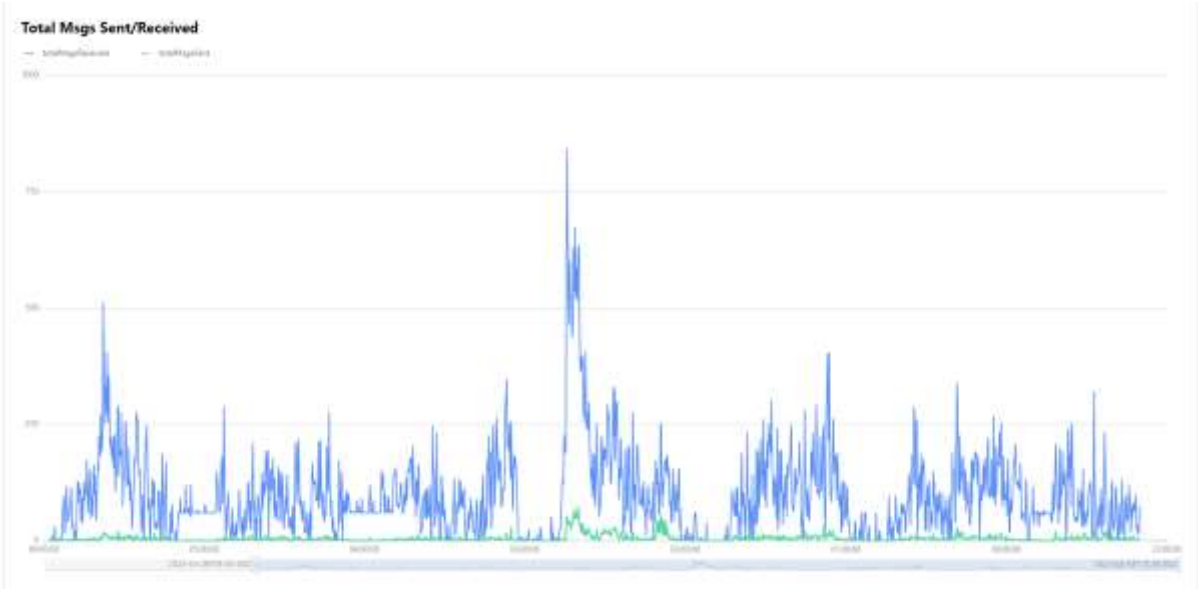
Fonte: Própria

Figura 49- Página de métricas do dashboard, hubs ativos



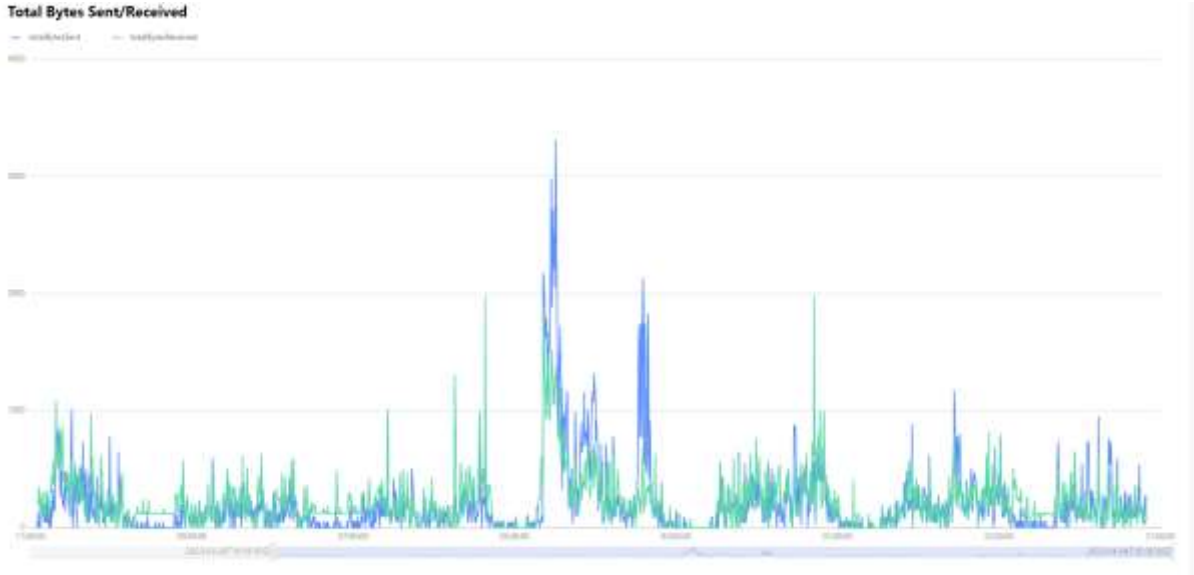
Fonte: Própria

Figura 50 - Página de métricas do dashboard, mensagens enviadas comparadas com mensagens recebidas



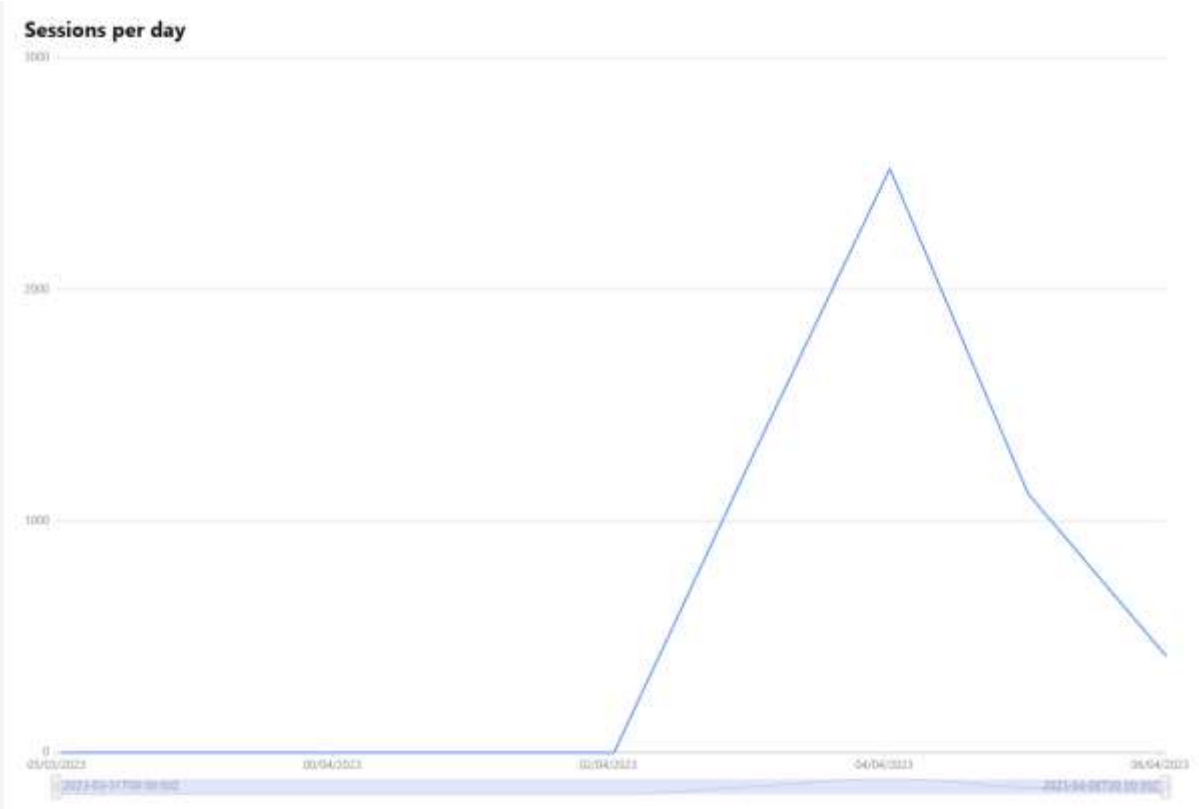
Fonte: Própria

Figura 51- Página de métricas do dashboard, bytes enviados comparados com bytes recebidos



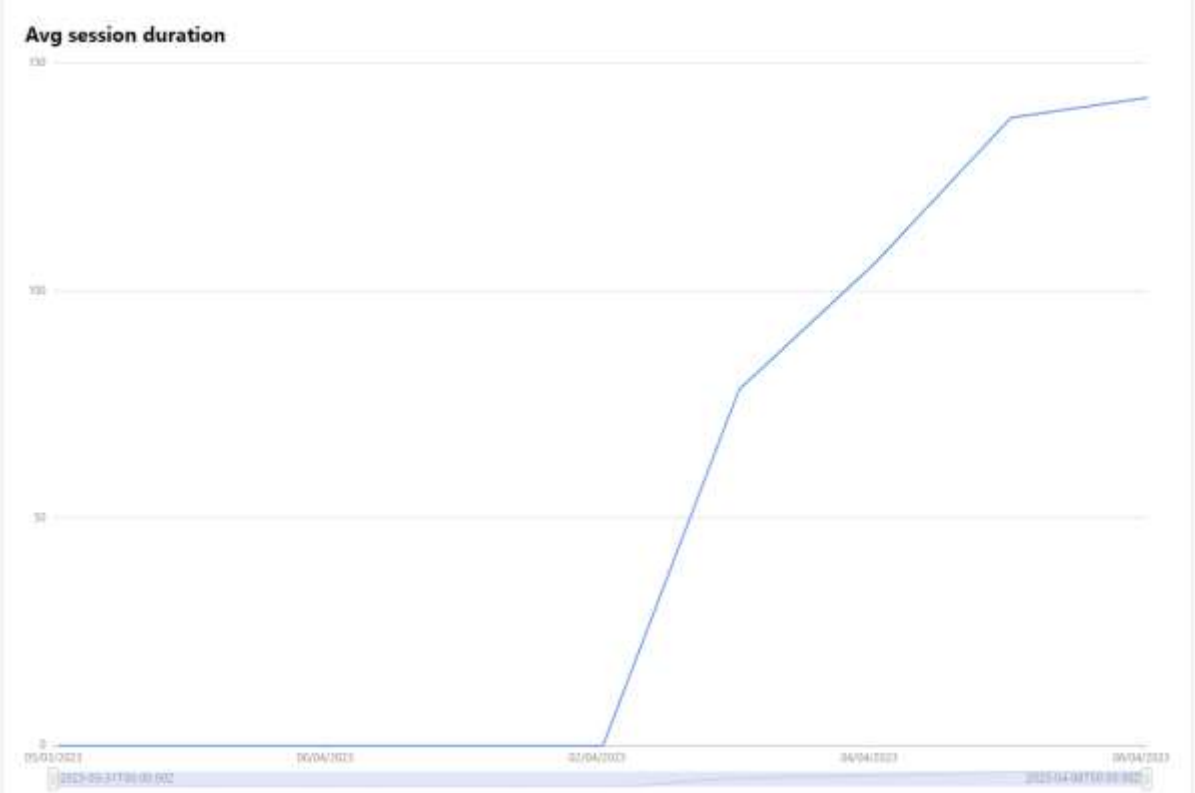
Fonte: Própria

Figura 52- Página de métricas do dashboard, sessões por dia



Fonte: Própria

Figura 53- Página de métricas do dashboard, média de duração de sessão por dia



Fonte: Própria

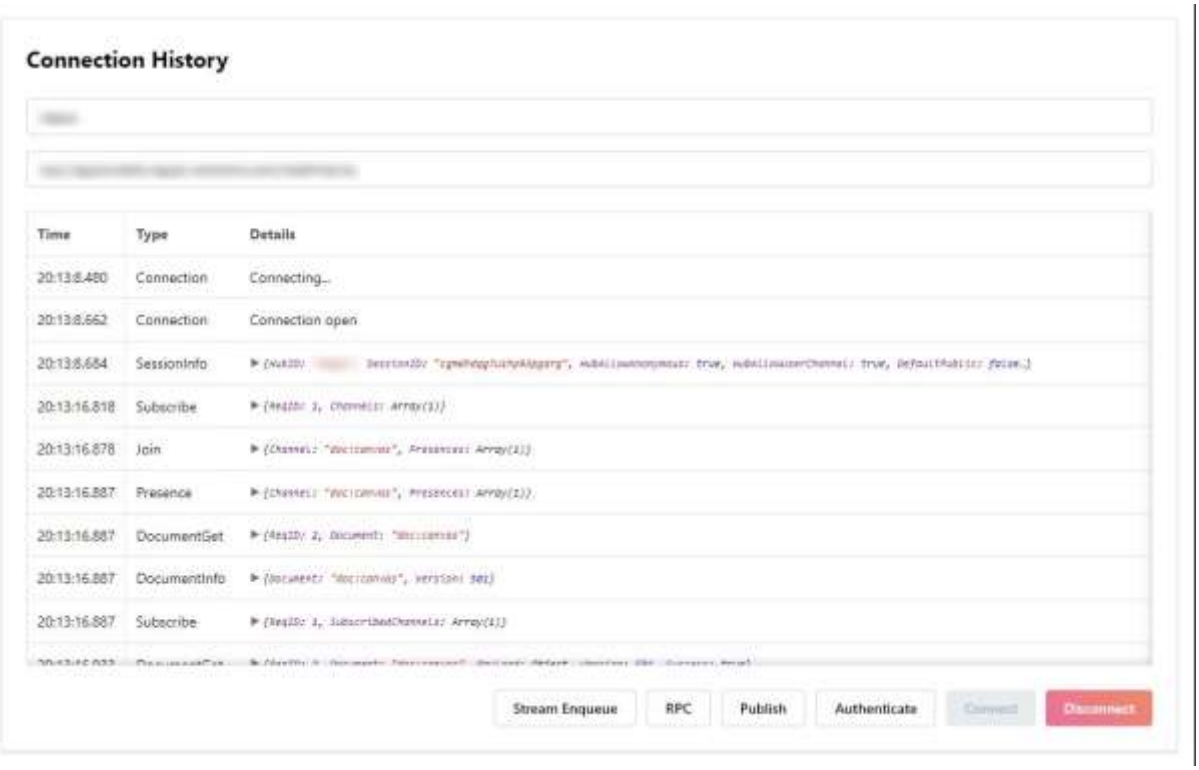
Figura 54 - Página de teste de sessão

Fonte: Própria

Figura 55 - Página de teste de sessão, detalhes de sessão

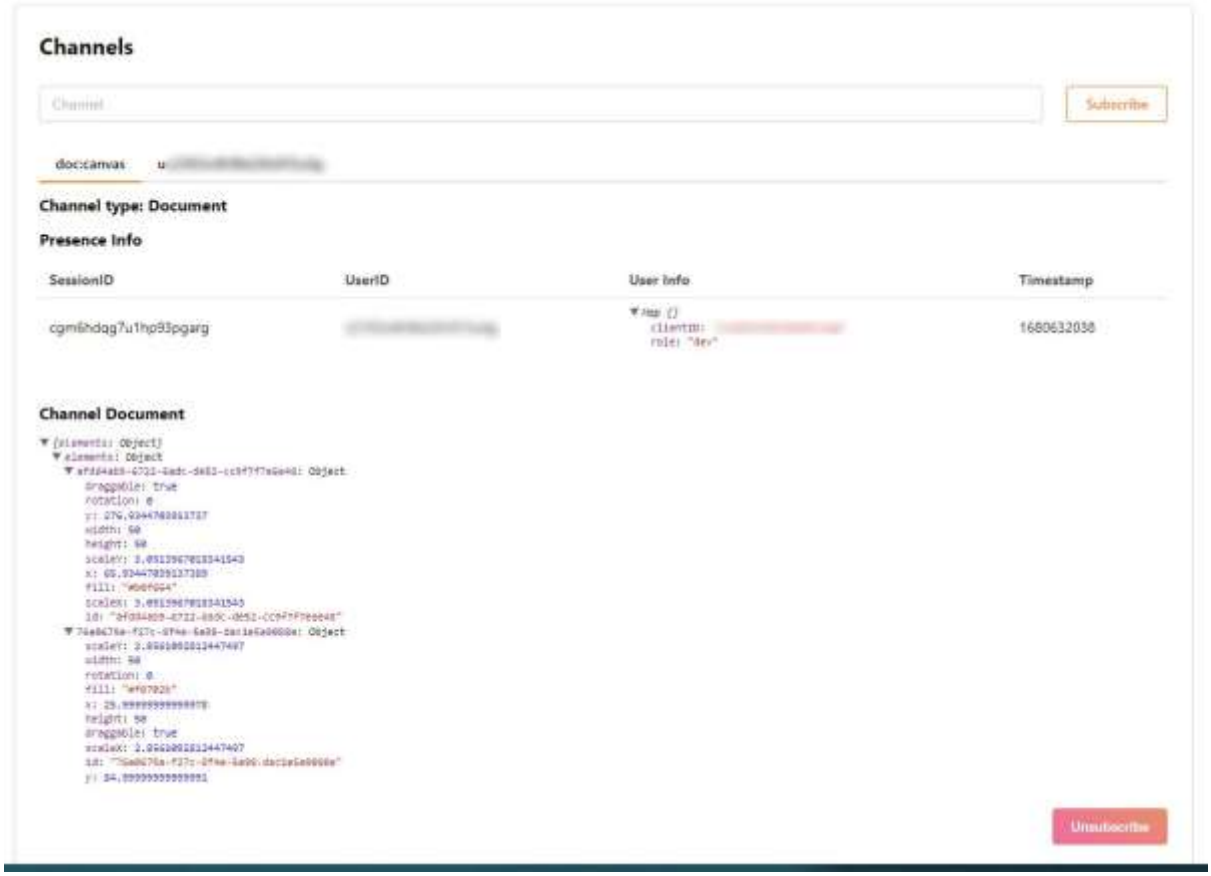
Fonte: Própria

Figura 56- Página de teste de sessão, histórico de conexão



Fonte: Própria

Figura 57 - Página de teste de sessão, histórico de channel



Fonte: Própria

APÊNDICE G

```
package protocol;

import "google/protobuf/timestamp.proto";

enum EventType {
  ...
  Publish = #;
  Ack = #;
  Subscribe = #;
  ...
}

/**
  Default response message
 */
message Response {

  /**
    Request id for req/res matching
   */
  int64 ReqID = 1 [json_name = "reqID"];

  /**
```

```
    If change went ok
   */
  bool Success = 2 [json_name = "success"];

  /**
    On publishes you may algo get the created message ID
   */
  optional string MsgID = 3 [json_name = "msgID"];
}

/**
  Response for ChannelPubRequest
 */
message ChannelPubResponse {
  repeated Response Published = 1 [json_name = "published"];
}

/**
  Request to batch publish
 */
message ChannelPubRequest {
  /**
    List of publishes to make
   */
  repeated PubRequestInfo Publishes = 1 [json_name = "publishes"];
}

/**
  Publish request information
 */
message PubRequestInfo {
  /**
    Publish target channel
   */
  string Channel = 1 [json_name = "channel"];

  /**
    Publish event name (user defined)
   */
  string Event = 2 [json_name = "event"];

  /**
    Publish payload
   */
  bytes Payload = 3 [json_name = "payload"];

  /**
    Extra features to apply
   */
  optional ChannelPubExtras Extra = 4 [json_name = "extra"];

  /**
    Request id for req/res matching
   */
  optional int64 ReqID = 5 [json_name = "reqID"];
}

/**
```

```
  Extra possible feature, will only work if channel supports it
 */
message ChannelPubExtras {
  /**
    If publish should be stored
   */
  bool store = 1 [json_name = "store"];

  /**
    If publish should trigger a push notification
   */
  bool push = 2 [json_name = "push"];

  /**
    If publish should be retained
   */
  bool retain = 3 [json_name = "retain"];
}

/**
  Request to batch subscribe
 */
message SubscribeRequest {

  /**
    Request id for req/res matching
   */
  int64 ReqID = 1 [json_name = "reqID"];

  /**
    List of channels to subscribe
   */
  repeated ChannelSubscribeInfo Channels = 2 [json_name = "channels"];
}

/**
  Response for SubscribeRequest
 */
message SubscribeResponse {

  /**
    Request id for req/res matching
   */
  int64 ReqID = 1 [json_name = "reqID"];

  /**
    List of channels that subscribe worked
   */
  repeated string SubscribedChannels = 2 [json_name =
"subscribedChannels"];
}

/**
  Channel to be subscribed and last client known timestamp for message
recover
 */
message ChannelSubscribeInfo {
  string Channel = 1 [json_name = "channel"];
```

```
  optional int64 Timestamp = 2 [json_name = "timestamp"];
}

/**
  Request to batch unsubscribe
 */
message UnsubscribeRequest {

  /**
    Request id for req/res matching
   */
  int64 ReqID = 1 [json_name = "reqID"];

  /**
    List of channels to unsubscribe from
   */
  repeated string Channels = 2 [json_name = "channels"];
}

/**
  Wrapper for every single message, it defines it's purpose and with
payload if available
 */
message Envelope {
  /**
    The reason for the message
   */
  EventType Event = 1 [json_name = "event"];

  /**
    The payload associated with the message if available
   */
  bytes Payload = 2 [json_name = "payload"];
}

/**
  Message for a channel publish
 */
message ChannelPublish {
  /**
   Target channel
  */
  string Channel = 1 [json_name = "channel"];

  /**
   Event of the publish (user defined)
  */
  string Event = 2 [json_name = "event"];

  /**
    Payload of the publish
   */
  bytes Payload = 3 [json_name = "payload"];

  /**
    Generated msgID
   */
  string MsgID = 4 [json_name = "msgID"];
```

```
  /**
    Timestamp of the publish
   */
  int64  Timestamp = 5 [json_name = "ts"];

  /**
    Node generated sequence
   */
  optional string ChannelSerial = 6 [json_name = "channelSerial"];
}
```

APÊNDICE H

Análise e Especificação de Requisitos

AppSockets

Versão 1.1

Preparada por Tiago Lima

17/05/2023

Índice

Lista de Tabelas do SRS

Histórico de Revisões

Nome	Data	Motivo da Revisão	Versão
Tiago Lima	2023-04	Documento Inicial	1.0
Tiago Lima	2023-05	Atualizações de requisitos funcionais	1.1

1. Introdução

1.1 Objetivos

O projeto ***AppSockets*** tem como objetivo servir de infraestrutura para o envio de informação em *soft real-time* entre clientes e servidores, e ao mesmo tempo substituir um sistema com objetivos similares, mantendo o máximo de compatibilidade possível de forma a facilitar a migração para o novo um sistema.

NAPPS é uma empresa SaaS (software como serviços) B2B (de empresa para empresa) com o propósito de construir a melhor experiência para os seus clientes e utilizadores, assim sendo, a empresa lida maioritariamente com aplicações móveis em contexto de *e-commerce*, e neste contexto surgiu uma necessidade de comunicar com as aplicações de forma quase instantânea sempre que a aplicação esteja em execução.

Para esse propósito, é necessário criar infraestrutura para o envio de informação em tempo real de forma bidirecional entre cliente e servidores. A infraestrutura não será exclusiva às aplicações móveis, tornando possível a sua utilização por outros serviços que possam necessitar de comunicação em tempo real.

1.2 Público-Alvo e Sugestões de Leitura

Este documento tem como principais destinatários os elementos da equipa de desenvolvimento e gestão de projeto, adicionalmente, também poderá ser utilizado pelo conjunto de utilizadores que pretendam contribuir e/ou maximizar a sua experiência enquanto potenciais *stakeholders*. Este documento poderá ainda ser distribuído por todos os outros *stakeholders* associados ao projeto.

Este SRS apresenta uma descrição detalhada do projeto AppSockets, bem como a descrição das características e de todo o conjunto de requisitos funcionais e não-funcionais acordados.

1.3 Âmbito do Projeto

O projeto AppSockets permitirá substituir um sistema interno existente enquanto adiciona novas funcionalidades. Sendo que já existe um projeto interno em produção com esta responsabilidade é necessário manter compatibilidade com o seu funcionamento, enquanto novas funcionalidades são adicionadas, e problemas existentes com o projeto interno são corrigidos. Os principais problemas a ter em conta são:

- Não ser escalável;
- Exige muita configuração;

Portanto, este projeto tem como objetivos:

- Ser capaz de substituir o projeto atual em produção
- Adicionar funcionalidades além do projeto existente
- Ser extensível para novos casos de uso
- Reduzir as configurações necessárias para sua utilização
- Fornecer API para serviços internos
- Fornecer API para aplicações clientes

1.4 Glossário

Termo	Definição
Especificação de Requisitos	Documento que descreve todas as funções do sistema proposto, bem como os requisitos não-funcionais e restrições sob as quais deve operar.
Stakeholder	Qualquer entidade com interesse direto ou indireto neste projeto
Utilizador	Cliente que utilize as funcionalidades implementadas via API
API	API (de *Application Programming Interface*) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por aplicativos que não pretendem envolver-se em detalhes da implementação das mesmas, mas apenas usar os seus serviços.
SRS	SRS (de *Software Requirements Specification*) é a tradução de Relatório de Entrega de Requisitos
Deliverable	Termo utilizado em gestão de projetos para descrever um produto ou serviço produzido e entregue ao cliente no contexto do desenvolvimento do projeto
IoT	IoT é uma rede de dispositivos conectados à Internet que coletam e compartilham dados, permitindo a automação de processos e tomadas de decisão baseadas em dados. Os dispositivos IoT usam tecnologias de rede sem fio e podem ser usados em várias áreas, como saúde, transporte e indústria.
PubSub	Publish and Subscribe, um modelo de comunicação em que os participantes se comunicam por meio de mensagens transmitidas por um intermediário (broker). Neste modelo, os participantes são divididos em duas categorias: publishers e subscribers. Publishers são responsáveis por enviar mensagens para o intermediário, enquanto subscribers se inscrevem em determinados tópicos de interesse. Quando um publisher envia uma mensagem, o intermediário envia a mensagem para todos os subscribers que estão inscritos no tópico relevante. Este modelo é muito utilizado em sistemas distribuídos para comunicação assíncrona e escalável entre diferentes partes do sistema, permitindo a comunicação eficiente entre muitos participantes sem a necessidade de cada participante saber com quem está se comunicando. O pub/sub é amplamente utilizado em aplicações de IoT (Internet das Coisas), sistemas de mensagens e sistemas de eventos, permitindo que os participantes se comuniquem de forma eficiente e escalável. Ao longo deste documento o nome PubSub ou Pub/Sub será utilizado, tendo ambos o mesmo significado.
Subscriber	Um subscriber em PubSub é um componente que se inscreve em tópicos para receber mensagens publicadas por publishers, geralmente implementado como um programa que se conecta a um broker de mensagens. Ao receber uma mensagem, o subscriber pode executar uma ação ou armazenar a mensagem para processamento posterior.
Publisher	Um Publisher em PubSub é um componente que publica mensagens em um ou mais tópicos. Ele geralmente é implementado como um programa ou processo que se conecta a um broker de mensagens e envia as mensagens para o tópico correspondente. Ao publicar uma mensagem, o Publisher não sabe quais subscribers, se houver algum, receberão a mensagem. É responsabilidade do broker encaminhar a mensagem para os subscribers inscritos nos tópicos relevantes. O processo de publicação de mensagens é geralmente assíncrono e permite que os Publishers continuem a enviar mensagens sem serem afetados pelo processamento dos subscribers.
Broker	Um broker é um componente de software usado como intermediário na comunicação entre sistemas ou aplicativos. Ele recebe mensagens de um emissor e as encaminha para um ou mais recetores interessados em recebê-las. Os benefícios do uso de um broker incluem redução de complexidade do sistema, melhoria da escalabilidade e redução da sobrecarga de comunicação direta entre sistemas.
Cluster	Um cluster é um conjunto de computadores interconectados que trabalham juntos como um único sistema para realizar tarefas complexas. Os membros do cluster distribuem a carga de trabalho entre si, permitindo que as tarefas sejam realizadas mais rapidamente.

1.5 Referências

Existem outros documentos da NAPPS no âmbito deste projeto, que suportam a informação contida neste plano, nomeadamente:

- Plano de Projeto
- Proposta Formal
- Cronograma

Este documento de especificação de requisitos foi desenvolvido de acordo com as regras vigentes e padronizadas relativas à descrição de requisitos funcionais e não-funcionais.

1.6 Organização deste Documento

As próximas seções têm como objetivo a visão geral dos atores e perfis de acesso, das funcionalidades do produto final e dos *entregáveis* a produzir no contexto do projeto.

2. Descrição Geral

2.1 Classes de Utilizadores

O sistema AppSockets prevê a existência de *[k]* tipos de entidades (atores do sistema e *stakeholders*) e *[n]* perfis de acesso distintos, ambos detalhados nas tabelas seguintes:

Ator/*StakeHolder*	Descrição	Perfil
Cliente	Entidade para quem vai ser desenvolvida a aplicação	Cliente
Programador	Entidade com a função de desenvolver a aplicação.	Administrador
Utilizador	Entidade com a função de utilizar a aplicação.	Serviço
Tester	Entidade com a função de testar a aplicação.	Administrador

Tabela 4 - Lista de stakeholders previstos para o AppSockets

A tabela seguinte ilustra os tipos de entidades/perfis de acesso e respetivo nível de privilégios, que podem aceder ao sistema de informação AppSockets, de modo a utilizarem as funcionalidades permitidas a cada nível (descrição detalhada nos casos de uso em secção adiante deste SRS).

Perfil	Descrição	Nível
Cliente	O perfil de cliente consiste numa aplicação cliente que irá se conectar a aplicação, este irá pertencer a hub e terá as permissões que sejam dados pelo AuthProvider do hub. Esta entidade nunca terá permissões para manipular qualquer tipo de configuração na aplicação.	Cliente
Serviço	Perfil de serviço consiste nas aplicações que irão interagir com a aplicação por meios internos, têm acesso a quase todas funcionalidades disponíveis.	Service
Administrador	Perfil de administrador consiste em um utilizador interno com permissão para manualmente alterar configurações na aplicação, tendo acesso a todas funcionalidades.	Admin

Tabela 5 - Lista de Níveis de Acesso e Perfis previstos para o AppSockets

De forma a respeitar possíveis situações de exceção relativamente às permissões dos perfis propostos na tabela 2, sugere-se a modularização do sistema de S.I. em grandes unidades orientadas a objetivos, de maneira a poder gerir permissões não apenas ao nível do perfil (tabela 2) mas também ao nível do utilizador individual. Sugere-se assim a seguinte divisão modular:

Módulo	Descrição	Acesso
Módulo 1	Conjunto de funcionalidades e informação para o Cliente	Cliente
Módulo 2	Conjunto de funcionalidades e informação para o Serviço	Service
Módulo 3	Conjunto de funcionalidades e informação para o Administrador	Admin

Tabela 6 - Divisão modular proposta para o S.I.

2.2 Funcionalidades do Produto

O projeto AppSockets a ser construído consiste numa aplicação multi-tenant e distribuída, que permite a intercomunicação entre cliente e serviços em soft real-time.

Sendo uma aplicação multi-tenant, existe uma separação entre estes, sendo esta representada por hubs. Cada hub tem as suas próprias sessões dos clientes, os seus próprios channels e suas próprias configurações. Cada hub, pode fornecer a sua própria forma de autenticação através vez de um objeto nomeado de AuthProvider, este objeto permite definir como o pedido de autenticação será entregue sendo as opções possíveis: NATS e HTTP, adicionalmente, podem ser configurados cabeçalhos e rota do pedido.
Utilizando o método PubSub os clientes e serviços serão capazes de publicar eventos para outros elementos interessados, organizando os eventos enviados por *channels*, equivalente a tópicos.
Cada channel pode pertencer a único tenant, permitindo nomes de channels iguais entre tenants. Este tem as suas próprias configurações, que podem ser definidas por cada um, ou num grupo com o nome de namespace, caso não exista em nenhum, o tenant que é representado por um hub na aplicação terá sempre uma configuração para aplicar caso mais nenhuma exista. De forma que um channel pertença a um namespace, este deve começar com o nome do namespace e ter um separador (":") antes do nome do channel, por exemplo, "namespace:channel".

Além de permitir simplesmente PubSub, cada channel tem um conjunto de funcionalidades disponíveis, sendo estas:

- Retain Message;
- Store Message;
- Push Message;
- Presence;
- Public;
- Client Publish;
- Allow Anonymous;
- Occupancy;
- Channel Live History.

Além das funcionalidades, existem tipos de channels que podem suportar ou não as funcionalidades previamente definidas, este tipo de channels consistem nos seguintes:

- Default;
- Document;
- Notification.

As funcionalidades previamente descritas, são modulares, de forma a permitir criar tipos de channels e reutilizar as funcionalidades. Este tipo de channels são definidos através vez de prefixos no channels por exemplo "doc:channel" e "notification:channel" e sendo o tipo Default, o ativo por defeito.

As sessões que representam uma ligação de uma aplicação cliente, estas podem utilizar os protocolos de comunicação WebSocket e SSE. Cada uma destas pode pertencer a somente um hub, e podem

subscrever ou publicar para qualquer channel dentro do hub desde que tenham permissões para tal que são definidas na resposta recebida ao realizar o pedido de autenticação definido pelo AuthProvider.

Por fim, de forma a permitir que serviços sejam capazes de comunicar com a aplicação, existem duas APIs sendo uma por NATS para serviços e outra por HTTP tanto para serviços como administrador.

O capítulo 4 deste SRS é totalmente dedicado à descrição dos requisitos funcionais do AppSockets.

2.3 Ambiente de Operação

A aplicação será planeada para ser executada na plataforma AWS Fargate ou ECS (Elastic Container Service) e localmente em caso de desenvolvimento. Naturalmente será utilizado a tecnologia Docker.

No ambiente da plataforma AWS, é necessário que todas as instâncias estejam na mesma rede, ou ligação entre estes, adicionalmente, tem que ser permitido tráfego TCP e UDP e as seguintes portas devem estar abertas: 80, 8080, 4040, 9999, 7946. Estas são as portas definidas por defeito.

Em ambiente local, cabe ao utilizar de fornecer um ambiente que suporte o funcionamento da aplicação, este pode ser feito através do software Docker, ou uma rede que autorize tráfego TCP e UDP. Caso seja somente necessário executar uma instância o localhost deverá funcionar normalmente.

Além de configurações de rede, existem configurações que devem estar presentes em conjunto com a aplicação. Estas configurações adicionais são divididas nas seguintes categorias:

- Broker;
- Admin;
- Cluster;
- Engine;
- Hub;
- Session;
- Channel;
- Connection;
- Redis;
- SQL;
- Auth;
- Storage;
- Debug;
- Discovery.

Todas estas configurações devem estar presentes num ficheiro “config.yaml”, que deve ser criado pelo elemento que deseja executar a aplicação.

As especificações das configurações serão apresentadas nos seguintes tópicos.

2.4 Broker

A categoria Broker define as propriedades necessárias para a aplicação se conectar e autenticar com um servidor NATS.

As propriedades da categoria Broker, utilizam como prefixo “broker” e existem as seguintes propriedades:

“broker.host” - URL para o servidor NATS;

“broker.user” - Utilizador de autenticação;

“broker.password” - Password de autenticação;

“broker.subject” - Prefixo para todos os subjects utilizados na aplicação.

2.5 Admin

A categoria admin define como administradores são autenticados, nesta categoria existe somente uma propriedade “admin.url” que aponta para o url onde o pedido de autenticação vai ser enviado.

2.6 Cluster

Nesta categoria são definidas propriedades relativamente à segurança de comunicação entre os membros do cluster. Existem somente duas propriedades.

“cluster.membership.secretKey” - Chave partilhada entre todos os membros para iniciar uma comunicação segura.

“cluster.rpc.secretKey” - Chave partilhada utilizada para realizar pedidos RPC.

2.7 Engine

Nesta categoria, são definidas propriedades que afetam todos os utilizadores da aplicação. As propriedades existentes são as seguintes:

“engine.sessionMetricsInterval” - Intervalo definido entre coleta o número de sessões.

“engine.allowDynamicHubs” - Se é permitido a criação dinâmica de hubs sem antes ser definido.

“engine.maxLiveHistorySize” - Número máximo de mensagens armazenada em live history.

“engine.statsCollectInterval” - Intervalo definido entre coleta de métricas gerais.

2.8 Hub

Na categoria hub, são definidas propriedades relativas a temporizações do hubs, com as propriedades “hub.closeTimeout”, onde é definido o intervalo de espera para fechar o hub assim que não tenha nenhuma sessão conectada. Por fim, existe a propriedade “hub.authenticateRequestTimeout” que define o tempo máximo de espera para o pedido de autenticação de uma sessão.

2.9 Channel

Na categoria channel, são definidas propriedades relativamente ao channel, consistindo nas seguintes propriedades.

“channel.historyPageSize” - Número de mensagens enviadas ao tentar recuperar mensagens, ou seja, controla o número máximo de paginação ao recuperar mensagens perdidas.

“channel.enableLiveHistory” - Se a funcionalidade Live History deve ser ativa ou não.

“channel.retainMsgDuration” - Define durante quanto tempo uma retained message deve ser armazenada.

2.10 Connection

Na categoria connection são definidas propriedades relativas aos tipos de conexões, nomeadamente SSE e WebSockets.

Para SSE existe somente a propriedade “connection.sse.msgBufferSize”, onde é definido o número máximo de mensagens em espera a serem enviadas para o cliente, mensagens recebidas além quando este buffer está no limite são ignoradas.

Para WebSockets existem três propriedades, sendo “connection.ws.msgSendBufferSize” que é equivalente à de SSE e “connection.ws.pingInterval” e “connection.ws.pongInterval” onde são definidos os intervalos para o PING/PONG dos WebSockets de forma a garantir que a conexão de mantêm aberta.

2.11 Redis

Assim como o nome da categoria indicada, aqui são definidas as propriedades para conectar à aplicação Redis ou um cluster da aplicação Redis.

“redis.address” - URL para o servidor Redis.

“redis.username” - Username de autenticação;

“redis.password” - Password de autenticação;

“redis.db” - Número de base de dados a ser utilizada.

2.12 SQL

propriedades relativas às conexões SQL em geral, com as seguintes propriedades.

“sql.idleConnections” - Número máximo de conexões paradas.

“sql.maxConnections” - Número máximo de conexões permitida.

“sql.connectionMaxLifetime” - Duração máxima de uma conexão.

2.13 Auth

Nesta categoria são definidas propriedades para a criação de tokens pela aplicação, com a seguintes propriedades.

“auth.algorithm” - Algoritmo de encriptação do token.

“auth.issuer” - Nome do criador do token;

“auth.session.secret” - Chave para encriptar e desencriptar tokens de sessões;

“auth.admin.secret” - Chave para encriptar e desencriptar tokens de administradores;

2.14 Storage

Nesta categoria é definido o tipo de base de dados a ser utilizada, por agora somente a base de dados PostgreSQL é suportada e com as seguintes propriedades.

“storage.postgres.host” - URL para o servidor ou proxy.

“storage.postgres.port” - Porta a ser utilizada.

“storage.sslMode” - Se SSL deve ser utilizada na ligação.

“storage.user” - User para autenticação.

“storage.password” - Password para autenticação.

“storage.dbName” - Nome da base de dados a ser utilizada.

2.15 Debug

Nesta categoria são definidas apenas algumas propriedades para permitir o inspecionamento da aplicação como “debug.pprof” se o pprof deve ser ativo, e “debug.level” para definir a verbosidade dos logs da aplicação.

2.16 Discovery

Nesta categoria existe somente um proprieadade onde é definido como o descobrimento de outros servidores deve ser realizada com a propriedade “discovery.type” que apenas suporta “udp” e “aws_ecs”.

A aplicação WEB será desenvolvida com suporte de dados baseado em PostgreSQL e suporte aplicacional baseado em Go. As APIs fornecidas por HTTP irão utilizar como formato de comunicação de informação o formato JSON, exceto nas APIs de conexão contínua onde será utilizado o formato Protobuf. A API fornecida pelo software NATS irá utilizar Protobuf como formato de comunicação de informação.

2.17 Restrições de Desenho e Implementação

O sistema será desenhado e implementado de modo a cumprir as recomendações de segurança da norma 27001, garantindo assim, desde logo, os requisitos de privacidade, confidencialidade e proteção de dados pessoais e legais que devem ser assegurados em aplicações deste tipo e área.
A comunicação com as API de interoperabilidade será efetuada de forma totalmente encriptada, sendo recomendado desde logo um acesso baseado em protocolo HTTPS.

2.18 Documentação para os Utilizadores

Estão previstos dois tipos de manuais para os dois tipos de utilizadores definidos no contexto do AppSockets: para a equipa técnica/informática, que pode efetuar a manutenção aos produtos que vão utilizar ou integrar com as API do AppSockets, está previsto um manual em formato digital com a lista das funcionalidades e respetivo contexto de utilização, para as funções da API.
Para os utilizadores da aplicação, está prevista a possibilidade de ligar/desligar os tutoriais da aplicação de forma contextual (ajuda por funcionalidade ativa).

2.19 Pressupostos e Dependências

Esta projeto tem as seguintes dependências:

- Existência de uma base de dados Postgres;
- Rede de internet com permissões para UDP e TCP e que todas instâncias tenham ligações entre estes;
- Existência de um servidor ou cluster Redis;
- Existência de um servidor ou cluster NATS;

Todas estas dependências externas devem estar acessíveis a todas as instâncias da aplicação ao longo do seu funcionamento.

A aplicação tem a capacidade de ser executada no sistema operativo Windows, MacOS e as distribuições mais comuns de Linux, sendo o sistema operativo principal a distribuição de Linux desenvolvido pela AWS.

Naturalmente, sendo este projeto para ser utilizado na plataforma AWS, espera-se que assim exista e com acesso aos recursos: AWS Load Balancer, Target Groups, ECS, Cloud Watch e ferramentas complementares destes.

3. Requisitos das Interfaces Externas

3.1 Interfaces de Utilizador

Uma vez que o sistema a desenvolver fornece somente APIs a serem utilizadas por outras aplicações, não estão previstas interfaces visuais.

3.2 Interfaces de *Hardware*

Uma vez que o sistema a desenvolver assenta em *frameworks* padronizados e devidamente estabelecidos, não estão previstas ligações ou interfaces especiais ao *hardware* dos dispositivos.

3.3 Interfaces de *Software* e de Comunicação

Este projeto consiste numa aplicação que irá correr num servidor, sem uma interface visual para utilizadores, no entanto, existe comunicação com outros serviços através do software NATS. Desta forma, existem duas interfaces de software, sendo uma API exposta por HTTP que espera pedidos em JSON e responde da mesma forma, e uma API acessível através do software NATS com o formato de mensagens definidas em protocol buffers.

De lembrar, que toda comunicação com interfaces não está assegurada com SSL providência pela aplicação, mas sim espera-se que esta seja providência pelo software NATS e no caso de HTTP através do serviço AWS Load Balancer.

4. Requisitos Funcionais do Sistema

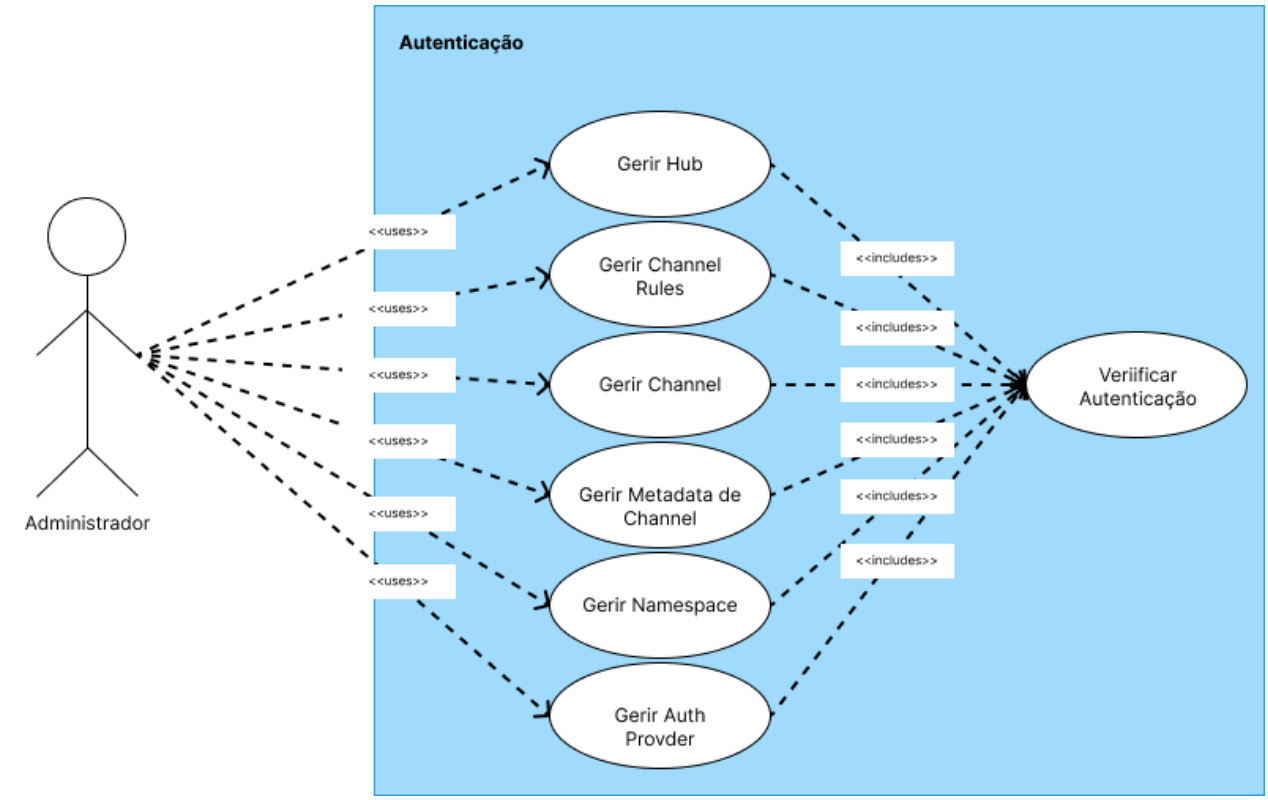
Esta secção descreve detalhadamente todos os requisitos funcionais do AppSockets e a enumeração dos requisitos não-funcionais que possam estar relacionados com a funcionalidade.

4.1 Vista Geral dos Requisitos Funcionais do S.I.

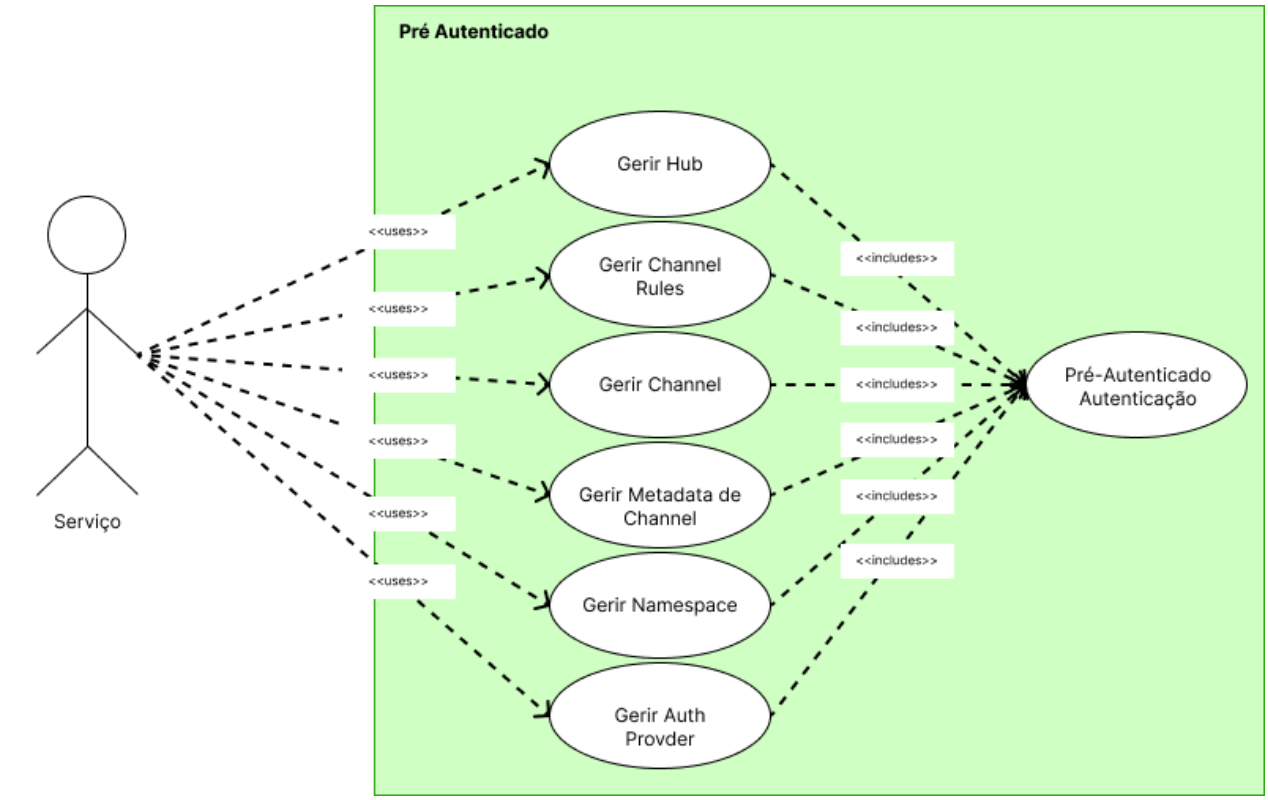
A figura seguinte ilustra as grandes funcionalidades do AppSockets para o perfil Cliente:

Identificador	Visão Geral Requisitos Funcionais AppSockets para Clientes
	
Descrição	O ator cliente acede à aplicação AppSockets
Requisitos não-funcionais	Ligação à Internet Ativa

A figura seguinte ilustra as grandes funcionalidades do AppSockets para o perfil Administrador:

Identificador	Visão Geral Requisitos Funcionais AppSockets para Administrador
	
Descrição	O ator Administrador gere a aplicação AppSockets
Requisitos não-funcionais	Ligação à Internet Ativa

A figura seguinte ilustra as grandes funcionalidades do AppSockets para o perfil Serviço:

Identificador	Visão Geral Requisitos Funcionais AppSockets para Serviços
	
Descrição	O ator serviço acede à aplicação AppSockets
Requisitos não-funcionais	Ligação à Internet Ativa Conexão com o software NATS Conexão autenticada e autorizada no software NATS

4.2 Os Requisitos Funcionais para Autenticação

Com o objetivo de controlar o acesso e ações do utilizador no contexto do AppSockets, estão previstas as seguintes funcionalidades:

4.2.1 Visão Geral das Funcionalidades de Autenticação

Identificador & Nome	Cliente:RF:01 – Conectar
Descrição	Um utilizador conecta-se à aplicação e estabelece-se uma sessão, de forma a poder realizar outras operações.
Pré-Condições	Acesso à internet e dispositivo compatível.
Pós-Condições	Acesso aos recursos públicos da aplicação.
Requisitos não-funcionais	Acesso à internet

4.2.2 Requisito Funcional “Autenticar”

Identificador & Nome	Cliente:RF.02 - Autenticar
Descrição	Um utilizador realiza uma autenticação. De forma a realizar esta, o Auth Provider configurado no hub será utilizado.
Pré-Condições	Acesso à internet e dispositivo compatível. Existência prévia do hub caso hub dinâmicos não estejam permitidos. Existência de um Auth Provider e de um recetor do pedido configurado no Auth Provider. Uma sessão estabelecida, por um meio de comunicação bidirecional.
Pós-Condições	Acesso aos recursos públicos e autorizados da aplicação, caso a autenticação seja sucedida, caso contrário, dependendo se o hub permite utilizadores anônimos ou não a sessão será terminada.
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de autenticação, a aplicação irá redirecionar esta mensagem pelo meio definido no AuthProvider e irá aguardar uma resposta. Assim que recebida, as permissões da sessão serão atualizadas, e mensagens com estas mensagens serão enviadas para o cliente. Na eventualidade, da autenticação não for sucedida, seja devido a falhas ou informação inválida e o hub não permitir sessões anônimas, esta será terminada de imediato, caso contrário será somente enviada uma mensagem a notificar que a operação não foi bem sucedida.
Percursos Alternativos	Além do processo normal de autenticação, é possível iniciar uma sessão pré-autenticada, através da utilização de um token de autenticação ao estabelecer a sessão, ou ao restaurar uma sessão.
Req. não-funcionais	Acesso à internet. A sessão tem um tempo limite para se autenticar caso o hub não permita sessões anónimas.

4.3 Os Requisitos Funcionais para "Cliente"

4.3.1 Requisito Funcional "Publicar"

Identificador & Nome	Cliente:RF.03 - Publicar
Descrição	Um utilizador publica um evento num channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Uma sessão estabelecida, por um meio de comunicação bidirecional. Autorização para o channel a ser publicado.
Pós-Condições	A mensagem deve ser processada e enviada para todas sessões subscritas no channel.
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de publicação com o channel indicado e de um tipo que o permite. A aplicação irá receber esta mensagem, verificar a autorização e enviar a mensagem para o channel a ser processado. Caso a mensagem envie um identificador de mensagem, o cliente irá receber uma confirmação da publicação da mensagem. Na eventualidade, da sessão não ter permissões suficientes, será somente enviada uma mensagem a notificar que a operação não foi bem sucedida.
Percursos Alternativos	Além do processo normal de publicação numa sessão, é possível o fazer utilizando um pedido HTTP com um token de autenticação com permissões para o channel em questão.
Req. não-funcionais	Acesso à internet.

4.3.2 Requisito Funcional “Subscrever”

Identificador & Nome	Cliente:RF:04 - Subscrever
Descrição	Um utilizador subscreve-se a um channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Uma sessão estabelecida, por um meio de comunicação bidirecional. Autorização para o channel a ser subscrito.
Pós-Condições	Futuras mensagens no channel serão recebidas pela sessão.
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de subscrição com o channel indicado. A aplicação irá receber esta mensagem, verificar a autorização e adicionar a subscrição ao channel. O cliente receberá uma confirmação da subscrição da mensagem. Na eventualidade, da sessão não ter permissões suficientes, será somente enviada uma mensagem a notificar que a operação não foi bem sucedida.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.3.3 Requisito Funcional “Remover Subscrição”

Identificador & Nome	Cliente:RF.05 - Remover Subscrição
Descrição	Um utilizador remove a sua subscrição a um channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Uma sessão estabelecida, por um meio de comunicação bidirecional. Sessão já subscrita ao channel.
Pós-Condições	Futuras mensagens no channel não serão recebidas pela sessão.
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de remover a subscrição com o channel indicado. A aplicação irá receber esta mensagem, verificar se este está subscrito e remover a subscrição ao channel. O cliente receberá uma confirmação da remoção da subscrição da mensagem. Na eventualidade, de uma falha ocorrer, será enviada uma mensagem a notificar que a

	operação não foi bem sucedida.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.3.4 Requisito Funcional "RPC"

Identificador & Nome	Cliente:RF.06 - RPC
Descrição	Um utilizador realiza um RPC (Remote Procedure Call).
Pré-Condições	Acesso à internet e dispositivo compatível. Uma sessão estabelecida, por um meio de comunicação bidirecional. Sessão autenticada e com permissão para o RPC a realizar.
Pós-Condições	Não identificadas
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de RPC com o método indicado. A aplicação irá receber esta mensagem, verificar se a sessão tem autorização para o método e irá redirecionar o conteúdo da mensagem por NATS para um recipiente interessado no método e aguardar a sua resposta. Assim que a resposta seja recebida, esta seria enviada para o cliente. Na eventualidade, de uma falha ocorrer, será enviada uma mensagem a notificar que a operação não foi bem sucedida.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.3.5 Requisito Funcional "Pedir histórico de channel"

Identificador & Nome	Cliente:RF:07 - Pedir histórico de channel
Descrição	Um utilizador envia um pedido para receber o histórico de um channel e recebe uma página com o histórico disponível.
Pré-Condições	Acesso à internet e dispositivo compatível.

	Uma sessão estabelecida, por um meio de comunicação bidirecional. Sessão com permissão para o channel a ser utilizado.
Pós-Condições	Não identificadas.
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de pedir histórico de channel com o channel indicado. A aplicação irá receber esta mensagem, verificar se a sessão tem autorização para o pedido e buscar uma página de últimas mensagens que será enviada de volta para o cliente. Neste pedido, pode ser enviado a última timestamp que o cliente tenha, de forma a receber mensagens mais recentes do que a timestamp enviada. Na eventualidade, de uma falha ocorrer, será enviada uma mensagem a notificar que a operação não foi bem sucedida.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.3.6 Requisito Funcional “Buscar Documento”

Identificador & Nome	Cliente:RF:08 - Buscar Documento
Descrição	Um utilizador envia um pedido para receber o valor atual de um documento pertencente a um channel e o recebe.
Pré-Condições	Acesso à internet e dispositivo compatível. Uma sessão estabelecida, por um meio de comunicação bidirecional. Sessão com permissão para o channel a ser utilizado. O channel a ser utilizado deve ser do tipo channel.
Pós-Condições	Não identificadas
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de buscar um documento pertencente de um channel. A aplicação irá receber esta mensagem, verificar se a sessão tem autorização para o pedido e enviar uma cópia do documento do channel. Na eventualidade, de uma falha ocorrer será enviada uma mensagem a notificar que a operação não foi bem sucedida.
Percursos Alternativos	Não existem percursos alternativos.

Req. não-funcionais	Acesso à internet.

4.3.7 Requisito Funcional “Atualizar Documento”

Identificador & Nome	Cliente:RF.09 - Atualizar Documento
Descrição	Um utilizador envia um pedido para atualizar o documento de um channel e recebe uma página com o histórico disponível.
Pré-Condições	Acesso à internet e dispositivo compatível. Uma sessão estabelecida, por um meio de comunicação bidirecional. Sessão com permissão para o channel a ser utilizado.
Pós-Condições	O documento deve manter as alterações efetuadas e enviar as operações realizadas para todas as sessões subscritas.
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de atualização de documento com o channel indicado. A aplicação irá receber esta mensagem, verificar se a sessão tem autorização para o pedido e aplicar as alterações enviadas, posteriormente deve enviar estas alterações para todas sessões e enviar uma confirmação de que a operação foi realizada ao cliente. Na eventualidade, de uma falha ocorrer, será enviada uma mensagem a notificar que a operação não foi bem sucedida.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.3.8 Requisito Funcional “Marcar notificação como lida”

Identificador & Nome	Cliente:RF:10 - Marcar notificação como lida
Descrição	Um utilizador envia um pedido para marcar uma ou mais notificações como lidas num channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Uma sessão estabelecida, por um meio de comunicação bidirecional. Sessão com permissão para o channel a ser utilizado.

Pós-Condições	As notificações devem manter o seu estado atualizado de lido e devem notificar todos inscritos das alterações.
Percurso Normal	Utilizando uma sessão bidirecional ativa, o cliente envia uma mensagem definida no protocolo como de marcar a notificação como lida com o channel indicado. A aplicação irá receber esta mensagem, verificar se a sessão tem autorização para o pedido e aplicar as alterações enviadas, posteriormente deve enviar estas alterações para todas sessões e enviar uma confirmação de que a operação foi realizada ao cliente. Na eventualidade, de uma falha ocorrer, será enviada uma mensagem a notificar que a operação não foi bem sucedida. Adicionalmente, este pedido suporta realizar a alteração para um conjunto de de notificações de forma a reduzir a quantidade de pedidos feitos.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.3.9 Requisito Funcional “Registar Push Token”

Identificador & Nome	Cliente:RF:11 - Registrar Push Token
Descrição	Um utilizador envia um pedido para registar o seu push token, de forma que este fique associado.
Pré-Condições	Acesso à internet e dispositivo compatível. Uma sessão estabelecida, por um meio de comunicação bidirecional. Token de autenticação. Token de Firebase pré-existente.
Pós-Condições	O token deve ficar registrado e ser utilizado sempre que uma mensagem com a funcionalidade de Push Notification for enviada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o cliente envia um pedido HTTP com o token de autenticação e Firebase token. Este fica posteriormente associado ao identificador de utilizador. Na eventualidade, de uma falha ocorrer, será enviado o código de erro HTTP 500.
Percursos Alternativos	Não existem percursos alternativos.

Req. não-funcionais	Acesso à internet.

4.3.10 Requisito Funcional “Buscar notificações não lidas de um channel”

Identificador & Nome	Cliente:RF:12 - Buscar notificações não lidas de um channel
Descrição	Um utilizador envia um pedido HTTP para buscar notificações de um channel do tipo notification e recebe de forma paginada as notificações não lidas. A paginação é baseada em timestamps.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação com permissão para o channel a ser utilizado.
Pós-Condições	Nenhuma identificada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o cliente envia um pedido HTTP com o token de autenticação, o channel e última timestamp disponível, utilizando esta informação a aplicação irá retornar as notificações não lidas a seguir à timestamp. Na eventualidade, de uma falha ocorrer, será enviado o código de erro HTTP 500.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.3.11 Requisito Funcional “Buscar notificações de um channel”

Identificador & Nome	Cliente:RF.13 - Buscar notificações de um channel
Descrição	Um utilizador envia um pedido HTTP para buscar notificações de um channel do tipo notification e recebe de forma paginada as notificações. A paginação é baseada em timestamps.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação com permissão para o channel a ser utilizado.
Pós-Condições	Nenhuma identificada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o cliente envia um pedido HTTP com o token de autenticação, o channel e última timestamp disponível, utilizando esta informação a

	aplicação irá retornar as notificações a seguir à timestamp. Na eventualidade, de uma falha ocorrer, será enviado o código de erro HTTP 500.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4 Os Requisitos Funcionais para "Administrador"

4.4.1 Requisito Funcional "Criar Auth Provider para um Hub"

Identificador & Nome	Admin:RF:14 - Criar Auth Provider para um Hub
Descrição	Um administrador envia um pedido HTTP, com a informação necessária para criar um Auth Provider pertencente a um Hub.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	Auth Provider criado deverá ficar armazenado de forma a poder ser a utilizado posteriormente.
Percurso Normal	Utilizando o token de autenticação pré-existente, o cliente envia um pedido HTTP com o token de autenticação, o hub e configurações necessárias, utilizando esta informação a aplicação deverá criar um Auth Provider e armazená-lo e retornar este ao administrador. Na eventualidade, de uma falha ocorrer, será enviado o código de erro HTTP 500.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.2 Requisito Funcional "Buscar Auth Providers"

Identificador & Nome	Admin:RF:15 - Buscar Auth Providers
Descrição	Um administrador envia um pedido HTTP e recebe uma lista de todos Auth Providers existentes.
Pré-Condições	Acesso à internet e dispositivo compatível.

	Token de autenticação de administrador.
Pós-Condições	Nenhuma identificada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o cliente envia um pedido HTTP com o token de autenticação, a aplicação deverá retornar uma lista com todos Auth Providers previamente criados. Na eventualidade, de uma falha ocorrer, será enviado o código de erro HTTP 500.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.3 Requisito Funcional "Definir configurações para um channel"

Identificador & Nome	Admin:RF:16 - Definir configurações para um channel
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a atribuir configurações a um channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	As configurações aplicadas no channel devem ser propagadas por todas as instâncias pertencentes ao cluster.
Percurso Normal	Utilizando o token de autenticação pré-existente, o cliente envia um pedido HTTP com o token de autenticação, o hub, channel e configurações a serem aplicadas, utilizando esta informação a aplicação deverá armazenar as configurações e notificar todo o cluster que as configurações para aquele channel foram alteradas. Na eventualidade, de uma falha ocorrer, será enviado o código de erro HTTP 500.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.4 Requisito Funcional “Buscar configurações para um channel”

Identificador & Nome	Admin:RF:17 - Buscar configurações de um channel
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a retornar as configurações atuais de um channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	Nenhuma identificada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o cliente envia um pedido HTTP com o token de autenticação, o hub e channel, utilizando esta informação a aplicação retornar as configurações atualmente existentes no channel em questão. Na eventualidade, de uma falha ocorrer será enviado o código de erro HTTP 500.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.5 Requisito Funcional “Publicar em um channel”

Identificador & Nome	Admin:RF:18 - Publicar em um channel
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a um evento ser publicado num channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	O evento enviado deve ser enviado para todas sessões subscritas ao channel em questão.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub, channel e eventos. Utilizando esta informação a aplicação irá publicar os eventos no channel e retornar se as publicações ocorreram com sucesso por cada evento.
Percursos Alternativos	Não existem percursos alternativos.

Req. não-funcionais	Acesso à internet.

4.4.6 Requisito Funcional “Definir metadata de um channel”

Identificador & Nome	Admin:RF:19 - Definir metadata de um channel
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma que a metadata enviada seja aplicada a um channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	A metadata enviada deve ficar armazenada
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub, channel e metadata a ser aplicada. Utilizando esta informação a aplicação irá armazenar a metadata enviada substituindo valores existentes.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.7 Requisito Funcional “Buscar metadata de um channel”

Identificador & Nome	Admin:RF:20 - Buscar metadata de um channel
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a buscar os metadados definidos num channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	Nenhuma identificada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub e o channel. Utilizando esta informação a aplicação irá armazenar buscar a metadata do channel e a retornar ao cliente.
Percursos	Não existem percursos alternativos.

Alternativos	
Req. não-funcionais	Acesso à internet.

4.4.8 Requisito Funcional “Definir/Atualizar namespace”

Identificador & Nome	Admin:RF:21 - Definir/Atualizar namespace
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a criar um namespace
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	As configurações aplicadas no namespace devem ser propagadas por todas instâncias pertencentes ao cluster.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub, identificador de Channel Rules e nome do namespace. Utilizando esta informação a aplicação irá armazenar a existência do namespace e notificar todos membros do cluster da atualização/criação do namespace.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.9 Requisito Funcional “Eliminar namespace”

Identificador & Nome	Admin:RF:22 - Eliminar namespace
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a eliminar um namespace
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	A eliminação no namespace deve ser propagada por todas as instâncias pertencentes ao cluster.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com

	o token de autenticação, o hub, e nome do namespace. Utilizando esta informação a aplicação irá eliminar a definições armazenadas sobre a existência do namespace e notificar todos membros do cluster da eliminação do namespace.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.10 Requisito Funcional "Definir configurações de um hub"

Identificador & Nome	Admin:RF:23 - Definir configurações de um hub
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a criar ou atualizar as configurações de um Hub.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	As modificações no hub devem ser propagadas por todas as instâncias pertencentes ao cluster.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub e configurações a serem aplicadas. Utilizando esta informação a aplicação armazena as configurações do hub e notifica todos membros do cluster das alterações do hub.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.11 Requisito Funcional "Buscar configurações de um hub"

Identificador & Nome	Admin:RF:24 - Buscar configurações de um hub
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a criar ou atualizar as configurações de um Hub.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.

Pós-Condições	Nenhuma identificada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação e o hub. Utilizando esta informação a aplicação ira buscar as configurações armazenadas do hub e retornar ao cliente.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.12 Requisito Funcional "Criar notificação num channel"

Identificador & Nome	Admin:RF:25 - Criar notificação num channel
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a criar uma notificação num channel
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	As sessões subscritas devem receber a nova notificação criada e uma atualização no número de notificações não lidas.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub, channel e notificação. Utilizando esta informação a aplicação ira processar a notificação, enviar para as sessões subscritas e enviar uma confirmação de operação ao cliente. Na eventualidade de um erro ocorrer, uma resposta com HTTP 500 será retornada.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.13 Requisito Funcional “Buscar informação de presença de um channel”

Identificador & Nome	Admin:RF:26 - Buscar informação de presença de um channel
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a buscar a presença atual num channel
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador. O channel deve ter a funcionalidade de presença ativa.
Pós-Condições	Nenhuma identificada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub e channel. Utilizando esta informação a aplicação ira copiar a informação de presença do channel atual e enviar ao cliente. Na eventualidade de um erro ocorrer, uma resposta com HTTP 500 será retornada.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.14 Requisito Funcional “Definir channel rules”

Identificador & Nome	Admin:RF:27 - Definir channel rules
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a definir ou atualizar um channel rules.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	Todos os channels e hub devem ser notificados caso estejam a usar o channel rule atualizado.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub, identificador de channel rules e configurações. Utilizando esta informação a aplicação armazena as novas configurações e notifica todas as instâncias do cluster da alteração do channel rules.

	Na eventualidade de um erro ocorrer, uma resposta com HTTP 500 será retornada.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.4.15 Requisito Funcional “Buscar channel rules”

Identificador & Nome	Admin:RF:28 - Buscar channel rules
Descrição	Um administrador envia um pedido HTTP com a informação necessária de forma a buscar as configurações definidas num channel rules.
Pré-Condições	Acesso à internet e dispositivo compatível. Token de autenticação de administrador.
Pós-Condições	Nenhuma identificada.
Percurso Normal	Utilizando o token de autenticação pré-existente, o administrador envia um pedido HTTP com o token de autenticação, o hub e identificador de channel rules. Utilizando esta informação a aplicação busca o armazenamento das configurações atuais e retornar ao cliente. Na eventualidade de um erro ocorrer, uma resposta com HTTP 500 será retornada.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5 Os Requisitos Funcionais para "Serviço"

4.5.1 Requisito Funcional "Publicar num channel"

Identificador & Nome	Service:RF:29 - Publicar num channel
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para publicar um ou mais eventos num channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	O evento enviado deve ser enviado para todas sessões subscritas ao channel em questão.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub, channel e eventos, utilizando esta informação a aplicação irá publicar os eventos no channel e retornar se as publicações ocorreram com sucesso por cada evento. Na eventualidade, de uma falha ocorrer, será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5.2 Requisito Funcional "Criar notificação num channel"

Identificador & Nome	Service:RF:30 - Criar notificação num channel
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para criar uma notificação num channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	A notificação criada deve ser enviada para todas sessões subscritas ao channel em questão.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub, channel e notificação, utilizando esta informação a aplicação irá publicar a notificação no channel e

	retornar se esta ocorreu com sucesso. Na eventualidade, de uma falha ocorrer, será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5.3 Requisito Funcional “Definir metadata num channel”

Identificador & Nome	Service:RF:31 - Definir metadata num channel
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para definir ou atualizar a metadata de um channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	A metadata deve ficar armazenada.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub, channel e metadata, utilizando esta informação a aplicação irá atualizar o metadata de um channel substituindo valores já existentes na base de dados. Na eventualidade, de uma falha ocorrer, será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5.4 Requisito Funcional “Buscar metadata de um channel”

Identificador & Nome	Service:RF:32 - Buscar metadata de um channel
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para buscar a metadata de um channel.

Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	Nenhuma identificada
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub e channel, utilizando esta informação a aplicação irá retornar a metadata do channel armazenados na base de dados. Na eventualidade, de uma falha ocorrer será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5.5 Requisito Funcional “Definir/Atualizar namespace”

Identificador & Nome	Service:RF:33 - Definir/Atualizar namespace
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para buscar a metadata de um channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	As configurações aplicadas no namespace devem ser propagadas por todas instâncias pertencentes ao cluster.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub, identificador de channel rules e nome do namespace. Utilizando esta informação a aplicação irá armazenar a existência do namespace e notificar todos membros do cluster da atualização/criação do namespace, e por fim retorna se a operação ocorreu com sucesso. Na eventualidade, de uma falha ocorrer, será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.

Req. não-funcionais	Acesso à internet.

4.5.6 Requisito Funcional “Eliminar namespace”

Identificador & Nome	Service:RF:34 - Eliminar namespace
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para eliminar um namespace.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	A eliminação no namespace deve ser propagada por todas as instâncias pertencentes ao cluster.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub e nome do namespace. Utilizando esta informação a aplicação irá eliminar a definições armazenadas sobre a existência do namespace e notificar todos membros do cluster da eliminação do namespace. Na eventualidade, de uma falha ocorrer será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5.7 Requisito Funcional “Definir configurações num channel”

Identificador & Nome	Service:RF:35 - Definir configurações num channel
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para definir as configurações de um channel.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.

Pós-Condições	As configurações aplicadas no channel devem ser propagadas por todas instâncias pertencentes ao cluster.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub, channel e configurações a serem aplicadas. Utilizando esta informação a aplicação deverá armazenar as configurações e notificar todo o cluster que as configurações para aquele channel foram alteradas. Na eventualidade, de uma falha ocorrer será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5.8 Requisito Funcional “Definir Channel Rules”

Identificador & Nome	Service:RF:36 - Definir Channel Rules
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para criar um channel rules.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	Todos os channels e hub devem ser notificados caso estejam a usar o channel rule atualizado.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub, identificador de channel rules e configurações a serem aplicadas. Utilizando esta informação a aplicação armazena as novas configurações e notifica todas as instâncias do cluster da alteração do channel rules. Na eventualidade, de uma falha ocorrer, será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5.9 Requisito Funcional “Publicar num channel”

Identificador & Nome	Service:RF:37 - Definir configurações num hub
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para definir as configurações de um hub.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	As modificações no hub devem ser propagadas por todas as instâncias pertencentes ao cluster.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub e configurações a serem aplicadas. Utilizando esta informação a aplicação armazenar as configurações do hub e notificar todos membros do cluster das alterações do hub. Na eventualidade, de uma falha ocorrer, será enviado uma mensagem no tipo NACK (Negative Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

4.5.10 Requisito Funcional “Publicar num channel”

Identificador & Nome	Service:RF:38 - Criar Auth Provider
Descrição	Um serviço envia um pedido através do NATS com a informação necessária para criar um Auth Provider pertencente a um Hub.
Pré-Condições	Acesso à internet e dispositivo compatível. Conexão autenticada com NATS. Permissões para o subjects no NATS.
Pós-Condições	Auth Provider criado deverá ficar armazenado de forma a poder ser a utilizado posteriormente.
Percurso Normal	Utilizando a conexão com o NATS, o serviço envia um pedido com o hub e configurações a serem aplicadas. Utilizando esta informação a aplicação deverá criar um Auth Provider e armazená-lo e retornar este ao serviço. Na eventualidade, de uma falha ocorrer será enviado uma mensagem no tipo NACK (Negative

	Acknowledge), definido pelo software NATS.
Percursos Alternativos	Não existem percursos alternativos.
Req. não-funcionais	Acesso à internet.

5. Requisitos Não-Funcionais

5.1 Requisitos de Performance

A performance e escalabilidade são dos pontos importantes desta aplicação, assim sendo, foram tomadas decisões sobre tecnologias a utilizar e métodos a aplicar. Alguns exemplos, passam pela utilização da linguagem Go, sendo uma linguagem com baixo consumo de memória e bastante eficiente, esta linguagem é importante visto que permite um controle mais granular sobre quais partes ficam no stack ou no heap, a utilização de apontadores de memória para um maior controlo sobre cópias de informação e a existência de lightweight threads de forma a permitir que aplicação utilize todos os núcleos do processador disponíveis. Toda a informação na aplicação é mantida em cache sempre que possível, evitando realizar pedidos à base de dados em partes da aplicação muito executados, sacrificando a velocidade de atualização da aplicação por um maior desempenho e latência reduzida.

A nível de escalabilidade é necessário que esta aplicação seja distribuída de forma a ter mais instâncias e maior capacidade de processamento em paralelo, adicionalmente, esta deverá evitar pontos únicos de falha. De forma a realizar estes pontos, a aplicação irá utilizar o protocolo gossip de forma a ser distribuída e evitar a existência de um líder central de informação.

5.2 Requisitos de Proteção

A aplicação não irá gerir dados sensíveis, toda a informação enviada pela aplicação é completamente transparente a esta, toda a informação é tratada como um conjunto de bytes e a pouca que é armazenada não é modificável sem ser pela base de dados diretamente, não sendo uma responsabilidade da aplicação.

Não é recomendado armazenar informações sensíveis em metadados de um channel, e o acesso a channels com metadados deve ser apenas permitida a utilizadores autenticados. Adicionalmente, não é recomendado o envio de informação sensível em eventos, caso seja necessário, o conteúdo dentro do evento pode ser encriptado pelo cliente, como já referido a aplicação não lê o conteúdo do evento.

5.3 Requisitos de Segurança

Esta aplicação permite a utilização por utilizadores anónimos, no entanto, é possível não os permite sendo passado essa responsabilidade a quem utiliza o Hub, adicionalmente, é possível definir configurações restritas por defeito a todos os channels do hub por defeito, definindo somente em alguns channels configurações mais permissíveis.

Os caminhos indicados no Auth Provider devem enviar a informação por protocolos com encriptação, sendo no caso de HTTP a utilização de SSL/TLS, e no caso do NATS uma conexão encriptada e com permissões restritas à API disponível por este. Adicionalmente, não é necessário enviar informação sensível para autenticar um utilizador, no conteúdo pode ser enviado um token, hash de password ou similares, sendo que a verificação de autenticação e permissões de uma sessão cabe ao destinatário do Auth Provider.

Em casos de sistemas já existentes onde tokens são utilizados para autenticação, estes podem ser usados como conteúdo de autenticação, tendo o destinatário do Auth Provider somente confirmar se o token é válido.

Todos os tokens gerados devem ser de curta duração e armazenados de forma temporário e segura nos dispositivos onde estes são utilizados.

Por fim, a utilização de HTTPS pelo cliente é recomendada e deveria ser obrigatória, sendo o caso no ambiente de execução na plataforma AWS.

5.4 Requisitos Não-Funcionais das Regras de Negócio

Estão contemplados os seguintes requisitos não-funcionais relacionados com regras de negócio:

- RNF001: A aplicação AppSockets deve estar sempre ligada à Internet/Intranet
- RNF002: Para utilizar qualquer uma das funcionalidades disponibilizadas pelas APIs, os utilizadores devem estar conectados à internet
- RNF003: Deve ser definido um tamanho máximo para mensagens de sessões

6. Outros Requisitos

Segue a lista de requisitos que, embora não estejam considerados no desenvolvimento da aplicação, podem influenciar a correta experiência e utilização da mesma. Para o efeito, listam-se os seguintes requisitos não-funcionais independentes da aplicação:

- O dispositivo cliente deverá ter um conjunto mínimo de especificações de hardware (RAM, processador, etc..) de modo a permitir a utilização sem falhas, visto que certos channels podem ter um fluxo de eventos muito elevado.
- Os servidores onde a aplicação irá ser executada devem ter boas especificações para suportar um maior número de sessões, a aplicação foi testada na instância mais fraca disponível no serviço AWS ECS com somente 0.25 VCpu e 0.5 GB de RAM.

7. Lista de Itens a Elaborar na Fase de Desenho

- Estrutura definitiva de cada classe a implementar no SI.
    - Atributos, respetiva visibilidade e modo de acesso, e exposição para o exterior
    - Métodos, públicos e privados
- Estrutura final da Base de Dados.
- Estrutura, métodos e interfaces das APIs de Administrador e Cliente.

8. Continuidade do Processo de Desenvolvimento

Após a conclusão da análise de requisitos do projeto AppSockets, transposta neste documento de SRS, sugere-se o seguinte itinerário para o desenvolvimento do projeto.

1. Avaliação do SRS e documentos de análise adicionais, nomeadamente a proposta e o cronograma
2. Proposta de Alteração ao SRS e/ou documentos de análise adicionais
3. Reavaliação da proposta na eventualidade de existirem pontos a alterar
4. Acordo para o arranque formal da fase de Desenho, Desenvolvimento e Implementação