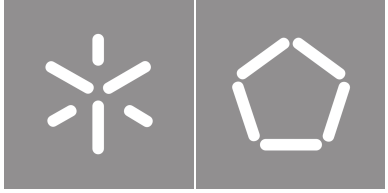
Universidade do Minho
Escola de Engenharia

Pedro Alexandre Gonçalves Ribeiro

Configuração Zero Touch

Outubro de 2022



Universidade do Minho
Escola de Engenharia

Pedro Alexandre Gonçalves Ribeiro

Configuração Zero Touch

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efetuado sob a orientação de:
António Luís Pinto Ferreira Sousa
Vitor Mirones
André Domingos Brízido

Outubro de 2022

Agradecimentos

A dissertação de mestrado é um percurso longo e em que nos deparamos com diversos desafios. Para a superação destes desafios é fundamental todo e qualquer tipo de apoio que possamos receber. Sinto-me feliz por, ao longo deste percurso, ter podido contar com o apoio de todos aqueles que me rodeiam, quer amigos quer familiares.

Agradeço à empresa **Altice Labs** que me acolheu de forma exemplar e a todos os colaboradores que tive a oportunidade de conhecer e que sempre mostraram a maior disponibilidade. Particularizo este agradecimento também ao Departamento de Sistemas de Rede.

Aos orientadores da empresa com quem pude contar: Vitor Mirones e André Domingos Brízido, agradeço o apoio, disponibilidade e motivação frequentes.

Agradeço à empresa **Inova-Ria** pela concessão de uma bolsa de investigação que ajudou imensamente a fazer face às despesas durante o período de desenvolvimento da dissertação.

Por último, agradeço ao meu orientador académico, Doutor António Luis Pinto Ferreira Sousa, por ter aceitado orientar o meu projeto e por assegurar o seu valor académico e científico.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

_____, _____

(Local) (Data)

_____

(Pedro Alexandre Gonçalves Ribeiro)

Resumo

Configuração Zero Touch

A gestão de equipamentos de redes de acesso é um problema complexo e pode tomar dimensões que tornam o tratamento individual de cada equipamento um conceito pouco interessante quer do ponto de vista económico quer do ponto de vista logístico.

Na **Altice Labs** foi desenvolvida uma solução para gerir os equipamentos das redes de acesso: o **AGORA**, um *Network Management System*. Este sistema é modular e escalável, razões pelas quais é constituído por vários componentes que configuram o modelo **FCAPS Model (FCAPS)** da gestão de infraestruturas de rede. Este modelo define as categorias de falha, configuração, responsabilidade, desempenho e segurança. [19]

Neste contexto foi desenvolvido no **AGORA** um módulo de automatização. O módulo *Zero Touch Provision (Zero Touch Provision (ZTP))* foi desenvolvido com o intuito de automatizar o aprovisionamento de *Optical Line Terminal (OLT)'s*. Este módulo faz uso de *templates*, baseados em *playbooks Ansible*, e que são adaptados ao cenário de cada cliente. Este módulo enquadra-se na categoria de configuração do modelo **FCAPS**.

O objetivo foi estender o conceito de automatização até aos *Optical Network Termination (ONT)'s*, que são essencialmente os aparelhos que residem no cliente e terminam a ligação de fibra ótica, sendo que existem ainda modelos mais recentes de *ONT's* que integram já funcionalidades de *routing*.

Deste modo, foi estudado neste trabalho o desenvolvimento de um módulo baseado no *Zero Touch Provision (ZTP)* que suporte a configuração automática das interfaces finais de rede, as *(ONT's)*. A automatização deste processo simplifica instalação de serviços no cliente, uma vez que os equipamentos passam a não necessitar de configurações manuais: tudo o que operador tem de fazer é correr um conjunto de testes que asseguram o correto funcionamento do equipamento uma vez configurado. As instalações passam assim a ser mais rápidas e menos suscetíveis a erros. Para além disto, este processo permite um aprovisionamento dos equipamentos mais rápido, assim como uma maior coerência nas configurações dos diversos equipamentos que constituem a rede.

Este trabalho foi focado num sub-grupo de problemas e tópicos de investigação que tiveram de ser cobertos antes de se avançar para a implementação final do *ZTP Optical Network Unit (ONU)'s*: o desenho de uma arquitetura escalável, o estudo da possibilidade de exploração de concorrência no processamento do aprovisionamento de equipamentos, o desenvolvimento de um método de relacionar um equipamento

com as regras de aprovisionamento disponíveis, o estudo das práticas de segurança a implementar de acordo com o estado da arte assim como o estudo das soluções possíveis para a concretização da monitorização do módulo de *software*.

Palavras-chave: Redes de acesso, Automatização, Ansible, ...

Abstract

Zero Touch Configuration

The managment of network equipments is a complex problem of huge dimensions that makes the indivual handling of each equipment a very unappealing task both from an economic and logistic standpoint.

Altice Labs developed a solution to manage the access network equipments: **AGORA**, which is a *Network Management System*. This system is modular and scalable, as it is made up of various components that make up for the network infrastrutcture managment model **FCAPS**. This model implies the following categories: failure, configuration, accountability, performance and security. [19]

In this context an automation module was developed in **AGORA**. The module *Zero Touch Provision (ZTP)* was developed for the *OLT's* provisioning. This module uses *templates*, based on *Ansible playbooks*, that are adapted for each client scenario. This module embodies the configuration category of the **FCAPS** managment model.

The goal was to extend the automation concept to the *ONT's*, which essentialy are the client devices like, for example, *fiber-gateways* and *routers*.

In short, the development of a new module based on the *Zero Touch Provisioning (ZTP)* that supports the automatic configuration of final network interfaces *(ONT's)* was studied. The automation of this process simplifies the installation and/or configuration process of services in the client, since the equipments will no longer need manual configurations: everything the operator has to do is to run a set of tests to ensure that the equipment is correctly configured. The deployments will then be quicker and less error prone. Furthermore, this process allows for faster provisioning of the network equipments as well as greater coherence among all equipment configurations.

This work was focused in a subgroup of problems and the investigation of topics that had to be covered before advancing for the final implementation of *ZTP ONU's*: the design of a scalable architecture, the study of the possibility of exploring concurrency mechanisms in the processing of equipment provisions, the development of a method of relating an equipment with the available provision rules, the study of security mechanisms according to the state of the art as well as the study of possibilities for accomplishing monitoring in the software module.

Keywords: Access networks, Automation, Ansible, ...

Índice

Índice de Figuras

Índice de Tabelas

Glossário

Deminishing Returns — É um princípio económico que diz que o aumento das unidades de *input* provoca um aumento nas unidades de *output* até que um ponto é atingido em que o aumento no *output* é cada vez menor [18]. 33

FCAPS Model — Este é um acrónimo para os cinco níveis de manutenção de redes considerados no modelo que apresenta o mesmo nome (em inglês): Fault, Configuration, Accounting, Performance e Security. vii, xvii

Multiprotocol Label Switching — É uma tecnologia de encaminhamento de informação que aumenta a velocidade e controlo do fluxo do tráfego de *internet*. Com esta tecnologia a informação é encaminhada tendo em conta determinados *labels* em vez de pesquisas complexas em tabelas de *routing* [37]. xvii, 5

Optical Line Terminal — É o ponto de início de uma *PON*. A função primordial é converter, segmentar, e transmitir sinais para a rede *PON* e coordenar a multiplexação dos terminais óticos de rede para a transmissão partilhada para a *upstream* [8]. vii, xvii

Passive Optical Network — Rede de tecnologia fibra que utiliza uma topologia de tipo ponto-multiponto e entrega informação vinda de um ponto único de transmissão a diversos pontos de utilização [8]. xvii, 4

Plesiochronous Digital Hierarchy É uma tecnologia utilizada em redes de comunicação para transportar grandes quantidades de informação em equipamentos digitais como, por exemplo, fibra ótica e sistemas de ondas rádio [36]. xvii, 5

WebSockets Esta é uma tecnologia que permite estabelecer uma canal de comunicação em ambos os sentidos entre um *browser* e um servidor [1]. 14

Siglas

API Application Programming Interface 17, 18, 66

FCAPS FCAPS Model vii, ix, 1, 5

HTTP Hyper-Text Transfer Protocol 11, 14, 18

MPLS Multiprotocol Label Switching 5

NMS Network Management System 1, 4, 7, 16, 21, 29, 33, 41, 65, 66

OLT Optical Line Terminal vii, ix, xiii, 2, 4, 5, 6, 7, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 28, 29, 31, 32, 33, 35, 36, 37, 39, 40, 41, 42, 48

OMCI ONT Management Control Interface 5, 21

ONT Optical Network Termination vii, ix, xiii, 1, 2, 4, 5, 6, 7, 15, 16, 17, 18, 20, 21

ONU Optical Network Unit vii, ix, 2, 23, 29, 65, 66

PDH Plesiochronous Digital Hierarchy 5

PON Passive Optical Network 4, 5, 6, 42

REST Representational State Transfer 17, 18, 41, 66

RPC Remote Procedure Calls 13

ZTP Zero Touch Provision vii, ix, xi, xii, xiii, 2, 7, 16, 17, 18, 19, 20, 22, 23, 24, 26, 27, 28, 30, 31, 32, 34, 36, 38, 40, 41, 42, 43, 48, 49, 55, 56, 57, 58, 59, 60, 61, 63, 64, 65, 66

1 Introdução

A monitorização de sistemas de redes de acesso é extremamente importante uma vez que permite saber, em tempo real, o estado de cada nó da rede, assim como as suas ligações. Esta tarefa, no entanto, exige uma ferramenta que permita agregar toda esta informação. Esta monitorização é, geralmente, consubstanciada através de *Network Management Systems*. Estes sistemas permitem ter informação sobre equipamentos da rede (*core*, transporte, acesso), como *switches* e *routers*. Por norma, estes sistemas têm acesso a um servidor centralizado de onde retiram informação sobre equipamentos e atualizam as informações existentes. [38]

Um dos primeiros modelos de gestão de serviços foi desenvolvido pela *International Telecommunications Union (ITU-T)* em relação à sua rede de manutenção que divide a gestão do seu funcionamento em cinco áreas fundamentais - falhas, configuração, responsabilização, desempenho e segurança, o modelo *FCAPS* [24].

A *Altice Labs* desenvolveu um *Network Management System (NMS)* cujo nome é *AGORA*. Este sistema tem componentes que configuram cada uma das cinco componentes do modelo *FCAPS* elencadas no parágrafo anterior. Esta dissertação tem por objetivo estudar e desenvolver a arquitetura de um módulo de *software* que permita configurar de forma automática novos equipamentos quando estes são conectados e passam a integrar a rede.

Tipicamente uma rede de acesso é constituída por muitos nós ou equipamentos. No entanto, para que os equipamentos possam ser geridos pelo *NMS* estes têm de ser corretamente configurados. O tipo de equipamento em análise nesta dissertação, o *ONT*, existe numa rede de acesso numa ordem de grandeza que chega facilmente aos milhares, por isso a configuração manual de cada um desses equipamentos é uma tarefa que, se evitada, permite um melhoramento substancial do processo de configuração dos elementos de rede. Assim, se aliarmos a pura dimensão da rede à uniformização e tipificação que se pretende na configuração destes equipamentos percebemos que a automatização do processo de configuração e aprovisionamento é uma tarefa que apresenta bastante valor e é uma oportunidade de otimizar os processos existentes em termos de tempo e probabilidade de ocorrerem erros.

Será realizada uma pesquisa de quais as ferramentas de automatização que podem ajudar a executar as tarefas de configuração relativas a cada *ONT*. Deste modo, a partir do momento em que o *NMS* está

ciente de que um novo equipamento de rede foi conectado pode despoletar um conjunto de ações que têm como objetivo final ter o equipamento completamente configurado e integrado. Importa referir que estas ações são realizadas mediante o contexto do novo equipamento, os dados que o caraterizam. Consequentemente, equipamentos diferentes podem ser configurados de maneira diferente.

1.1 Objetivos

A existência do módulo *ZTP - Zero Touch Provision* de *OLT's* permitiu testemunhar os seus benefícios no aprovisionamento desse tipo de equipamentos. O facto de minimizar os erros ocorridos na configuração destes equipamentos fez do módulo um sucesso e estabeleceu o precedente necessário para que idealizasse um serviço análogo para os *ONT's*. No entanto, estes últimos equipamentos, pelo facto de existirem num número incomparavelmente maior fazem com o que o desenho do módulo *ZTP* para os *ONT's* inclua outro tipo de preocupações nomeadamente ao nível da escalabilidade, razão que justifica a necessidade deste trabalho para perceber quais as linhas que devem orientar o desenho deste novo módulo de *software*.

O principal objetivo deste projeto é estudar a possibilidade de automatização do aprovisionamento e configuração de *ONT's*. Este estudo deve incluir a proposta de uma arquitetura para o módulo de *software* que se propõe a cumprir o objetivo proposto assim como a análise detalhada de todas as vertentes dessa arquitetura em termos de viabilidade e escalabilidade.

No entanto, é útil e produtivo tentar dissecar as sub-tarefas que compõe a tarefa principal, de modo a ter uma lista que orienta o trabalho a realizar:

- Desenho, em traços gerais, de uma arquitetura para o novo serviço que satisfaça todas as suas exigências em termos de funcionalidade.
- Levantamento do estado da arte: relativamente a conceitos arquiteturais, ferramentas para a implementação de microsserviços e ferramentas para a automatização de tarefas.
- Desenvolvimento de algumas alternativas de arquiteturas para o módulo *ZTP ONU's*.
- Análise das propostas de arquitetura apresentas e desenvolvimento de testes que permitam comparar o seu desempenho em cenários representativos dos cenários esperados em produção.
- Análise dos resultados proporcionados pelos testes de desempenho e escolha da arquitetura a recomendar.
- Análise de mais pontos de interesse como, por exemplo, segurança e monitorização.

1.2 Estrutura do documento

No Capítulo 2 são apresentados os fundamentos que sustentam o conhecimento que será aplicado na conceção da solução, sendo que são também elencadas algumas tecnologias e *frameworks* que serão consideradas para a implementação da solução projetada.

Seguidamente, no Capítulo 3, é mostrada uma versão, embora simplificada, daquilo que é a arquitetura do módulo de *software* a desenvolver.

No Capítulo 4 são apresentadas e explicadas algumas alternativas da arquitetura a implementar no módulo do *ZTP ONU's* e explicadas as suas vantagens e desvantagens.

No Capítulo 5 é explicado, com detalhe, um serviço instrumental para o funcionamento do módulo de *software* idealizado, desde quais os requisitos que necessita de cumprir até aos mecanismos encontrados para assegurar esse cumprimento.

No Capítulo 6 é apresentado o resultado de pesquisa realizada no âmbito da segurança em ambiente de microsserviços e, como resultado disso, são apresentadas algumas sugestões para que se possa desenvolver o módulo *ZTP ONU's* tendo já esta preocupação.

No Capítulo 7 é apresentado, mais uma vez, o resultado da pesquisa realizada no âmbito da monitorização em ambiente de microsserviços e apresentada uma proposta de arquitetura por forma a centralizar a recolha de *logs* no *ZTP ONU's*.

Por fim, no Capítulo 8, é feita uma análise do trabalho realizado, os pontos que podem ainda ser desenvolvidos e os pontos que, porventura, foram satisfeitos na totalidade.

2

Estado da arte

A configuração dos equipamentos das redes de acesso é uma tarefa bastante complexa e repetitiva e que, por isso, claramente beneficia da automatização de alguns processos. Esta complexidade advém da diversidade de ações que têm de ser realizadas de modo a que o equipamento seja devidamente gerido e reconhecido pelo *NMS* e para que seja assegurado o seu correto funcionamento. Primeiro, assim que aparece o equipamento deve ser cadastrado no sistema de gestão de redes. Posteriormente devem ser configurados uma diversidade de serviços de rede que asseguram o correto funcionamento do equipamento e o correto fornecimento dos serviços contratados com o operador de telecomunicações.

Esta automatização permite que as empresas que gerem estas redes aloquem menos tempo para a configuração dos equipamentos e permite ainda que estes processos estejam menos suscetíveis a erros, o que é benéfico em diversos níveis. Tendo isto em mente, neste Capítulo são explicados alguns conceitos necessários para o entendimento do objeto de estudo. É ainda feito o levantamento de algumas ferramentas que são utilizadas para o tipo de tarefas que será necessário realizar.

2.1 Redes PON

As redes *Passive Optical Network (PON)* são uma série de tecnologias de banda alargada que oferecem enormes vantagens em cenários de fibra até à casa do cliente. As vantagens incluem arquiteturas de ponto a ponto, capacidade para *streams* triplas de alta qualidade de voz, vídeo e conexões de internet de alta velocidade com um custo relativamente baixo e competitivo. [7]

Existem quatro variações principais das *PON* que podem ser distinguidas em dois grandes grupos. O primeiro grupo diz respeito à arquitetura baseada em *Asynchronous Transfer Mode (ATM)* e inclui: *ATM PON - APON*, *Broadband PON - BPON* e *Gigabit PON - GPON*. O segundo grupo consiste em *Ethernet PON - EPON*. As variações mais populares das PON são *EPON* e *GPON*. [7]

Em termos de componentes físicos, uma rede *PON* é constituída por: *OLT*, *ODN - Optical Distribution Network* e *ONT*.

A *OLT* é um agregador de sinais *Ethernet* que normalmente está localizada num qualquer estabelecimento da empresa fornecedora dos serviços. A *OLT* substitui múltiplos *switch* nos pontos de distribuição.

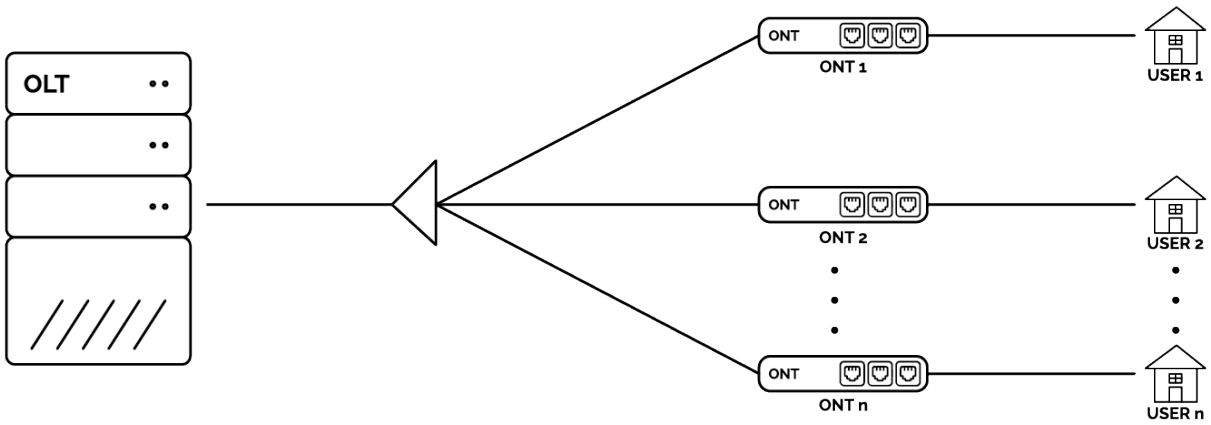


Figura 1: Esquema de uma rede PON a partir da OLT

[28]

Os *ONT* são normalmente colocados muito próximos ou no próprio destino de utilização. Este aparelho converte os sinais óticos *GPON* para sinais elétricos que são entregues aos *routers*. Estes aparelhos têm normalmente várias portas *Ethernet* para conexão a outros aparelhos [28]. Pode ver-se na Figura 1 um esquema simples daquilo que configura uma rede *PON*.

O protocolo de comunicação utilizado entre *OLT* e *ONT* é o protocolo *ONT Management Control Interface (OMCI)* [25].

2.2 AGORA

O **AGORA**[10] é um *Network Management System* que permite controlo de tecnologias como *xPON*, *G.fast*, *Multiprotocol Label Switching (MPLS)* e *Ethernet* e ao mesmo tempo fornece suporte a tecnologias de legado como *Plesiochronous Digital Hierarchy (PDH)*, *TDM*, *ATM*, *SDH* e *xDSL*, minimizando eventuais investimentos em novas tecnologias daqueles que pretenderem utilizá-lo como plataforma de gestão das suas redes de acesso. Este sistema oferece ainda um conjunto de aplicações interativas que visam fornecer ao utilizador todas as ferramentas que necessita para gerir os componentes da sua rede. Este sistema de gestão de redes de acesso fornece todas as funcionalidades contempladas no modelo **FCAPS**.

A aplicação corre em ambiente *Linux*, sendo que não carece de *hardware* específico, utilizando **Java** e fornece ao utilizador uma *interface* de gestão dos elementos de rede.

Para um possível cliente desta aplicação é importante que a aplicação cumpra tudo aquilo a que se propõe e que seja de adaptação fácil às suas necessidades e serviços eventualmente estabelecidos. Por esta razão a plataforma apresenta um conjunto de características que a tornam apetecível: escalabilidade, programabilidade, modularidade e simplicidade [10].

A sua escalabilidade faz com que o sistema apresente desempenho satisfatório independentemente da dimensão da rede do cliente, o que é absolutamente indispensável. A programabilidade é fundamental

para que o cliente consiga estabelecer comunicação entre o *AGORA* e os seus próprios serviços, abrindo assim a possibilidade de criar as suas próprias interações com o serviço. A modularidade facilita a adaptação da aplicação às necessidades de cada cliente. Por último, a simplicidade faz com que, apesar da complexidade que é inevitavelmente inerente a um serviço tão completo, seja impercetível para o utilizador, facilitando a sua utilização e o aproveitamento completo das funcionalidades da aplicação.

Desta forma, conseguimos perceber que o *AGORA* entrega vários benefícios: permite que através de uma única plataforma, que requer pouco em termos de *hardware*, um único utilizador consiga desempenhar várias tarefas relacionadas com a gestão e manutenção de um sistema de redes. Através desta simplicidade e concomitante versatilidade este sistema reduz os custos de operação da rede ao mesmo tempo que aumenta a eficiência na realização das tarefas que lhe dizem respeito. Assim com um baixo custo, os utilizadores conseguem ter uma ideia, em tempo real, do estado da rede, permitindo que se detetem falhas com grande rapidez e se detetem degradações de qualidade de serviço, tornando possível uma postura pró-ativa na manutenção da integridade e qualidade da rede.

2.2.1 Componentes físicos geridos pelo *AGORA*

Uma *OLT* é uma unidade bastante complexa, quer em termos lógicos quer em termos físicos. Este dispositivo é o início de uma rede *PON*. A função primordial destes dispositivos é converter, segmentar e transmitir sinais para a rede *PON* e coordenar a multiplexação dos terminais óticos de rede para a transmissão partilhada para *upstream* [8]. Este aparelho é modular e pode ser munido de várias cartas com várias funções: cartas matriz, cartas *PON*, cartas de *uplink* e cartas de refrigeração. Estes são aparelhos que residem, normalmente, em instalações do fornecedor de serviços ou instalações próprias mais próximas do cliente. A configuração automática deste tipo de dispositivos não faz parte do âmbito deste trabalho, pelo que o foco é agora situado no aprovisionamento automático das *ONT's*, equipamentos que são instalados no cliente. Este dispositivo é responsável pela conversão do sinal da rede *PON* para *Ethernet*, por forma a que o cliente possa usufruir do serviço contratado. Para que este componente funcione corretamente necessita de ser configurado através de regras que podem ser geradas tendo em conta algumas informações relativas ao equipamento. Existem dois cenários possíveis a considerar para a configuração do equipamento: o equipamento não foi pré-configurado no *AGORA* pelo que a sua configuração tem de ser feita integralmente, ou o equipamento foi pré-configurado no *AGORA*, cenário em que a sua conexão física é suficiente para garantir a sua operabilidade. No primeiro caso, quando a *ONT* é fisicamente conectada, emite alarmes que indicam a sua presença e são acompanhados das informações que são essenciais para a configuração do equipamento.

2.3 Zero Touch Provision

A instalação de novos equipamentos numa rede é algo extremamente comum e como tal é algo que rapidamente salta á vista como um alvo apetecível de automatização. Desta forma, o *Zero Touch Provision*

é o modulo do *AGORA* que se ocupa da configuração e aprovisionamento automático dos equipamentos que integram as redes de acesso.

A rede de acesso, no entanto, é constituída por vários tipos de equipamentos: *OLT's*, *ONT's*, *switches* e mais. O *Zero Touch Provision* suporta, para já, a configuração e o aprovisionamento de *OLT's*.

Este processo de configuração automática destes dispositivos tem, essencialmente, três fases: descoberta, atualização de *firmware* e aprovisionamento. O processo é despoletado pela própria *OLT* que, uma vez conectada, emite uma notificação que indica que entrou na rede, no caso de a instalação de uma nova unidade deste tipo. Uma *OLT* é, no entanto, uma unidade altamente modular em que podem ser instaladas cartas de vários tipos que necessitam de ser configuradas. Este tipo de configuração é também suportada pelo *ZTP* e acontece quando a *OLT* informa o *NMS* que uma nova carta foi conectada.

Estes alarmes são depois tratados por este módulo. No caso do alarme que alerta para a entrada de um novo *OLT* é despoletado um conjunto de ações que correm impreterivelmente na seguinte ordem: descoberta, atualização de *firmware* e aprovisionamento. A fase de descoberta verifica se o equipamento é já conhecido pelo sistema e se não for procede ao cadastro do equipamento no *NMS* e às configurações associadas a esse registo. Seguidamente é verificado se o equipamento necessita que o *firmware* seja atualizado de modo a cumprir com todos os requisitos de operabilidade. Por último é corrido o aprovisionamento que corresponde à aplicação das configurações geradas para o equipamento tendo em conta os parâmetros fornecidos.

Existe um mecanismo que faz com que os alarmes sejam reenviados periodicamente até que a ação que eles despoletam tenha sido completada com sucesso. No caso de a configuração ser totalmente aplicada o alarme é desativado pelo que não se repete. Por outro lado, se o processo não for concluído com sucesso esse mesmo mecanismo assegura que o processo de configuração é repetido.

2.4 Ferramentas de automatização

Uma rede *PON* é normalmente constituída por imensos *ONT's*. No tipo de redes em análise cada *OLT* pode ter conectadas até mais de 65000 *ONU's*. Ora, estes equipamentos têm de ser corretamente configurados para que possam desempenhar a sua função na rede.

Atualmente, no momento de instalação, estes equipamentos têm de ser configurados por operadores, o que constitui um passo subótimo no processo de configuração dos elementos da rede. O facto de cada equipamento ser manualmente e individualmente configurado torna o processo suscetível a erros e mais dispendioso em termos de tempo, ou seja, os operadores demoram mais tempo na instalação do que demorariam caso existisse um processo de automatização que assegurasse a aplicação de todas as configurações necessárias, de acordo com o contexto do equipamento, assim que o mesmo se conecta com a rede.

O processo de configuração dos equipamentos é repetitivo e realizado em diversos equipamentos pelo que se torna um alvo preferencial de automatização.

Para além disso, as configurações destes equipamentos são algo que pode sofrer alterações ao longo do tempo pelo que uma ferramenta que permita gerir versões das configurações pode ser extremamente útil.

Com efeito, faz todo o sentido investigar ferramentas que possam assegurar a configuração automática dos equipamentos e versionamento das mesmas por forma a poder otimizar-se este passo da configuração dos equipamentos de rede.

As ferramentas de gestão de configurações tornam possível a utilização de práticas de desenvolvimento de software consagradas para a manutenção e aprovisionamento de estruturas computacionais através da definição de ficheiros de texto [34].

Por todas estas razões procedeu-se ao levantamento das ferramentas que poderiam desempenhar estas tarefas, sendo estas explicadas e analisadas ao longo das seguintes secções.

2.4.1 CFEngine

Esta é uma das ferramentas mais antigas que fornece automatização de configurações. O *CFEngine* fornece a possibilidade de automatizar infraestruturas computacionais, ou seja, permite realizar alterações ao nível da infraestrutura e fazer com que essas alterações se verifiquem de forma muito rápida em todas as máquinas que a constituem. O facto de se manterem as configurações numa única fonte de verdade faz com que a infraestrutura se mantenha coerente e minimize, por isso, *"drifts"* de configurações.

O facto de ser baseado em *C*, uma das linguagens mais *low-level*, faz com que esta ferramenta seja inerentemente rápida e escalável para um número muito elevado de nós [34].

2.4.2 Rudder

Rudder é uma solução de fácil utilização, pensada para *web* e baseada em funções para a automatização de estruturas computacionais assim como a sua conformidade com regras previamente estabelecidas. O seu foco é monitorizar continuamente as configurações e centralizar a informação resultante desse processo de modo a que o gestor da plataforma tenha acesso a ela [2].

A ferramenta permite estender regras que já contém para que o utilizador consiga implementar padrões de configuração específicos. A ferramenta possui um ambiente gráfico, opcional, que permite realizar estas tarefas tornando-a bastante apetecível para utilizadores menos versados neste tipo de tecnologias. O facto de a ferramenta ser escrita em *C* permite que corra em praticamente qualquer tipo de dispositivo permitindo ao utilizador gerir uma variada gama de equipamentos.

A equipa de desenvolvimento da ferramenta acredita que existe um distanciamento crescente entre o tempo de desenvolvimento e *deployment* da aplicação e a maior duração da infraestrutura. Neste contexto, o *Rudder* foi desenvolvido para monitorizar e garantir o funcionamento da infraestrutura, permitindo perceber qual a origem de possíveis problemas e quais os nós cujas configurações se encontram desviantes. Com isto percebemos que esta ferramenta apesar de poder ser utilizada para forçar configurações

nos *hosts*, o seu principal foco é garantir que essas configurações se mantêm ao longo do tempo.

2.4.3 Ansible

É uma ferramenta *open source* que permite a automatização do aprovisionamento, gestão de configurações, instalação de aplicações e orquestração de processos em geral numa máquina, quer localmente quer esta seja acessível por *ssh*. A ferramenta utiliza o protocolo de *ssh* para executar as tarefas pretendidas na máquina destino [30].

O *Ansible* é uma ferramenta que não carece de um agente na máquina onde se pretende que as tarefas executem, ou seja, não requer a instalação de *software* específico, sendo por isso especialmente atrativo [30].

Existe um conjunto de conceitos que são elementares para a utilização de todas as potencialidades desta ferramentas: inventário, *playbook*, *play* e *role*. O inventário corresponde, essencialmente, ao conjunto de endereços das máquinas onde devem ser executados os *playbooks*. Um *playbook* é um ficheiro em formato *yaml* onde se definem o conjunto de tarefas a executar. A tarefa é a unidade mais granular de um *playbook*, sendo por isso vulgarmente denominada por *play*. Por último, um conceito igualmente importante é o *role* que permite agrupar tarefas que têm por fim concretizar um determinado objetivo.

2.4.4 Puppet

O *Puppet* é uma ferramenta que acaba por servir diversos cenários de utilização. Pode ser utilizada para: operação e instalação de aplicações, automatização de manutenção de configurações, validação contínua das configurações e mais... Obviamente, o ponto com mais interesse para o estudo em questão é a automatização da implementação e manutenção de configurações. Esta aplicação permite ainda que se definam e forcem configurações ao longo de uma série de sistemas operativos, *middleware* e aplicações de um modo programático [3].

Por outro lado, esta é uma ferramenta que tem uma curva de aprendizagem maior uma vez que utiliza uma linguagem declarativa própria. O estado desejável do sistema é descrito em ficheiros, nesta linguagem, que são chamados de *manifests*. Estes ficheiros descrevem como os recursos de rede e do sistema operativo, como, por exemplo, ficheiros, *packages* e serviços devem ser configurados. O *Puppet* compila os *manifest's* para *catalog's* e aplica-os em cada um dos nós para garantir que cada um deles está devidamente configurado [4].

2.4.5 Juju

O *Juju* [27], desenvolvido pela Canonical, é uma ferramenta bastante sofisticada que visa ajudar no processo de instalação, configuração e manutenção de infraestruturas de computação bastante complexas.

No entanto, a filosofia da ferramenta é radicalmente diferente daquelas que foram enumeradas anteriormente. A ferramenta foi desenvolvida tendo em mente o reaproveitamento de configurações desenvolvidas por peritos em cada ferramenta a ser instalada.

Este é um sistema para ajudar a mover da gestão de configurações para a gestão de aplicações em nuvem - através de aplicações partilháveis e reutilizáveis que são chamadas de *Charmed Operators* [9]. Isto permite que em vez de, para cada caso de uso, se escreverem e manterem *scripts* de *deployment* podem simplesmente ser compostos *charms* que englobam, em si, toda essa lógica. Desta forma, a equipa do projeto pode focar-se mais no desenvolvimento da lógica de negócio em vez de gastar tempo no controlo de *deployment* da infraestrutura computacional.

Pacotes de código de operações reutilizáveis são mais escrutinados o que ajuda a que problemas de segurança sejam encontrados e resolvidos com maior celeridade. Atraem um maior número de colaboradores com um conjunto de habilidades maior. Por outro lado, como a intervenção do utilizador no processo de *deployment* é menor, este é menos sujeito a erros manuais e que, acontecendo, são mais fáceis de serem detetados [9].

2.4.6 Comparação entre ferramentas de automatização

Para uma comparação rápida entre as ferramentas elencadas é útil ter acesso a um quadro com as careterísticas de cada uma delas relativamente a aspetos fundamentais no que diz respeito à interação e utilização das mesmas. Uma caraterística extremamente importante é a linguagem dos ficheiros de configuração ou da lógica a definir, razão pela qual esta caraterística foi incluída no quadro. Uma outra caraterística que é importante é a necessidade de instalar um agente na máquina que se pretende configurar, o que, como podemos ver, não acontece para todas as opções. Por último, podemos ver que todos utilizam o protocolo de *SSH* para a comunicação entre a máquina que emprega as configurações e a máquina em que as mesmas se refletem.

	Linguagem	**Agente**	**SSH**
CFEngine	YAML	Yes	Yes
Rudder	GUI	Yes	Yes
Ansible	YAML	No	Yes
Puppet	Puppet	Yes	Yes
Juju	YAML	Yes	Yes

Tabela 1: Comparação entre ferramentas de automatização

No caso do *AGORA* a existência de módulos desenvolvidos para a comunicação com o mesmo foi também um motivo que teve peso na escolha de qual das ferramentas elencadas utilizar. Deste modo, a programabilidade e extensibilidade foram também fatores considerados.

2.5 Arquiteturas aplicacionais

As arquiteturas monolíticas são, provavelmente, o tipo de arquitetura mais comum e consistem numa aplicação com uma única base de código que inclui múltiplos serviços [16]. Este tipo de arquitetura consiste em acoplar todos os componentes de uma aplicação, o que faz com toda a sua funcionalidade seja colocada no mesmo processo da máquina.

As arquiteturas de microsserviços surgiram como resposta às seguintes questões relativas ao que aconteceria no caso de um serviço monolítico falhar - Que implicações isso teria na camada de negócio? Como é que se pode fazer com que em caso de falha algumas componentes do negócio continuem a funcionar e se possa continuar a assegurar serviço.

Para perceber a arquitetura de microsserviços é importante entender primeiro o conceito de microsserviço e as ideias e princípios que estão normalmente associados. Não existe uma definição oficial de microsserviço, no entanto, os microsserviços podem ser entendidos como unidades pequenas, completamente contidas em si mesmas, leves e capazes de desempenhar um ou mais serviços. Segundo Martin Fowler, o estilo arquitetural de microsserviços é uma abordagem para o desenvolvimento de uma aplicação como um conjunto de pequenos serviços, cada um correndo os seus processos e que se comunicam através de mecanismos leves, como por exemplo *Hyper-Text Transfer Protocol (HTTP)* para consumir uma *API* [21].

Conseguimos perceber o quão útil e eficiente esta arquitetura é estabelecendo uma comparação com a arquitetura mais tradicional, a arquitetura monolítica. No caso de se querer realizar algum tipo de alteração em alguma parte da lógica/componente da aplicação enfrentamos processos bastante distintos em cada uma das arquiteturas: na arquitetura tradicional teríamos de atualizar a aplicação, parando a sua execução e recomeçando, o que provocaria períodos em que o serviço não estaria disponível. Pelo contrário, numa arquitetura de microsserviços, simplesmente teríamos de atualizar as instâncias do microsserviço que seriam afetadas pela alteração, tornando assim o processo de atualização muito mais localizado e preciso [23].

Para além disso, numa arquitetura baseada em microsserviços podemos implementar heterogeneidade tecnológica, isto é, componentes assentes em diferentes tecnologias desde que concordem num mecanismo de comunicação. Uma outra vantagem deste tipo de arquitetura é que uma falha numa máquina ou *container* que aloja um serviço não inutiliza necessariamente a aplicação como um todo ao passo que isso pode acontecer numa arquitetura monolítica [16].

Um serviço mal construído numa arquitetura monolítica, na eventualidade de encontrar problemas normalmente inviabiliza a totalidade da aplicação, pondendo até causar a indisponibilidade da mesma. Num cenário de microsserviços apenas o microsserviço comprometido é afetado pelo que fica limitado o impacto da falha, não afetando necessariamente a totalidade da aplicação [29]. No entanto, a desagregação dos serviços exige maior rigor na gestão do versionamento dos protocolos de comunicação entre serviços, i.e, a liberdade de atualizar os serviços individualmente obriga ao cuidado de garantir que se mantém retro-compatíveis e "à prova de futuro"na comunicação com outros serviços.

Um ponto fulcral é também a maneira como podemos escalar a aplicação e que toma contornos distintos em ambas as arquiteturas. Na arquitetura tradicional não temos outra alternativa senão lançar novos processos da aplicação por novos servidores, o que nem sempre é possível e acaba por recorrer-se a um escalamento vertical que corresponde ao aumento de recursos nos servidores já existentes. Em microsserviços podemos escalar distribuindo os serviços por servidores distintos e replicando-os tanto quanto necessário e atribuindo recursos da máquina necessários a cada uma das instâncias [29]. Noutras palavras, na arquitetura monolítica não conseguimos escalar componentes de forma independente, conforme as necessidade de cada componente aplicacional.

2.6 *Frameworks* para microsserviços

Convém referir que não é estritamente necessária a utilização de *frameworks* para o desenvolvimento de *software* tendo em conta a arquitetura de microsserviços. É perfeitamente possível desenvolver um microsserviço sem recurso a tecnologias fornecidas por estas *frameworks*, no entanto, fornecem implementações e abstrações de funcionalidades o que permite que nos foquemos mais no desenvolvimento da lógica de negócio, delegando a implementação de camadas mais tecnológicas a esses serviços.

O trabalho de pesquisa realizado revelou que existem diversas alternativas no que toca a *frameworks* pelo que é da maior importância perceber quais as alternativas que mais se ajustam ao trabalho em mãos.

2.6.1 Spring Boot

Esta é a *framework* mais popular para o desenvolvimento de microsserviços em *Java*. Existem várias vantagens que advêm da utilização desta ferramenta: o facto de ser inerentemente *multi-threaded* faz com que estejamos relativamente protegidos de casos de pedidos que impliquem uma resposta mais demorada bloquearem o microsserviço, uma vez que novos pedidos podem ser atendidos por uma nova *thread*, isto se estivermos a falar de um ambiente completamente síncrono; existem diversos outros componentes como o *Spring Data* e *Spring Cloud* que facilitam a interação do serviço com bases de dados e outros microsserviços instalados na *cloud*, respetivamente; contém, opcionalmente, servidores aplicacionais já embebidos, o que facilita imenso a instalação do microsserviço; oferece inversão de controlo assim como injeção de dependências. [39]

2.6.2 Dropwizard

Dropwizard está tão próximo de uma *framework* como de uma biblioteca. O seu objetivo é fornecer implementações com bom desempenho e confiáveis de tudo o que uma aplicação *web* necessita num ambiente de produção. Como estas funcionalidades são extraídas para uma biblioteca reutilizável a

aplicação, em si, permanece simples e objetiva reduzindo o tempo necessário para a lançar no mercado e os recursos necessários para mantê-la. [22].

2.6.3 Go Micro

Esta é uma *framework* destinada ao desenvolvimento de sistemas distribuídos. A *Go Micro* fornece os componentes principais para o desenvolvimento deste tipo de sistemas incluindo Remote Procedure Calls (RPC) e comunicação orientada aos eventos [5].

O que a *Go Micro* oferece é um conjunto de serviços, por omissão, para cada uma das diferentes componentes de sistemas distribuídos, sendo que existem muitas outras alternativas para cada componente que podem ser consideradas, dependendo do ambiente pretendido para o lançamento da aplicação. A ferramenta fornece código *boilerplate* e funcionalidades necessárias para cada uma das componentes deste tipo de serviços em larga escala.

A *framework* em questão fornece vários serviços incluindo: autenticação, fornece uma interface para a comunicação com bases de dados, descoberta de serviços - registo automático de serviços e resolução de nomes, codificação de mensagens dinâmica de acordo com o seu conteúdo, sistema assíncrono de mensagens e muitos outros serviços [5].

2.6.4 Moleculer

Esta *framework* corre em *Node.js* sendo, por isso, destinada a programadores que queiram utilizar *Javascript*. É também completa e bastante fácil de utilizar. A ferramenta oferece diversas valências: ser rápida, ou seja, apresentar latências baixas; é *open source* o que faz com que erros e eventuais falhas possam ser identificadas e corrigidas com maior rapidez uma vez que o projeto tem o apoio da comunidade; é extensível, o que quer dizer que todos os módulos incluídos podem ser substituídos e o programador pode, inclusivamente, fazer alterações a esses módulos para que sirvam melhor os seus propósitos; por último, é tolerante a falhas, uma vez que conta com um balanceador de carga e mecanismos que refazem os pedidos quando estes não são bem sucedidos [6].

A comunidade que utiliza esta ferramenta é bastante grande e, mais uma vez, o facto de o projeto ser *open source* faz com que possa progredir e evoluir de maneira mais rápida e suportada.

Existem, no entanto, alguns conceitos chave que são parte integrante do modelo de funcionamento da *framework*. Um *service* é um módulo que contém parte da aplicação, é isolado e contido o que significa que case falhe não causa falhas noutros serviços. Um *node* é uma unidade na rede que corre *services*. *Local services* são serviços que correm numa mesma unidade de rede e partilham, por isso, *hardware*. *Remote services* são serviços que correm num *node* externo ao serviço que constitui o ponto de vista. O *service broker* é um serviço que corre em cada nó da rede, que coordena o serviço e gere a comunicação com outros serviços. O *transporter* é o mecanismo através do qual os serviços se comunicam, ou seja, que garante o trânsito das mensagens. Por último, o *gateway* é um serviço *Moleculer* como qualquer

	Spring Boot	**Dropwizard**	**Go Micro**	**Moleculer**
Linguagem	Java	Java	Golang	Javascript
HTTP	Tomcat (default) Jetty Undertow	Jetty	go-micro	got
REST	Spring(default) JAX-RS	Jersey	go-micro	moleculer-web
Métricas	Spring	Dropwizard Metrics	go-micro	moleculer-console-tracer moleculer-jaeger moleculer-prometheus moleculer-zipkin
Registos	Logback Log4j Log4j2 slf4j Apache common-logging	Logback slf4j	go-micro	logger built-in Pino Bunyan Winston

Tabela 2: Comparação entre *frameworks* para a implementação de microsserviços

outro que corre HTTP ou WebSockets e que recebe pedidos do exterior e disponibiliza os serviços da aplicação para o exterior e providencia ainda as respostas para os pedidos recebidos.

2.6.5 Comparação entre *frameworks*

Mais uma vez, é útil uma comparação simplificada onde possamos ver com clareza quais as implementações que sustentam vertentes fulcrais dos microsserviços que as *frameworks* implementam.

Um fator extremamente importante na escolha da *framework* a utilizar será a linguagem de programação que a suporta uma vez que, dependo do tempo disponível para a implementação para a solução, este pode ser fator extremamente importante numa filtragem inicial. Outros fatores como a tecnologia que suporta os serviços de comunicação *HTTP* e *REST* são extremamente relevantes, pelo que são incluídos neste quadro.

Muitas *frameworks* disponibilizam também serviços de métricas que permitem perceber o volume de comunicação e o estado dos recursos da máquina que suporta o serviço. Como todas as elencadas permitem, com recurso a uma ou outra tecnologia, consultar as métricas considerou-se relevante incluir este parâmetro de comparação no quadro.

Numa arquitetura baseada em microsserviços os serviços de *logging* têm o objetivo de garantir o princípio da responsabilização e ajudar a detetar anomalias no seu funcionamento. Desta forma, é fundamental para quem concebe a arquitetura de segurança de uma aplicação utilizar e compreender padrões existentes por forma a implementar os *logs* neste tipo de sistemas [14]. Esta importância faz com que as tecnologias disponíveis em cada *framework* para a implementação deste serviço sejam também elencadas neste quadro.

Com a Tabela 2 reparou-se que a maior parte das *frameworks* apresentadas são bastante versáteis oferencedo mais que uma alternativa para cada uma das componentes necessárias de um microsserviço. No entanto, a linguagem de programação em que assentam é extremamente importante quer por razões de desempenho quer pelo facto de o projeto se integrar numa empresa que tem uma conjunto de pessoas que são qualificados numa linguagem de programação. Conforme veremos em 3.2 este fator teve um grande peso na escolha de uma destas *frameworks* para a realização deste trabalho.

2.7 Escalabilidade

A escalabilidade da aplicação a desenvolver é obviamente um aspeto de grande importância tendo em conta o tipo de carga que é expectável, sendo que devemos considerar redes de acesso que têm imensos *ONT's* a serem continuamente conectados. Estando mais ou menos evidente que arquitetura a ser adotada na conceção da solução é a de microsserviços é prudente pesquisar quais são os padrões mais utilizados e testados para escalar a capacidade da aplicação.

Antes de entrar em detalhe sobre quais os padrões de escalabilidade a adotar é preciso definir quais as situações que ditam que deve haver um aumento de capacidade de processamento de pedidos. É útil definir um período de tempo de processamento de cada pedido que se considere aceitável e, por exemplo, a partir de um determinado período tempo em que o tempo de resposta é excedido deve acontecer um aumento de capacidade de processamento de pedidos.

O aumento da capacidade de processamento pode acontecer através de escalabilidade vertical ou horizontal. A escalabilidade vertical corresponde a adicionar mais recursos a um serviço existente [33], o que no nosso caso corresponde a utilizar mais *threads* capazes de atender pedidos ou aumentar os recursos de *CPU* e memória alocados ao *container* ou máquina virtual. A escalabilidade horizontal corresponde, essencialmente, a lançar mais instâncias do serviço em diferentes máquinas [33], o que tendo em conta que falamos de um microsserviço não representa, de todo, uma dificuldade. Este último confere, caso seja devidamente configurado, *high availability* e *fault tolerance* ao serviço a desenvolver uma vez que no caso de um dos serviços falhar o pedido pode ser atendido por uma das instâncias redundantes.

3

Protótipo da arquitetura

O objetivo principal desta dissertação é automatizar o processo de aprovisionamento de um novo *ONT* no sistema de gestão de redes, assim como forçar a aplicação das configurações adequadas no equipamento para que ele possa desempenhar a sua função na rede.

3.1 Elementos arquiteturais

Para que consigamos desenhar uma arquitetura que faça sentido e realmente represente uma possível solução para o problema em questão é útil percebemos quais os elementos que farão parte da mesma. Estes elementos têm que se comunicar e coordenar de modo a assegurar a utilidade e correção do processo. Sendo assim, resta elencar quais os componentes que farão parte da proposta de arquitetura da solução a desenvolver.

Em primeira instância temos, claramente, o *ONT*, uma vez que é a sua entrada na rede que despoleta todo o processo. Quando isto acontece a *OLT* deteta a sua presença e envia uma mensagem ao sistema de gestão dos dispositivos de rede.

Como é óbvio esta mensagem tem um destino, onde será processada e desencadeará um conjunto de processos que culminam com o correto aprovisionamento do equipamento. No *NMS* utilizado para suportar este trabalho, o *AGORA*, existe um componente específico encarregue de processar alarmes (incluindo os alarmes gerados pela descoberta dos novos *ONT*'s). Os alarmes relevantes são encaminhados para o módulo *ZTP*.

O aprovisionamento dos equipamentos referidos é um processo relativamente complexo. Com a finalidade de o tornar tão configurável quanto possível escolhemos utilizar *templates* de *Ansible* que conta com *templates* em texto facilmente editáveis e o facto de o formato destes ser *yaml* faz com que colaboradores não tão familiarizados com a programação consigam contribuir, interpretar e interagir com os mesmos. No entanto, constatámos que um *template* não seria suficiente para configurar e aprovisionar todo e qualquer *ONT*, uma vez que se podem querer realizar configurações arbitrariamente diferentes com base numa diversidade de critérios: *OLT* a que o *ONT* está conectado, em que porta, qual o *service provider* que suporta o equipamento. Com base nesta necessidade surgiram as regras do *Zero Touch*

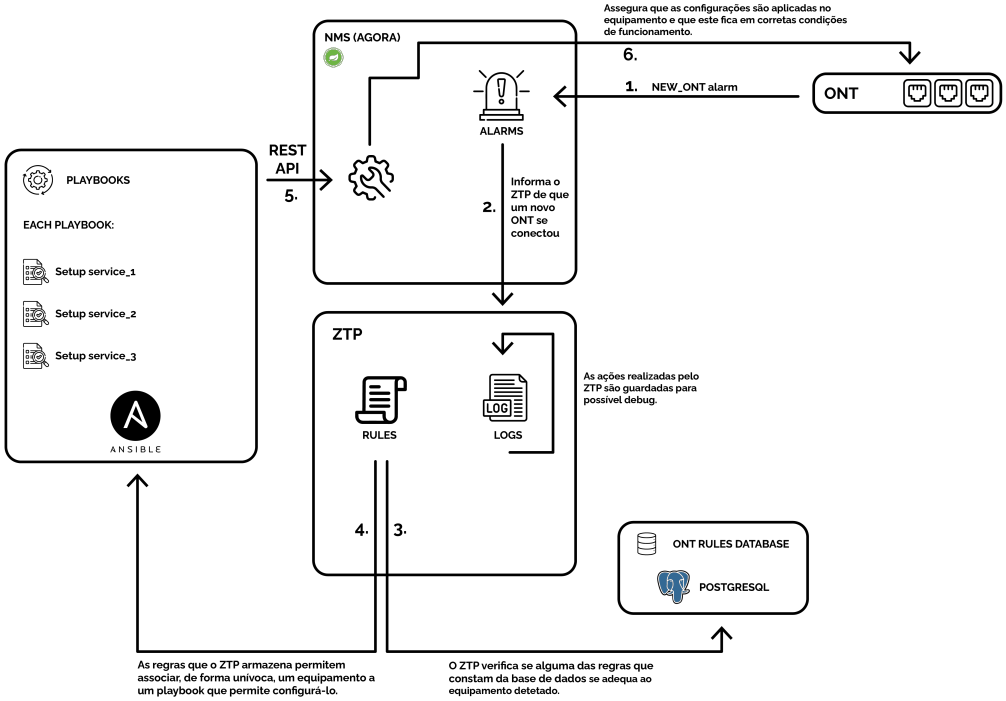


Figura 2: Arquitetura da solução a conceber

Provision: estas regras são, essencialmente, filtros que se alimentam das informações fornecidas pela *ONT* no alarme que indica que esta se conectou e a canalizam para um dos *templates*.

As regras são constituídas por um conjunto de campos opcionais, tais como: *OLT*, *card*, *interface*, *equipmentId*, *firmwareVersion*, *serialNumber*, *password* e *vendor*. Esta configuração permite ter flexibilidade quanto às combinações de parâmetros que se podem escolher. Podemos, por exemplo, determinar que *ONT*'s com determinada versão de *firmware* que são simultaneamente de um determinado *vendor* devem ser configuradas utilizando um *template*, entre muitas outras combinações. No entanto, é fornecido mais contexto e mais detalhes relativamente a este conceito de regras no Capítulo 5.

Desta forma conseguimos perceber quais os principais elementos que fazem parte da solução a conceber.

Conforme mencionado, diferentes operadores de rede podem pretender que os seus *ONT's* sejam configurados de determinadas maneiras pelo que existe a liberdade de eles desenvolverem os seus próprios *playbooks*, sendo que deve ficar claro na regra do *ZTP* qual o *template* a ser utilizado no aprovisionamento.

Para que seja possível criar regras do tipo mencionado, o *ZTP* disponibiliza uma série de *endpoints* de uma *Representational State Transfer (REST) Application Programming Interface (API)* que permitem criar e manipular regras de modo a que se consiga mapear qualquer *ONT* que possa vir a ser descoberto numa regra.

3.2 Tecnologias escolhidas

O trabalho a desenvolver é constituído por duas partes essenciais: o serviço do *ZTP* que permite criar as referidas regras e a componente que diz respeito à criação do serviço de rede necessário ao correto funcionamento da *ONT*.

Na Secção 2.6 foram elencadas algumas ferramentas e *frameworks* que auxiliam na implementação microsserviços pelo facto de facilitarem o foco na lógica de negócio. Uma destas ferramentas será utilizada para implementar o serviço *ZTP*, ou seja, a componente que nos permite criar as regras que permitem associar equipamentos a *playbooks* e processar as mensagens que indicam a presença de novos equipamentos. Por outro lado é necessário também escolher uma ferramenta para suportar a automatização da criação do serviço de rede para cada *ONT* de modo a garantir as configurações necessárias ao seu funcionamento.

Para a implementação do primeiro serviço a opção escolhida foi *Spring Boot*, por algumas razões: o facto de esta ser uma *framework* muito utilizada a nível da indústria e comunidade teve um peso grande na decisão, uma vez que facilita bastante a procura de suporte quando dúvidas e problemas surgem. Para além disso, é preciso relembrar que esta dissertação decorre num contexto empresarial, e que a *framework* utilizada para este tipo de casos de uso é esta mesmo. Portanto, para que o projeto possa ser mantido finda esta dissertação é importante que a ferramenta que o suporta seja uma com a qual os colaboradores da empresa estejam confortáveis e sejam capazes de manter e evoluir.

A ferramenta escolhida para suportar a criação dos serviços de rede foi o *Ansible*. A diversidade de módulos disponíveis para os mais diversos fins, assim como a possibilidade de desenvolvimento dos nossos próprios módulos faz com que a ferramenta seja bastante apelativa. No contexto empresarial esta ferramenta foi já utilizada na automatização da configuração das *OLT's* e foi também já desenvolvido um módulo para comunicação com a *REST API* do *AGORA*, o que é mais um fator que fez com que a decisão final fosse a de utilização desta ferramenta. Para além disso, a configuração dos serviços de rede assenta essencialmente em pedidos *HTTP* a uma *REST API*. O facto de a ferramenta suportar nativamente estes pedidos aliado à possibildiade de criar *playbooks* que são depois alimentados pelos dados relativos a cada *ONT* foi mais um argumento que pesou nesta escolha.

4

Gestão de filas de espera no *ZTP*

Conforme já foi dado a entender nos Capítulos anteriores desta dissertação o módulo de *software* a desenvolver integra-se com vários outros já existentes e como tal é necessário garantir essa vertente e estudar como essa integração será atingida. Deste modo, torna-se obrigatório compreender se existem já limitações conhecidas desses módulos que devem ser contempladas e porventura colmatadas nesta nova vertente do *ZTP*.

De acordo com a pesquisa realizada relativamente às vantagens e desvantagens apresentadas por arquiteturas baseadas em microsserviços no Capítulo 2 conclui-se que esta nova implementação deve assentar também numa arquitetura baseada em microsserviços. As razões que sustentam esta escolha são, essencialmente, o facto de cada componente ser indepente quer em termos de funcionamento e implementação quer em termos de escalabilidade. Como cada componente da arquitetura tem funções bem definidas e é autónomo parece imediata a tomada de decisão relativamente à arquitetura aplicacional. A escolha deste tipo de arquitetura tem também em conta a evolução futura facilitada de cada componente da arquitetura apresentada. A título de exemplo, consideremos que a determinada altura se conclui que um dos componentes seria mais fácil de manter e apresentaria melhor desempenho numa versão diferente da *framework Spring Boot*: numa arquitetura de microsserviços esta evolução não apresenta nenhum tipo de problemas, uma vez que o componente pode ser atualizado independentemente, não causando nenhum inconveniente na comunicação com outros componentes nem no funcionamento do próprio. Numa arquitetura monolítica, sendo que é bastante comum algumas funcionalidades ficarem comprometidas com alterações da versão da *framework*, uma alteração da versão da mesma poderia, com relativa facilidade, comprometer o correto funcionamento de outros componentes. Lembremos também que neste caso a falha de um componente implica, inevitavelmente, a falha do módulo de *software* como um todo.

De forma resumida, com a escolha da arquitetura de microsserviços prende-se uma maior resiliência da arquitetura, maior desacoplamento tecnológico entre os componentes e, por isso, maior independência entre os mesmos.

Como consequência da adoção deste tipo de arquitetura para a solução, a natureza da comunicação entre os componentes é evidentemente assíncrona e como tal é necessário um tipo de estrutura que

armazene as mensagens que chegam a cada componente. O tipo de estrutura adotada é a fila do tipo *FIFO*, ou seja, um tipo de fila em que a ordem de tratamento de mensagens é a mesma da ordem de chegada das mesmas.

O módulo do *ZTP* consiste essencialmente num processador de eventos relativos a equipamentos do tipo *ONT* que fazem parte da rede de acesso ou entram na mesma. Neste estudo, pelo menos numa fase inicial, apenas os pedidos relativos ao aprovisionamento de um novo *OLT* são considerados uma vez que o foco é o desenvolvimento de uma arquitetura otimizada para a gestão das filas de mensagens que chegam ao módulo. Tendo isto em mente, no momento inicial do desenvolvimento, o foco foi desenvolver propostas de arquitetura que preconizam um modo de comunicação e gestão de filas entre os diferentes componentes do módulo *ZTP*.

Por forma a orientar o desenvolvimento das diferentes propostas de arquitetura é importante definir as métricas que se pretende otimizar para que seja possível comparar as alternativas de um modo objetivo e perceber qual a que permite alcançar os objetivos propostos. No caso do *ZTP* o objetivo é maximizar o número de *ONT's* bem aprovisionados no menor tempo possível. No entanto, a utilização da concorrência pode despoletar a ocorrência de *timeouts*, o que justifica a implementação de mecanismos de gestão das filas de espera.

4.1 Objetivos e especificidades do ambiente de simulação

Para que consigamos simular uma determinada arquitetura para o módulo de *software* em estudo é importante que definamos bem quais os serviços que a compõem bem como a função de cada um desses serviços. Sendo assim podemos imaginar um primeiro serviço a que atribuímos o nome *worker* cujas funções são de realizar as funções que dizem respeito ao estrito processo de aprovisionamento, isto é, atualizações, registos de equipamentos e execuções de *playbooks*. O nosso estudo deve também contemplar o tempo que as *OLT's* levam a processar os pedidos que lhes chegam pelo se simula também um serviço chamado *olt*. Por último, algumas das propostas de arquitetura que se apresentam na Secção 4.2 contemplam um serviço chamado *broker* que serve para realizar o encaminhamento de mensagens perante determinados critérios.

Numa abordagem mais simples os tipos de processos que integram a solução a desenvolver serão os *workers*, que são as *threads* que o módulo disponibiliza para o atendimento de pedidos. Estas *threads* são responsáveis por realizar as tarefas que os pedidos de aprovisionamento implicam, ou seja, comunicação com outros módulos do *AGORA* e/ou execução de *playbooks*.

Conforme referido, a comunicação que o *ZTP* realiza com outros serviços deve ser alvo de uma análise pormenorizada de modo a evitar de modo preemptivo possíveis pontos de estrangulamento de desempenho que têm origem na forma como esses mesmos serviços funcionam. Ou seja, o que se pretende é arquiteturar este componente de modo a que este consiga contornar tanto quanto possível as falhas ou limitações dos serviços com os quais comunica.

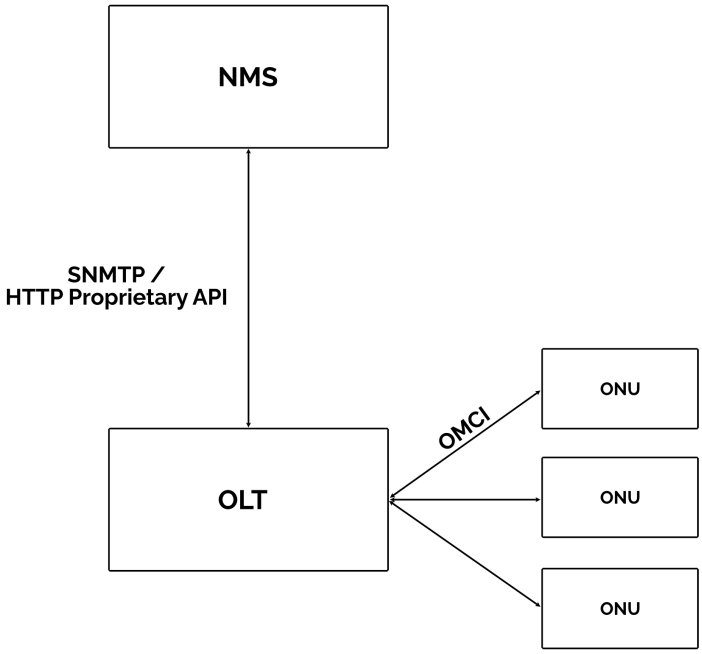


Figura 3: Comunicação entre *OLT* e *ONT*

Como se consegue perceber na Figura 3, a *OLT* e a *ONT* intercomunicam-se através do canal *OMCI* [25]. No nosso caso a interação que importa analisar é a do *NMS* com a *OLT* e o modo como são determinados os pedidos encaminhados a cada momento.

No caso em estudo, a principal limitação conhecida é o facto de as *OLT's* não suportarem o processamento concorrente de pedidos, ou seja, cada pedido é tratado de modo atómico. Os pedidos realizados pelo módulo que contacta diretamente a *OLT* estão sujeitos a um período de *timeout*, ou seja, a partir do momento em que o referido componente contacta a *OLT*, caso não obtenha uma resposta num período de tempo bem definido então considera que esta já não chegará em tempo útil pelo que o pedido é concluído unilateralmente sem sucesso. Aliando o facto de o módulo que contacta a *OLT* ser *multi-threaded* ao facto de a mesma não ter essa capacidade, percebemos que existem determinados casos em que podem ser causadas cascatas de *timeouts* ao nível do referido módulo pelo simples facto de não se ter em conta nenhuma das limitações e permitir o tratamento concorrente de pedidos que resolvem na mesma *OLT* ao nível dos *workers*.

Consegue perceber-se pela Figura 4 que é possível que o tratamento mais demorado de uma mensagem comprometa o tratamento atempado (da perspetiva do *worker*) das restantes mensagens que são processadas bastante rápido ao nível da *OLT*. No esquema apresentado a segunda mensagem (apresentada a vermelho) demora mais do que *timeout* a ser processada e, consequentemente, as mensagens

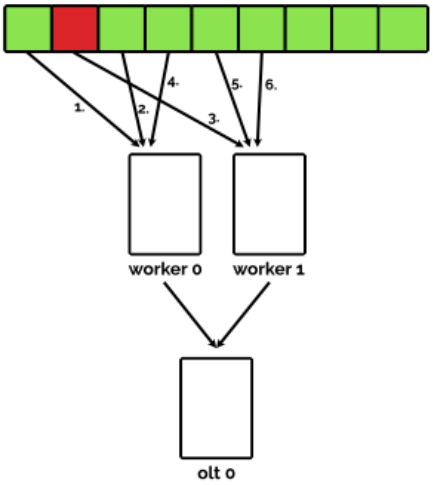


Figura 4: Esquema da passagem de mensagens no *ZTP*

que a sucedem na fila não serão tratadas a tempo visto que a mensagem mais demorada ocupa a *OLT*, neste caso a única, durante demasiado tempo.

Neste caso, em vez de se utilizar a concorrência para fomentar o mais rápido tratamento dos pedidos e por isso ter um maior *throughput*, utiliza-se para fornecer mais pedidos por segundo à *OLT* sem critério algum que permita fazê-lo de maneira consciente e proveitosa.

Por forma a ter em conta estas limitações e tentar otimizar as métricas descritas acima foram desenvolvidas algumas arquiteturas que tentam otimizar aquela que seria a arquitetura mais simples num ambiente em que é possível o tratamento concorrente de pedidos por parte dos *workers*. A arquitetura que seria imediata é aquela em que cada *thread* do *ZTP* consome e trata cada pedido assim que tem disponibilidade. Esta alternativa seria aquela que se implementaria no caso de não se ter nenhum tipo de conhecimento em relação aos restantes serviços com os quais a nossa aplicação comunica, assumindo-se que todos os serviços são *multithreaded* e ilimitados em termos de capacidade de processamento.

Como, por razões já apontadas, esta arquitetura apresenta potencial de manifestar problemas como cascatas de *timeouts* em determinados contextos, de seguida apresentam-se algumas alternativas que visam um melhoramento dos pontos de falhas da arquitetura base.

4.2 Arquiteturas para a gestão de filas de espera

Nesta Secção dão-se a conhecer as alternativas de arquiteturas para o novo módulo que foram concebidas. As arquiteturas são apresentadas por uma ordem que se acredita ser incremental em termos de qualidade e desempenho, impressão essa que será corroborada ou não pelos resultados das simulações que se realizaram.

As arquiteturas apresentadas foram desenvolvidas tendo em conta as limitações conhecidas dos equipamentos ou módulos de *software* com os quais o módulo a desenvolver interage como o facto de as

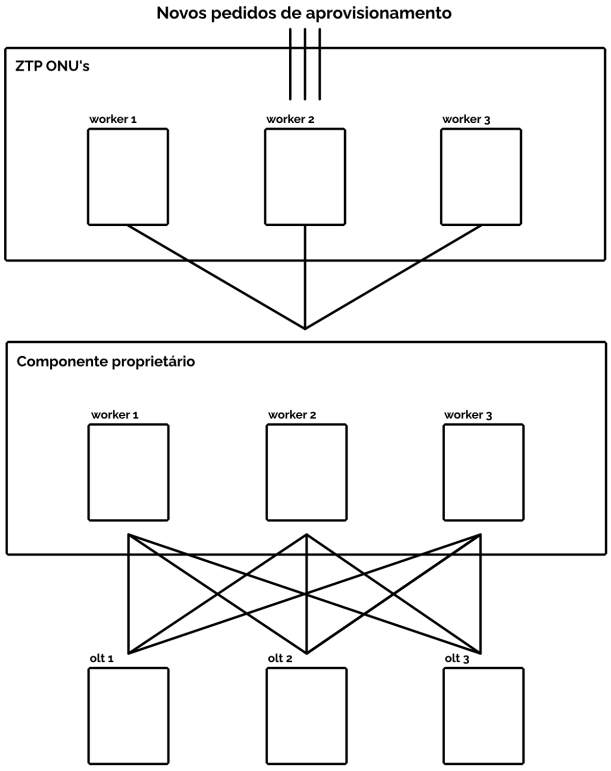


Figura 5: Arquitetura simplificada do ambiente de simulação

OLT's suportarem apenas o processamento sequencial de pedidos. Desta forma, começou-se com uma arquitetura que não considerava estas limitações e desenvolveram-se alternativas que visam minimizar o tempo de aprovisionamento e diminuir a probabilidade de ocorrência de *timeouts* no processamento de aprovisionamentos ou outros alarmes por parte do *ZTP ONU's*.

4.2.1 Arquitetura base - sem mecanismos de gestão de filas

Esta arquitetura serve como controlo por forma a se perceber o benefício real das restantes três alternativas em que se implementam alguns tipos de mecanismos de gestão de filas de espera. Nesta alternativa nenhum dos componentes implementa algum tipo de lógica que não seja processamento ou *forwarding* desinformado de mensagens, ou seja, cada *worker* assim que está disponível retira uma mensagem da fila de entrada do *ZTP* e processa-a. O pedido inicial é desdobrado nos diversos pedidos ao módulo que comunica com a *OLT*. Todos os resultados obtidos com as outras alternativas são comparados com os desta alternativa para perceber se, de facto, os mecanismos de gestão implementados preconizam os benefícios que foram idealizados. A Figura 6 configura um esquema ilustrativo desta arquitetura.

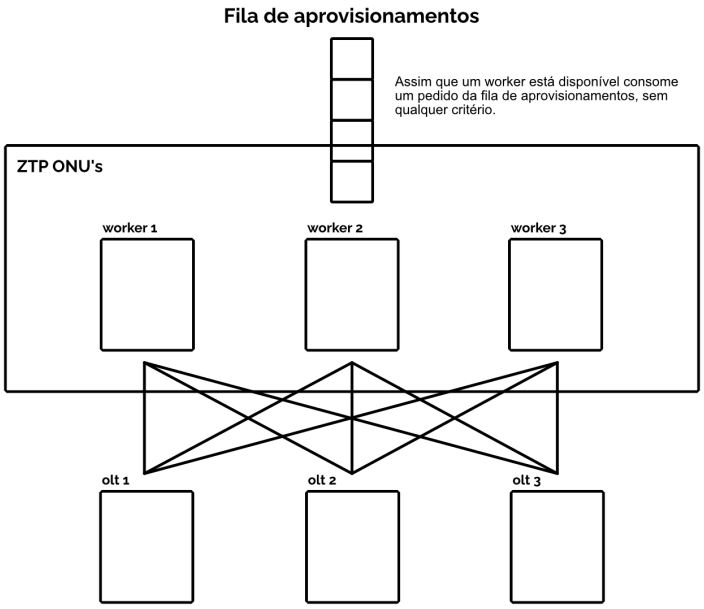


Figura 6: Esquema da alternativa base - sem mecanismos de gestão de filas

4.2.2 Arquitetura melhorada - 1ª alternativa

A diferença que esta alternativa apresenta em relação à arquitetura base é o facto de cada *worker* esperar que o tratamento de um pedido para a *OLT* a que o pedido que ele tem de processar se destina acabe antes de prosseguir com o seu processamento. De uma forma mais clara: imaginemos que o *worker-0* recebe uma mensagem que exige um processamento na *olt-0*. Se um qualquer outro *worker* estiver à espera da conclusão do tratamento de um qualquer pedido na *olt-0* então o *worker-0* aguarda a conclusão desse processamento e só depois realiza o pedido à *olt-0*. O que se espera é que esta alternativa reduza a ocorrência de *timeouts* uma vez que serializa os pedidos às *olts* ao nível dos *workers*. Esta arquitetura, no entanto, exige uma fonte de informação que guarda informação de qual é a *olt* destino do processamento que cada *worker* está a executar a cada momento. Esta fonte de informação encontra-se no *broker*, que é um processo que em algumas das próximas arquiteturas tem a função de distribuir mensagens e guardar a informação descrita. A Figura 7 configura um esquema do funcionamento desta alternativa de arquitetura.

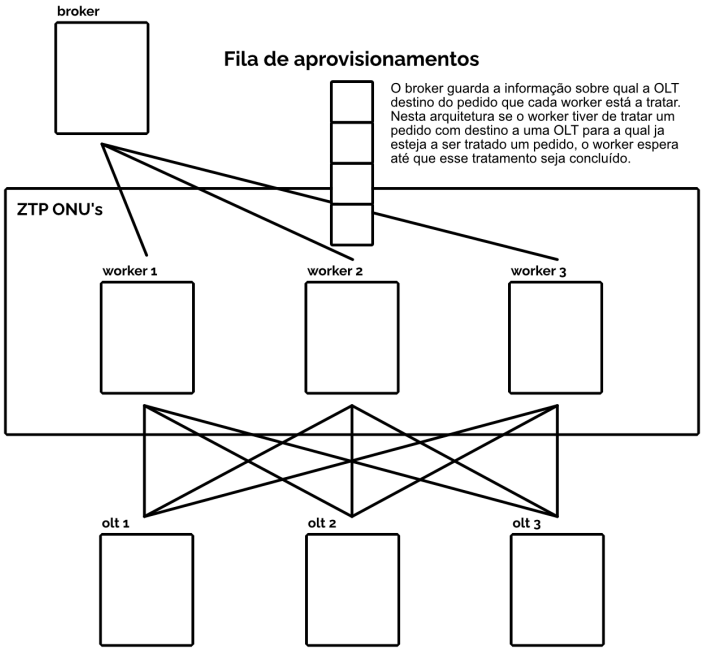


Figura 7: Arquitetura melhorada - 1ª alternativa

4.2.3 Arquitetura melhorada - 2ª alternativa

Esta arquitetura pressupõe já a existência de um processo no *broker* que tem mais funções do que simplesmente manter a fonte de informação descrita na arquitetura anterior. O objetivo desta arquitetura é que pedidos que tenham como destino a mesma *olt* sejam invariavelmente encaminhados para o mesmo *worker* o que acaba por impedir que pedidos direcionados à mesma *olt* sejam processados concorrentemente pelos diversos *workers*. O efeito esperado é a minimização da probabilidade de ocorrência de *timeouts* no tratamento destes pedidos.

Para materializar esta arquitetura recorre-se a uma função de *hash* que recebe como parâmetro o endereço de IP da *OLT*. Este mecanismo garante uma distribuição relativamente justa dos pedidos pelas *OLT's*. Com o algoritmo descrito também se garante a condição que sutenta esta alternativa de arquitetura: pedidos que se destinam à mesma *OLT* são, impreterivelmente, tratados pelo mesmo *worker*.

No cenário de simulação este algoritmo foi simplificado pelo facto de as *OLT's* emuladas não possuírem endereço de IP levou à necessidade de simplificação do algoritmo, que é possível pelo facto de se manterem as propriedades de distribuição de mensagens. Neste ambiente recorre-se à função do resto da divisão do número que identifica a *OLT* a que se destina o pedido pelo número de *workers* disponíveis na simulação. Como se pode ver, pedidos que se destinam à mesma *OLT* continuam a ser, impreterivelmente, tratados pelo mesmo *worker*. A Figura 8 configura um esquema do funcionamento desta

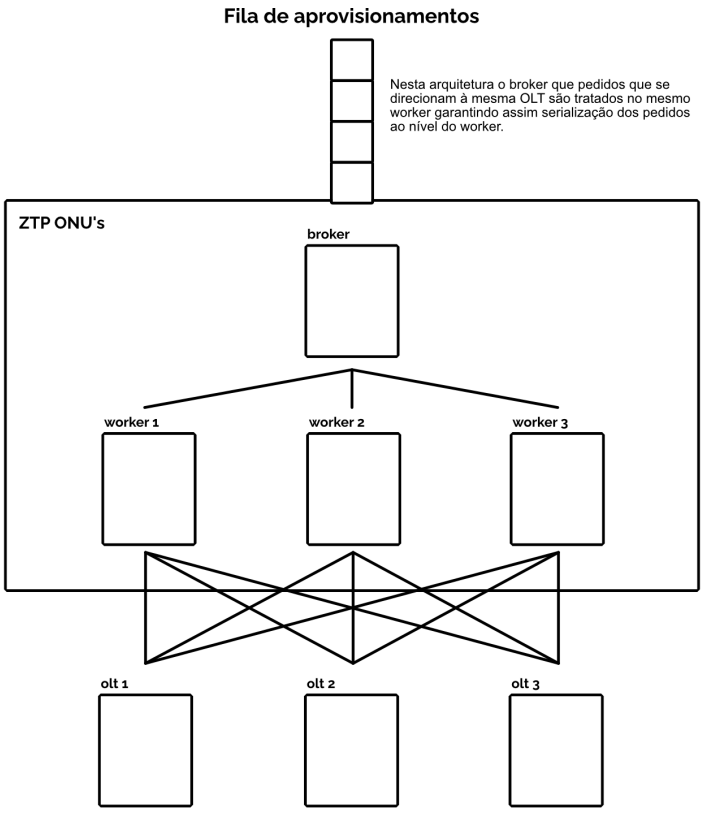


Figura 8: Arquitetura melhorada - 2ª alternativa

alternativa de arquitetura.

4.2.4 Arquitetura melhorada - 3ª alternativa

Esta última alternativa representa um melhoramento em relação à alternativa anterior na medida em que limita menos o aproveitamento do potencial de concorrência ao nível dos *workers* porque, como vamos perceber, não estabelece uma ligação entre o destino da mensagem e o *worker* que a trata. O encaminhamento neste caso é dependente do contexto.

Nesta arquitetura o algoritmo de encaminhamento é diferente do anterior, na medida em que se no momento de encaminhamento existir algum *worker* a tratar de um pedido para a *olt* destino do nosso pedido, então o *broker* encaminha o pedido para esse *worker*. Desta forma, é evitado o tratamento concorrente de pedidos para a mesma *OLT*. A Figura 9 configura um esquema do funcionamento desta alternativa de arquitetura.

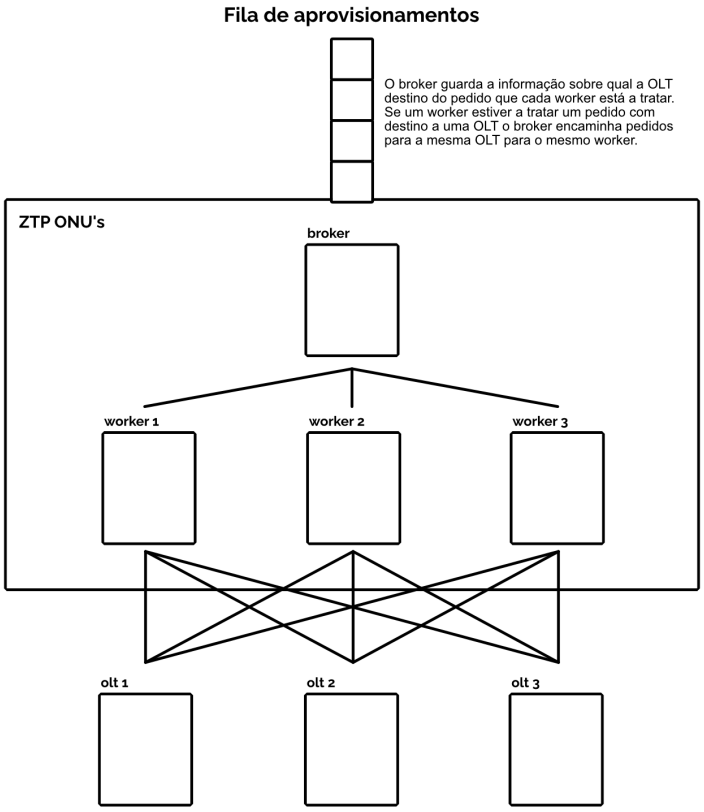


Figura 9: Arquitetura melhorada - 3ª alternativa

4.2.5 Análise comparativa das arquiteturas alternativas para a gestão de filas

Obviamente, a melhor maneira de comparar as arquiteturas alternativas é ter implementações do *ZTP* em cenários reais de utilização e recolher as métricas referidas durante um período alargado de tempo de utilização do sistema. Desta forma obter-se-iam resultados totalmente representativos, dado que a carga imposta sobre o sistema seria uma carga real, ou seja, aquela que se espera encontrar em cenários de produção. No entanto, este tipo de cenário não é possível em virtude do tempo que demoraria a recolher os dados relativos a cada um dos cenários assim como pelo tempo que demoraria a implementar cada uma das soluções alternativas para arquitetura e lógica do *ZTP*. No entanto, é essencial recolher dados que permitam realizar uma escolha informada por uma das arquiteturas. Por todas estas razões escolheu-se desenvolver um ambiente de simulação que, embora simplificado, mantém as propriedades essenciais do sistema preconizado pelo *ZTP*. Esta necessidade deu, assim, origem ao projeto descrito na próxima secção.

4.3 Descrição e justificação do ambiente de simulação

Para realizar uma escolha informada e devidamente justificada considera-se necessário possuir dados relativos ao número de pedidos que incorreram em *timeout* e o tempo médio de tratamento de cada pedido. As dificuldades de implementar cada uma das arquiteturas num ambiente de produção foram já elencadas pelo que escolheu-se desenvolver um ambiente de simulação que, conforme se explica, conserva as propriedades essenciais da comunicação do *ZTP* com os restantes componentes do *AGORA*.

Este ambiente tem quatro componentes essenciais: um produtor que simula os clientes do *ZTP* na medida em que cria os pedidos, os *workers* que simulam as *threads* do módulo de *software*, o *broker* que é um processo necessário na terceira e quarta arquiteturas e, por último, processos que emulam o comportamento das *OLT's* no tratamento dos pedidos que recebem do serviço que com elas comunica.

É importante deixar claro que o funcionamento do ambiente de simulação, tal como o *ZTP*, é garantido através da utilização de filas de espera de mensagens que são o principal foco desta parte da investigação. Cada um dos componentes mencionados tem uma fila de mensagens que é necessária tendo em conta a natureza assíncrona da comunicação entre os componentes. É igualmente importante deixar claro que tal como se pretende no cenário real, os *timeouts* são decretados ao nível do *worker*, ou seja, o *worker* não espera mais do que *timeout* pela resposta, embora o tempo de processamento do pedido ao nível da *OLT* possa ser bastante superior.

A maneira como este projeto é configurado permite correr uma simulação com um número variável de mensagens, *workers* e *olts*. Para além disso, pode ser escolhida qualquer uma das arquiteturas descritas acima. Assim, uma simulação pode ser agendada através do envio de um objeto *json* de configuração com os seguintes campos: `workers`, `olts`, `messages`, `algorithm` e `sequence`. O valor fornecido para os `workers` deve estar compreendido entre 1 e 30, o das `olts` deve estar entre compreendido entre 1 e 20. Já o número de mensagens não tem limite superior devendo, por isso, ser um valor maior ou igual a 1. A arquitetura da simulação deve ser um número entre 1 e 4 conforme explicado acima. Por último, uma simulação pode estar associada a um conjunto de mensages. Da sequência 1 para 3 a probabilidade de ocorrerem mensagens cujo tempo de processamento é maior do que *timeouts* aumenta.

O resultado de cada simulação é um conjunto de métricas que permite perceber o comportamento da arquitetura e tirar ilações que permitem posteriormente realizar uma escolha fundamentada de qual a arquitetura a implementar no *ZTP*. O resultado de cada simulação contém as seguintes métricas: tempo total de realização total da simulação e o número de *timeouts* verificados em cada uma das simulações.

4.3.1 Descrição do processo de aprovisionamento

Faz sentido relembrar que estamos também a tentar simular em termos de tempo aquilo que seria o processo de aprovisonamento, que aqui se configura como o tratamento das mensagens que chegam aos *workers*. Desta forma, para se perceber que o intervalo de tempo que se simula faz sentido convém deixar claro quais os tipos e tempos médios das ações que se estão a simular.

Consequentemente, antes de focar nas ações que são realizadas pela *OLT* faz sentido perceber quais as ações realizadas no âmbito do aprovisionamento de um novo *ONU*, de um modo geral. Lembrando quais são as funções de um *NMS* é mais fácil atribuir sentido a cada uma das ações que aqui são enumeradas. Um *NMS* tem a função de gerir tanto as redes como as informações relativas aos elementos de rede. Assim, as ações que ocorrem durante o aprovisionamento de um novo equipamento são:

1. Verificar, de acordo com as informações disponíveis na notificação do novo equipamento, se o mesmo já encontra registado no *AGORA*.
2. Se o equipamento não estiver registado, efetuar o registo do equipamento no *AGORA*.
3. Verificar se o aprovisionamento do equipamento implica uma atualização de *firmware*.
4. Realizar a atualização de *firmware*, no caso de esta se ter afigurado necessária.
5. Correr o *playbook* de aprovisionamento: ações que culminam com a ativação dos serviços do cliente e configuração completa do equipamento.
6. No caso de o aprovisionamento ser bem sucedido desativar o alarme de novo equipamento do tipo *ONU*.

O tempo que todos estes passos levam pode ser considerado, de alguma forma, constante para as mesmas condições de rede. Os passos **1**, **2** e **3**. Consistem simplesmente em fazer uma chamada à **API** do componente proprietário que gere os equipamentos pelo que em termos de tempo pode considerar-se que, para as mesmas condições de rede, o tempo que estes passos levam será o mesmo. O passo **4** implica uma conexão com maior transferência de dados com o componente que assegura a transferência da nova versão de *firmware*, no entanto, mais uma vez, para as mesmas condições de rede e mesma versão de *firmware* é expectável que o intervalo de tempo que este passo demora seja o mesmo em todos os aprovisionamentos. No passo **5** espera-se também que para a mesma conexão e o mesmo *playbook* o tempo de execução seja igual. Por último, o passo **6** é uma situação semelhante às três primeiras pelo que se justifica da mesma maneira.

Na Figura 10 ficam claras todas estas etapas do processo de aprovisionamento.

Por todas estas razões e pelo facto de se dever tentar introduzir, de alguma maneira, a imprevisibilidade que se espera num ambiente de produção se justifica o sistema de geração de mensagens que se expõe na Sub-Secção 4.3.2.

4.3.2 Condições do gerador de mensagens do ambiente de simulação

É importante que o processo encarregue de gerar a carga de mensagens seja pensado de modo a emular, tanto quanto possível, o comportamento real dos clientes e que gere um padrão de mensagens que seja próximo daquele que é expectável em produção. A forma encontrada para que se consiga aproximar

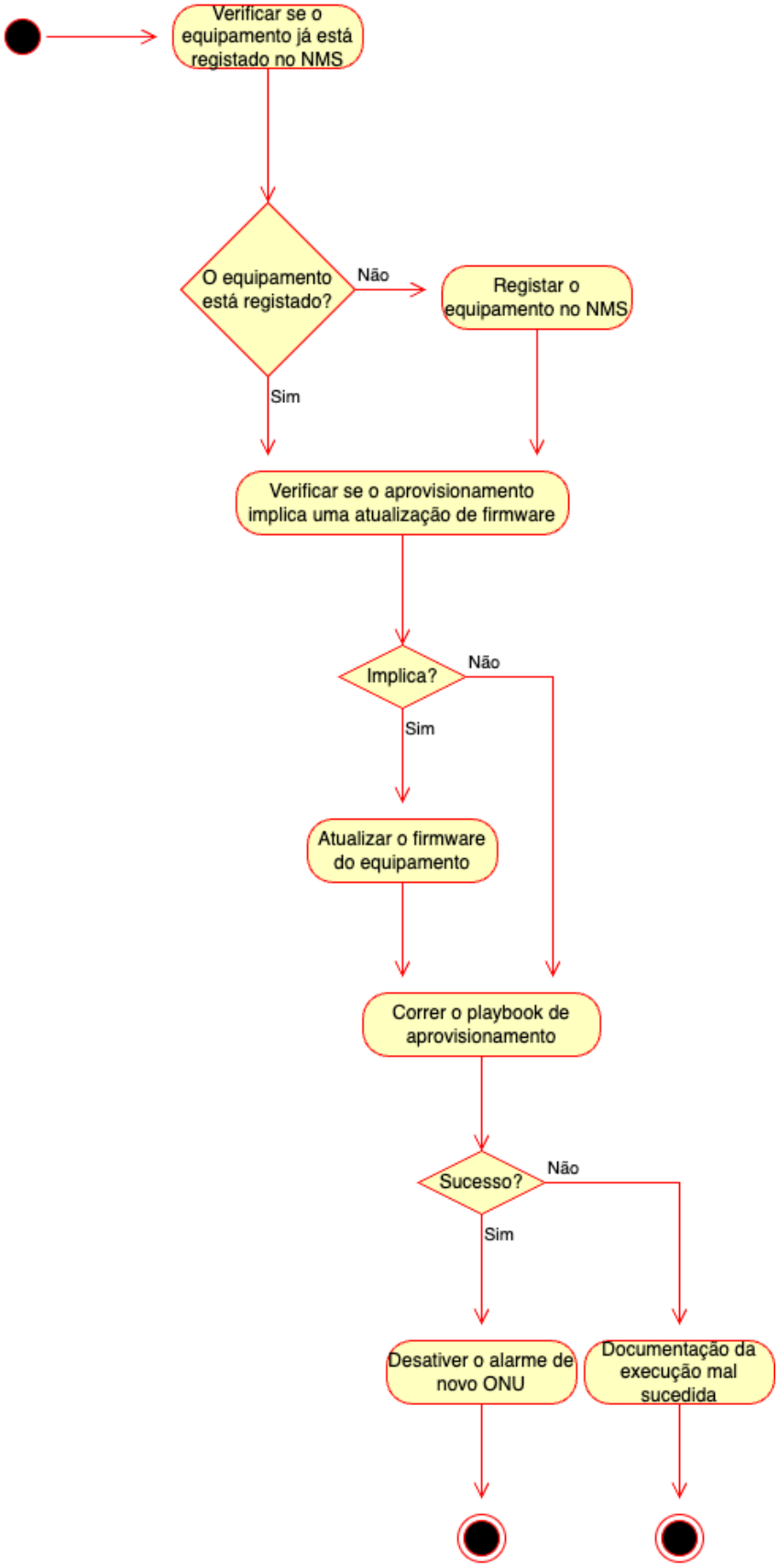


Figura 10: Diagrama de fluxo das etapas do aprovisionamento

o tipo de mensagens que chegam em ambientes de produção foi gerar uma determinada percentagem de mensagens cujo tempo de processamento é menor do que o tempo que demora a ser decretado *timeout* (**tipo 1**), uma determinada percentagem de mensagens cujo tempo de processamento é bastante próximo de *timeout* segundos (**tipo 2**) e um número de mensagens cujo tempo de processamento é bastante superior a *timeout* segundos (**tipo 3**). Desta forma, o número de mensagens de cada um dos tipos é diretamente proporcional ao número de mensagens total da simulação. A *OLT* à qual se destina a mensagem é escolhida de modo aleatório. No entanto, o objeto encarregue de implementar a aleatoriedade é gerado sempre com a mesma *seed* o que permite que a sequência de mensagens gerada para um determinado conjunto de parâmetros de configuração seja sempre a mesma.

Tipo de mensagem	Tempo de processamento
Mensagem tipo 1	$t \ll timeout$
Mensagem tipo 2	$t < timeout$
Mensagem tipo 3	$t \gg timeout$

Tabela 3: Tipos de mensagem geradas pelo ambiente de simulação

Conforme levemente abordado previamente existem três sequências possíveis de mensagens. O primeiro tipo de sequência é aquela em que todas as mensagens originam pedidos de processamento bastante rápido ao nível da *OLT* e que por isso não devem, em teoria, dar origem a um número elevado de *timeouts*. A sequência do **tipo 2** garante que 10% dos pedidos originados terão um tempo de processamento próximo do tempo findo o qual é decreado o *timeout*. Por último, a sequência três garante de 10% dos pedidos originados estão próximo de *timeout* e 10% têm um tempo de processamento acima de *timeout* garantido assim um número mínimo de *timeouts*.

Tipo de sequência	Constituição da sequência
Sequência tipo 1	100% mensagens do tipo 1
Sequência tipo 2	10% mensagens do tipo 2 + 90% mensagens do tipo 1
Sequência tipo 3	10% mensagens do tipo 3 + 10% mensagens do tipo 2 + 80% mensagens do tipo 1

Tabela 4: Consituição dos tipos de sequências geradas pelo ambiente de simulação

4.4 *Ratio* entre o número de *workers* e o número de *olts*

Como sabemos o dimensionamento de um serviço é um fator extremamente importante no desempenho que este apresenta nos mais diversos cenários de carga. O objetivo final do projeto do *ZTP* é instalar o módulo de *software* num ambiente de *cloud* que permite escalar a aplicação de modo reativo perante a carga a que está sujeita. Deste modo, têm de ser estabelecidas métricas que orientem a escalabilidade da aplicação e tem de ser percebido o ponto a partir do qual escalar a aplicação não apresenta vantagens em termos de desempenho. Com este objetivo em mente, decidiu-se fazer variar, no cenário de testes,

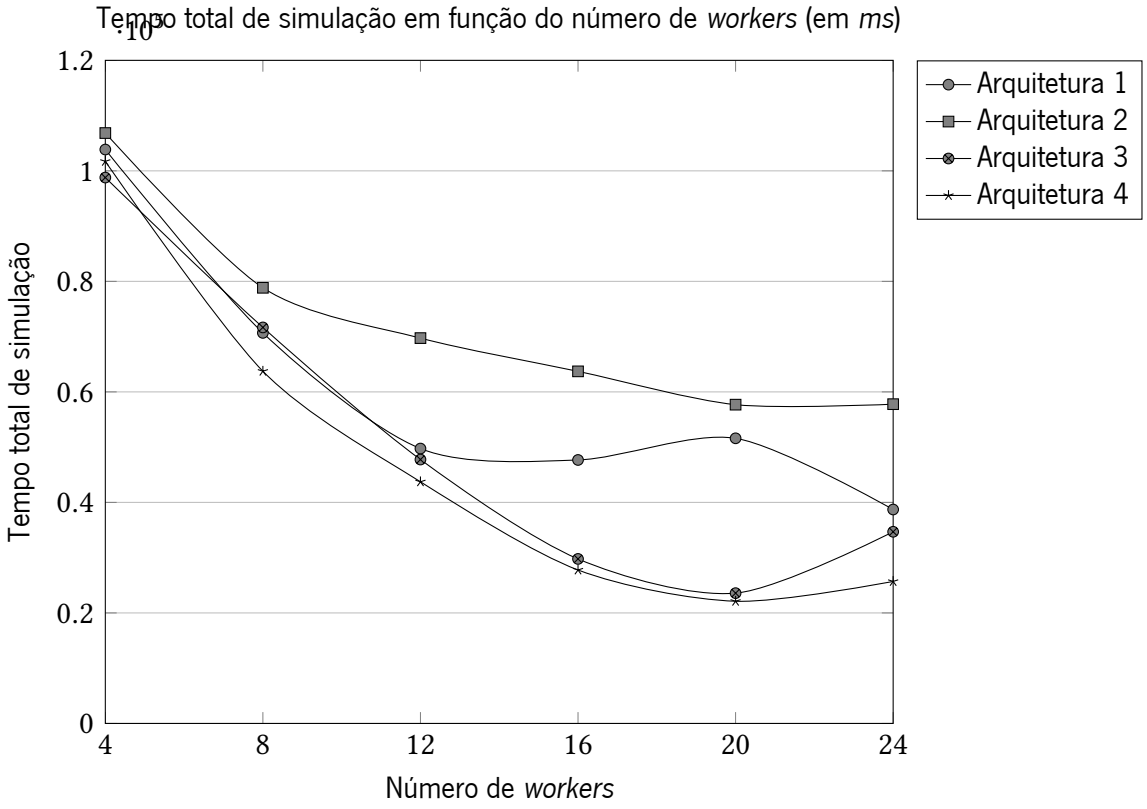


Figura 11: Comparação do tempo total da simulação em função do número de *workers*

o número de *workers* perante um número fixo de *olts* e tentar perceber a partir de que ponto adicionar novas *threads* deixava de preconizar um benefício em termos computacionais que justifica os seus custos. Para este cenário, é útil perceber o conceito da economia *diminishing returns* que explica que existe um ponto a partir do qual a adição de unidades de *input* não se traduz num aumento de unidades de *output* proporcional. Assim, o que se espera ver nos resultados à medida que se aumenta o número de *workers* é uma diminuição no tempo de processamento médio das mensagens até determinada altura em que a diminuição no tempo de processamento das mensagens é cada vez menor até deixar mesmo de se verificar.

É importante clarificar que nos testes cujos resultados são apresentados a seguir o número de *olts* simuladas foi 20 e o número de aprovisionamentos simulados foi de 400 e estes valores mantiveram-se constantes ao longo de todos os testes. Além disso, a sequência de mensagens utilizada foi a sequência de mensagens 3.

Recordemos que o objetivo principal das simulações realizadas é descobrir as configurações do ambiente de *deployment* tais que se consiga fazer o máximo de aprovisionamentos bem sucedidos no menor intervalo de tempo possível. Sendo assim, consultando a Figura 11 percebemos mais uma vez que o algoritmo 4 é aquele que minimiza o tempo de aprovisionamento e que o tempo de aprovisionamento desce até que atinge o seu mínimo quando o número de *workers* iguala o número de *OLT's*.

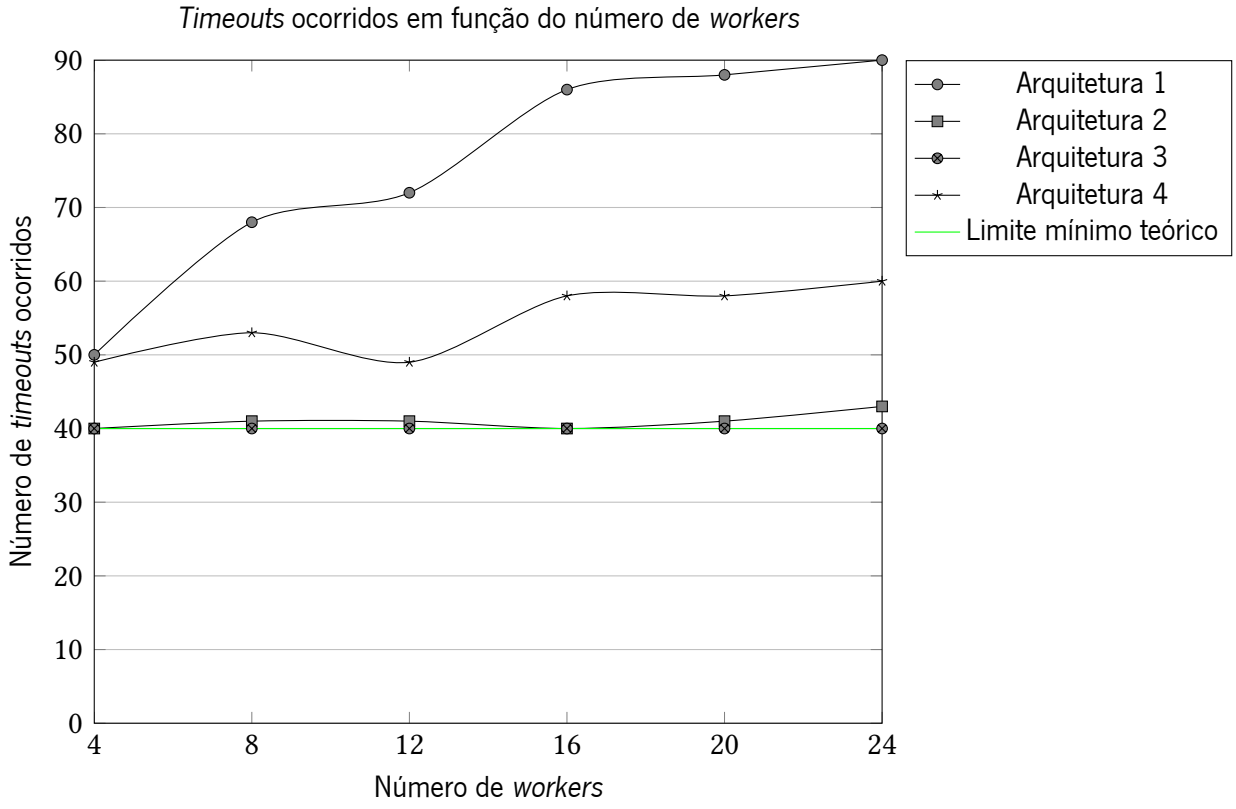


Figura 12: *Timeouts* ocorridos em função do número de *workers*

Relativamente ao número de aprovisionamentos falhados para cada configuração, consultando a Figura 12 conseguimos perceber que os algoritmos 2 e 3 estão, como seria de esperar, bastante próximos do mínimo teórico. No entanto, não se considera a variação do número de aprovisionamentos falhados significativa no algoritmo 4 pelo que a conclusão proveniente do gráfico relativo ao tempo de aprovisionamento se mantém - o valor ótimo do número de *workers* seria igual ao número de *OLT's*.

Para retirar uma conclusão dos dados apresentados é útil perceber o conceito da economia de *Diminishing Returns*: atendendo a este conceito, à medida que adicionamos *workers* obtemos um benefício cada vez menor em termos de tempo de processamento, até que em determinado ponto acrescentar mais *workers* não diminui o tempo de processamento. Pesados todos os factos demonstrados, o que se conclui é que em ambiente de produção o número de *threads workers* deve ser tão próximo quanto possível do número de *OLT's*. O número de *threads* igualar o número de *OLT's* é pouco exequível, dado que se podem gerir com o *NMS* redes arbitrariamente grandes e com elevado número de equipamentos do tipo *OLT*, pelo que se deve tentar ter tantas *threads* quantas sejam viáveis quer economicamente quer em termos de máquinas disponíveis.

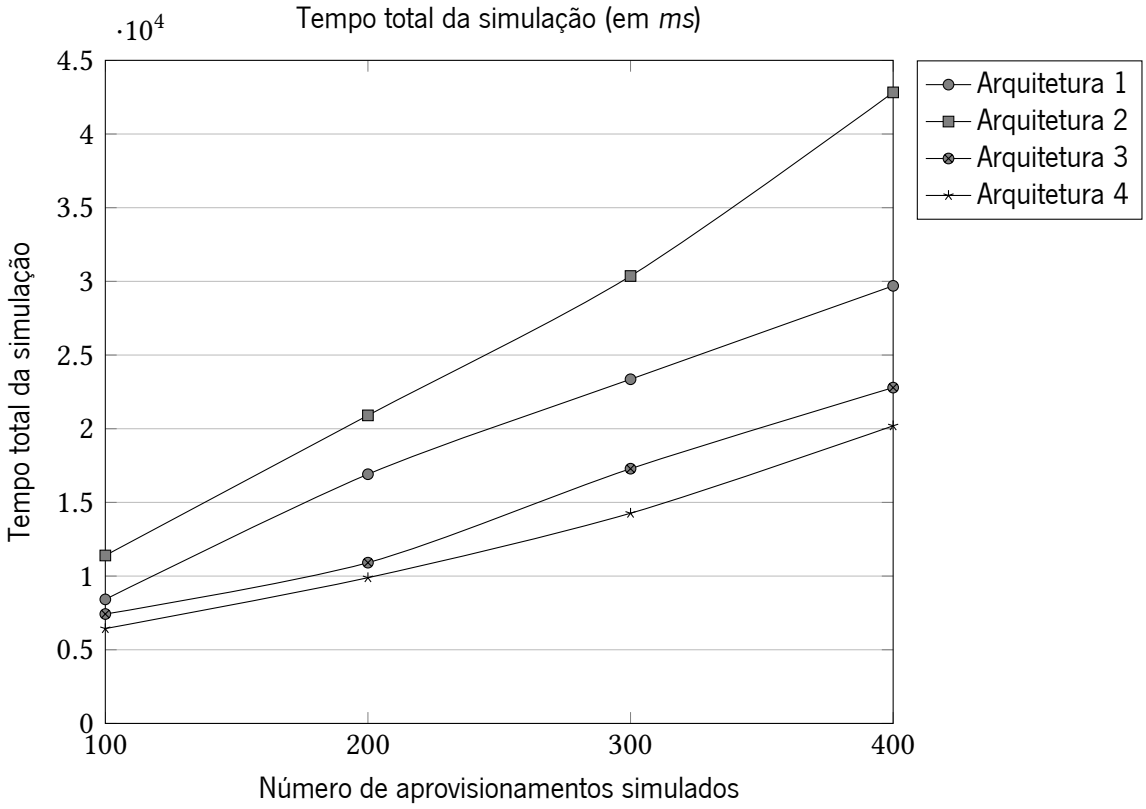


Figura 13: Tempo total de simulação a sequência do tipo 1

4.5 Apresentação dos resultados das simulações realizadas

Os resultados aqui apresentados dizem respeito a simulações efetuadas com o seguinte ambiente: 16 *workers* e 20 *olts*. Desta forma, o número de *workers* e *olts* não são variáveis de tal forma que influenciem os resultados finais.

Os gráficos de resultados apresentados dizem respeito a duas métricas: o tempo total de simulação e o número de aprovisionamentos falhados. Ambos os gráficos apresentam no eixo das abcissas o "Número de aprovisionamentos simulados", isto é, o número de notificações `NEW ONT` que se pretende simular ou, por outras palavras, o número de equipamentos que precisam de ser aprovisionados.

4.5.1 Simulações realizadas com a sequência do tipo 1

4.5.1.1 Tempo total de aprovisionamento e *timeouts*

Relembremos que esta é a sequência em que todos os pedidos têm um tempo de tratamento muito menor do que o período de *timeout* pelo que não é de esperar a ocorrência de muitos aprovisionamentos falhados. Isto é, em termos teóricos o limite mínimo de aprovisionamentos falhados é 0.

Na Figura 13 conseguimos constatar que as arquiteturas que permitem terminar mais cedo o tratamento de todas as mensagens são a 3 e a 4.

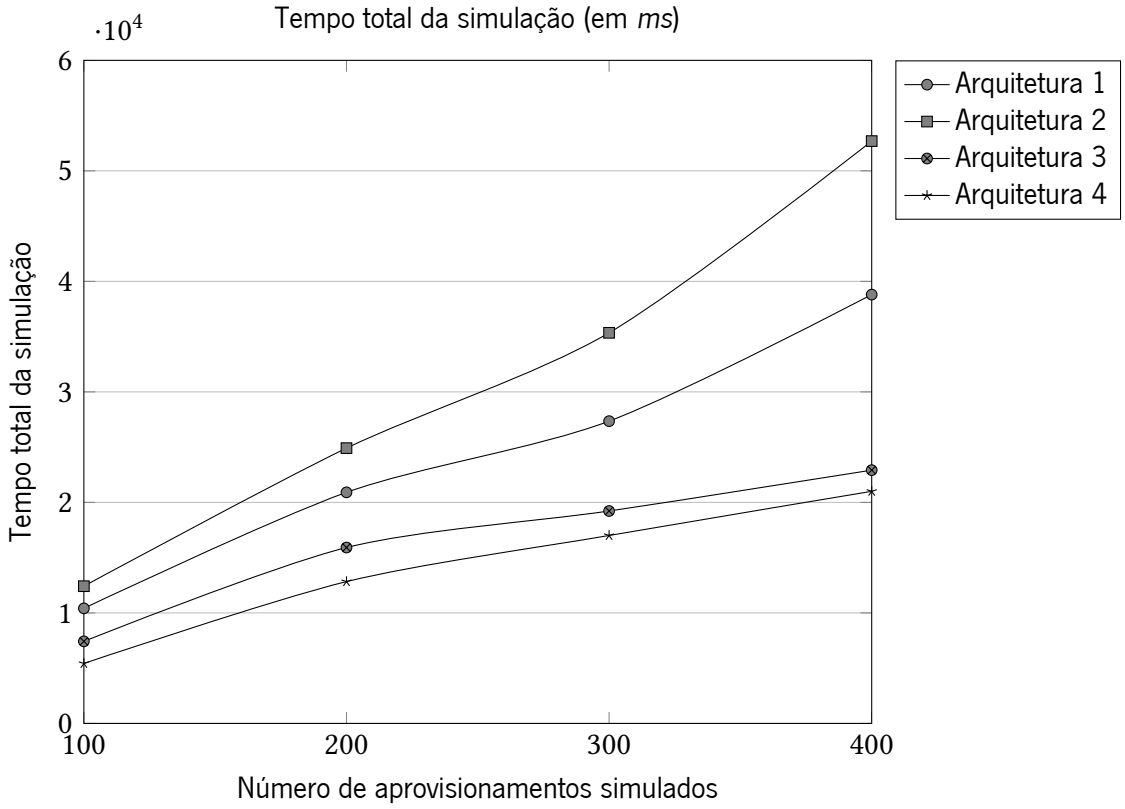


Figura 14: Tempo total de simulação com a sequência do tipo 2

Em termos de *timeouts* não é apresentado o gráfico uma vez que o valor verificado foi 0 para todas as simulações, ou seja, em nenhuma das circunstâncias se verificaram aprovisionamentos mal sucedidos.

É de ressalvar, no entanto, que este cenário não representa uma simulação precisa daquilo que se espera em produção. Em produção, quer por limitação das próprias *OLT's*, quer por outros fatores existem pedidos que demoram mais tempo a ser processados, sendo o objetivo principal deste exercício perceber qual a influência desses pedidos nos tempos globais de aprovisionamento e na criação de cascatas de *timeouts* em que a falha de determinados aprovisionamentos tem como consequência a falha de aprovisionamentos subsequentes.

Tendo isto em mente foi produzida uma sequência em que 10% dos pedidos demoram um pouco mais a ser processados (um tempo ainda abaixo do tempo de *timeout*), a **sequência do tipo 2**.

4.5.2 Simulações realizadas com a sequência do tipo 2

4.5.2.1 Tempo total de aprovisionamento e *timeouts*

Neste caso, conforme se constata na Figura 14, os tempos totais de aprovisionamento das arquiteturas 3 e 4 estão bastante próximos, o que faz algum sentido porque as estratégias implementadas têm como principal objetivo a diminuição da ocorrência de *timeouts*.

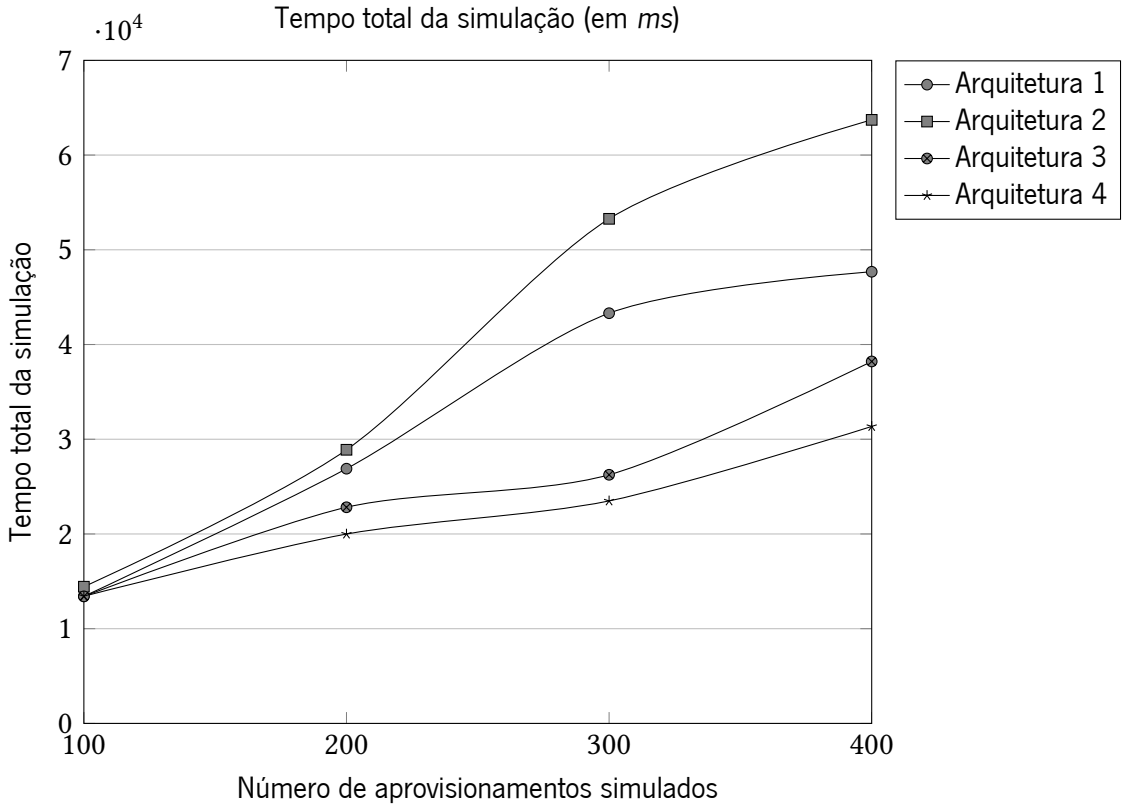


Figura 15: Tempo total de simulação com a sequência do tipo 3

Constata-se também que nesta sequência, com o número de aprovisonamentos realizados, não se verificam ainda aprovisionamentos falhados como consequência das mensagens de processamento mais longo.

Mas mais uma vez, esta não é a sequência que representa com maior fidelidade um cenário de produção. Num cenário de produção, em determinadas condições de rede, alguns pedidos podem causar um *"stall"* das *OLT's*. Isto é, demoram um tempo maior que o expectável a serem processados (tempo superior ao considerado como *timeout*).

4.5.3 Simulações realizadas com a sequência do tipo 3

4.5.3.1 Tempo total de provisão e *timeouts*

Os resultados apresentados na Figura 15 mostram que a arquitetura que permitiu um aprovisionamento mais rápido em todas as *OLT's* foi a 4. Num cenário em que a distribuição pelos pedidos é perfeitamente balanceada faria sentido que a arquitetura que apresenta melhores resultados fosse a 3, uma vez que constituiria o cenário perfeito para esta arquitetura, dado que se temos o mesmo número de pedidos para cada *OLT* estes serão perfeitamente distribuído pelos *workers*, de acordo com o algoritmo descrito para esta arquitetura. Precisamente para recriar cenários não ideais, ou seja, em que existe uma tendência para direcionar mais pedidos para determinadas *OLT's*, foi implementada uma probabilidade de 20% de

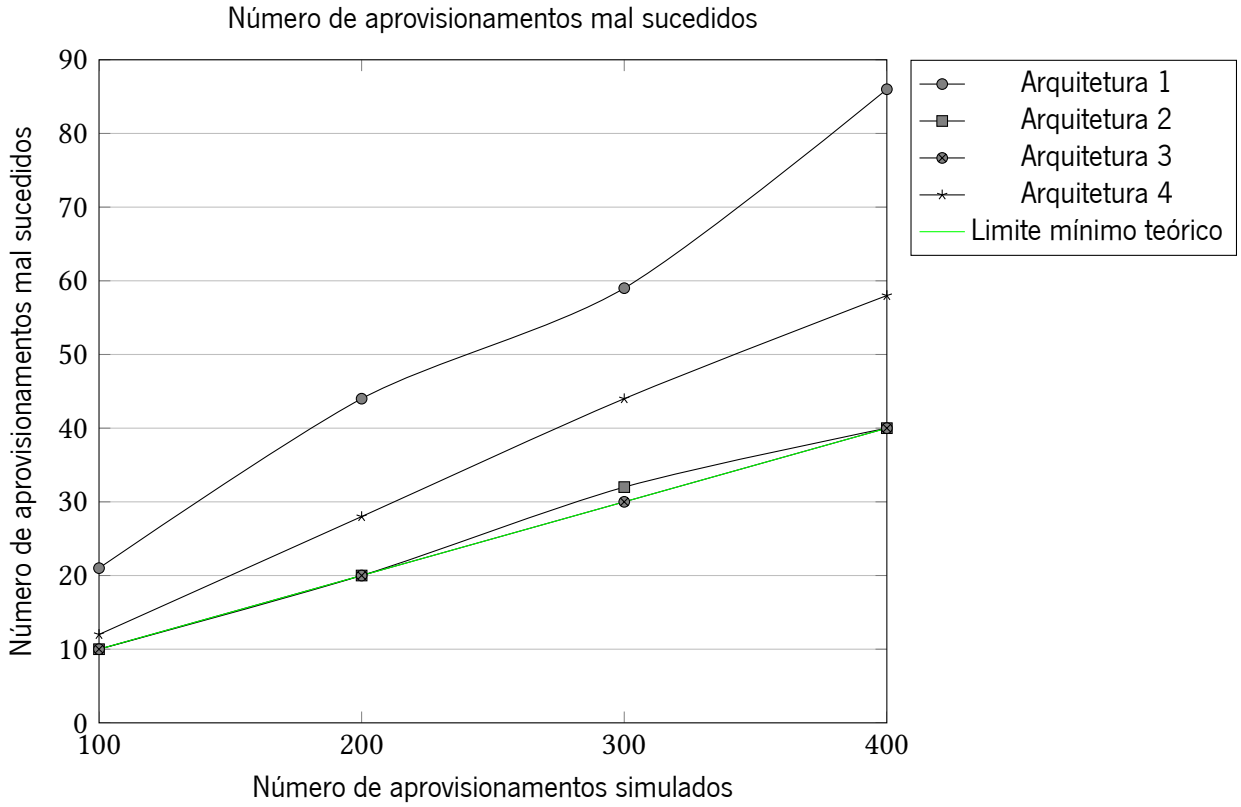


Figura 16: Tempo total de simulação com a arquitetura melhorada 3

ocorrer um *burst* de 3 pedidos com a mesma *OLT* destino. Neste tipo de cenários, como conseguimos constatar pelos resultados obtidos a arquitetura 4, ou seja, a 3ª melhoria apresenta resultados ótimos.

Estes resultados são também aqueles que mais devem pesar na decisão dado que este cenário é aquele que melhor representa cenários de produção em que o processamento dos pedidos tem as mais variadas durações e pode causar *timeouts*.

Como podemos perceber pelos resultados na Figura 16, as arquiteturas que minimizam em absoluto o números de aprovisionamentos mal sucedidos são as arquiteturas 2 e 3 uma vez que o objetivo é garantir a total serialização dos pedidos para a mesma *OLT* ao nível do *worker*, ao passo que a arquitetura 4 faz isto quando é possível, ou seja, quando o *broker* percebe que existe já um *worker* a processar pedidos relativos a uma mesma *OLT* encaminha para lá o pedido garantindo assim que caso o seguinte pedido incorra num *timeout* é porque demorou demais a ser processado na *OLT* e não porque esteve tempo a mais na fila da *OLT*.

Pode perceber-se pela natureza do algoritmo de encaminhamento que a arquitetura 3 tem uma fragilidade: uma *OLT* mal comportada, ou seja, que demora mais do que o normal a processar os pedidos que lhe são encaminhados pode fazer com que os pedidos subsequentes não sejam completados com sucesso, uma vez que o encaminhamento é independente da situação atual da *OLT*. Este problema é reduzido na arquitetura 4.

4.5.4 Seleção da arquitetura de gestão de filas a utilizar

Com os resultados apresentados temos a informação necessária para, primeiro, concluir se, de facto, os mecanismos de gestão de filas conferem vantagens em termos de tempo de processamento dos aprovisionamentos e do número de *timeouts*. No entanto, não de se deve perder de vista o objetivo principal: a escolha de qual o algoritmo que melhor serve o cenário de produção.

É importante analisar os resultados relativos a todos os tipos de sequências de mensagens enumeradas (quer as sequências de mensagens que incluem as de maior duração com as sequências de mensagens mais rápidas), uma vez que todos eles são cenários passíveis de ocorrerem em cenários de utilização real do serviço, embora, de acordo, com informações transmitidas por colaboradores da **Altice Labs** o cenário mais comum é aquele em que se verificam quer mensagens de duração rápida, média e longa. Por toda esta diversidade de cenários é importante escolher o algoritmo que apresenta o melhor desempenho na generalidade dos casos.

4.5.4.1 Resultados sequência 1

O cenário da sequência 1 é aquele em que o número de *timeouts* devia ser mínimo em cada uma das simulações, isto é, dado que todas as mensagens são de processamento rápido (bastante abaixo do tempo de *timeout*) e, consequentemente, o número mínimo teórico de provisões falhadas é zero. Na prática, estas previsões são confirmados pelos resultados: em nenhuma das arquiteturas se verificaram aprovisionamentos falhados, pelo que o único critério pelo qual poderemos estabelecer uma análise comparativa é, então, o tempo total necessário para que todos os aprovisionamentos sejam concluídos.

Neste cenário, o tempo de processamento das mensagens é tão rápido que não se espera que estas permaneçam durante muito tempo nas filas de espera pelo que a importância dos mecanismos implementados é menor. Ainda assim como se pode constatar na Figura 13, as arquiteturas 3 e 4 apresentam um desempenho muito satisfatório, melhor que a arquitetura 1, inclusivamente. Desta forma, as duas arquiteturas que se destacam nesta sequência são a 3 e a 4 com resultados bastante parecidos, constituindo isto um argumento a favor da sua possível escolha como a arquitetura a utilizar no *ZTP*.

4.5.4.2 Resultados sequência 2

Neste cenário temos já 10% dos pedidos que demoram mais tempo a ser processados, um tempo ligeiramente menor do que *timeout*. Neste cenário devia ser já mais evidente o efeito dos mecanismos de gestão de filas de espera implementados. O limite inferior teórico no que diz respeito aos *timeouts* continua a ser zero, visto que não existe nenhuma mensagem cujo tempo de processamento ultrapasse *timeout*.

O que se verifica na Figura 14 é que mais uma vez a arquitetura em que se verificam maiores tempos de processamento é a arquitetura 2 pelo facto de esta ter uma lógica puramente bloqueante. A arquitetura 1, apesar de apresentar melhores resultados que a 2, fica ainda bastante atrás da 3 e da

4. As arquiteturas 3 e 4 apresentam os menores tempos de processamento e encontram-se bastante próximas, razão pela qual se destacam.

4.5.4.3 Resultados sequência 3

Este é o cenário em que o limite teórico para os aprovisionamentos falhados não é zero, é 10% das mensagens geradas em cada uma das simulações. Constata-se na Figura 16 que a arquitetura que apresenta o maior número de aprovisionamentos falhados é aquela em que não são implementados mecanismos de gestão de filas de espera. Este resultado corrobora aquelas que são as previsões, visto que o processamento desinformado de pedidos leva à sobrepopulação das filas das *OLT's* levando assim a que, inevitavelmente, se verifiquem mais *timeouts*. As arquiteturas 2 e 3, muito por conta da sua natureza totalmente bloqueante, apresentam um número de aprovisionamentos falhados muito próximo daquilo que é o limite mínimo teórico. No entanto, esta natureza traduz-se depois num maior tempo total de processamento dos pedidos em relação à arquitetura 4, conforme se constata na Figura 15.

A arquitetura que fornece melhores resultados em termos de tempos de processamento de todos os pedidos é a arquitetura 4.

4.5.5 Seleção e justificação da arquitetura

Por todos os resultados que foram sendo apresentados e explicados fica claro que a arquitetura que deve ser adotada em ambiente de produção é a arquitetura 4 (4.2.4) principalmente por ser aquela que maximiza o número de aprovisionamentos bem sucedidos para o mesmo intervalo de tempo. Apesar de não ser a arquitetura que apresenta consistentemente as menores ocorrências de *timeouts*, o facto de o tempo total da simulação ser consistentemente menor nos cenários que melhor simulam o ambiente de produção, nomeadamente a sequência de mensagens 3, faz com que ainda assim se escolha esta arquitetura como a alternativa a adotar.

Por forma a justificar também a escolha da arquitetura de uma forma mais objetiva decidiu-se projetar e realizar um cálculo que permitisse para cada algoritmo e para cada número de aprovisionamentos perceber quão longe (em termos decimais) o tempo obtido pelo algoritmo se encontra do melhor resultado obtido no conjunto de todos os algoritmos. Se repararmos, por exemplo, na Figura 13, as simulações relativas a cada arquitetura têm 4 números diferentes de aprovisionamentos. O que se faz para cada uma das simulações é somar cada um dos desvios decimais e elege-se como melhor arquitetura aquela cuja soma desses desvios em relação aos resultados ótimos no conjunto de todos os números de aprovisionamentos simulados. De uma forma simples, a fórmula que avalia cada arquitetura para cada uma das sequências, sendo que $otimo$ representa o menor tempo de aprovisionamento para a combinação de sequência e número de aprovisionamentos e $arquitetura$ representa o tempo que a arquitetura em avaliação demorou a completar os aprovisionamentos, é a seguinte:

$$\sum \frac{(arquitetura - otimo)}{otimo} \tag{4.1}$$

Com efeito, a fórmula apresentada foi utilizada para corroborar as conclusões a que se chegou no parágrafo anterior. Apresentam-se, agora, os resultados sob a forma de tabelas:

Arquitetura	**Soma dos desvios percentuais**
Arquitetura 1	2.126
Arquitetura 2	4.134
Arquitetura 3	0.597
Arquitetura 4	0.000

Tabela 5: Desvios percentuais em relação ao ótimo na sequência de mensagens do tipo 1

Arquitetura	**Soma dos desvios percentuais**
Arquitetura 1	3.005
Arquitetura 2	4.818
Arquitetura 3	0.831
Arquitetura 4	0.000

Tabela 6: Desvios percentuais em relação ao ótimo na sequência de mensagens 2

Arquitetura	**Soma dos desvios percentuais**
Arquitetura 1	1.710
Arquitetura 2	2.821
Arquitetura 3	0.478
Arquitetura 4	0.001

Tabela 7: Desvios percentuais em relação ao ótimo na sequência de mensagens 3

Com os resultados das Tabelas 5, 6 e 7 consegue perceber-se, mais uma vez, que a arquitetura 4 é aquela que apresenta consistentemente os tempos mais próximos do resultado ótimo. Em todas as Tabelas a arquitetura 4 é aquela que apresenta a menor soma dos desvios percentuais. Desta forma, são corroboradas de uma forma mais objetiva as conclusões previamente transmitidas.

4.5.5.1 Diferenças entre a arquitetura de simulação e a arquitetura de produção

Existem, no entanto, algumas diferenças entre a arquitetura que será utilizada na realidade e aquela que foi utilizada no ambiente de simulação assim como existem diferenças entre os componentes que compõem essa mesmam arquitetura. Conforme foi já explicado os componentes que constituem a arquitetura de simulação têm o objetivo de simular o tempo que levaria aos componentes reais a realizarem determinada tarefa ao passo que os componentes reais realizam, de facto, a tarefa em questão. Tomemos o exemplo da *OLT* - no ambiente de simulação o processo que trata o pedido simplesmente executa um

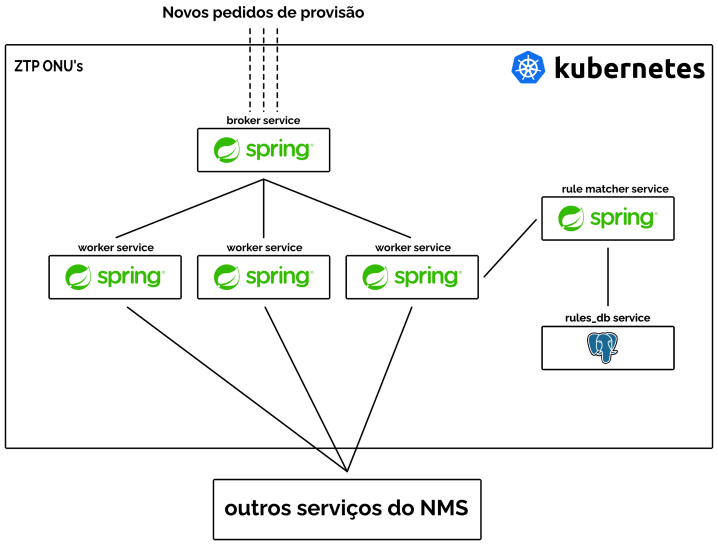


Figura 17: Arquitetura real do *ZTP*

`Thread.sleep()` durante o tempo que o pedido demora a ser processado; no ambiente real o que acontecerá é o pedido ser realmente tratado.

A grande diferença entre a arquitetura de simulação e a arquitetura real é o facto de a primeira não comunicar com serviços externos, ou seja, é como se o *ZTP* fosse completamente autosuficiente. No entanto, na arquitetura real como é evidente, os serviços necessitam de comunicar com outros serviços do *NMS* para realizar ações como registar o *ONU*.

Reparamos também, em comparação com a Figura 5, que na arquitetura real, apresentada na Figura 17, os *workers* não estabelecem comunicação direta com as *OLT's*, pelo facto de esta ser feita através de um outro módulo do *NMS* através de uma API REST proprietária.

5

Rule Matcher

O *ZTP* tem como objetivo a automatização do processo de aprovisionamento de novos equipamentos de rede e utiliza para isso *playbooks Ansible*. No entanto, é concebível e expectável a existência de diversos *playbooks* com diversos fins, com destino a serem executados em diferentes equipamentos em diferentes circunstâncias. O facto de a escolha de qual o *playbook* a utilizar depender do contexto e de qual o tipo de equipamento a aprovisionar ou ainda quais as ações que esse aprovisionamento implica faz com que se torne imperativa a existência de um módulo de *software* que dado um pedido de aprovisionamento e todas as informações nele contidas seja capaz de determinar qual o *playbook* a utilizar.

O referido pedido de aprovisionamento transporta informações absolutamente necessárias para determinar qual o *playbook* que deve ser utilizado. As informações transportadas são relativas ao equipamento gerador da mensagem, por exemplo, o endereço de *IP* do equipamento, a versão de *firmware* instalada, identificação relativa à *OLT* a que se encontra conectado, a carta a que se encontra conectado assim como a *interface*, entre outros.

5.1 Requisitos do módulo *Rule Matcher*

Para que o módulo seja completamente funcional é importante perceber quais os cenários de utilização que se pretende que sejam cobertos e ao mesmo tempo criar um sistema de tal modo flexível que permita lidar com parâmetros que tornem possíveis cenários de utilização e configuração que ultrapassem aqueles inicialmente previstos. Lembrando que o único cenário que tem sido mencionado é o do aprovisionamento de novos equipamentos, não se pode esquecer que o módulo deve suportar as mais variadas ações no futuro, como atualizações, instalações de *software*, conforme as ações que forem configuradas nos *playbooks*.

Foram identificados alguns cenários e ações de que devem ser contemplados pelo *ZTP* ainda que, de momento, sejam completadas através de outros módulos de *software*, é possível que no futuro venham a ser realizadas com recurso ao *ZTP*.

É importante recordar a topologia das redes *PON* para perceber alguns dos cenários que são descritos. Recordemos, então, que existe uma componente *core* da rede que se conecta com as *OLT's*. Os *ONU's*,

por sua vez, contectam-se aos anteriores.

Os cenários de utilização do *rule matcher* estão extremamente dependentes daquilo que são os cenários de utilização do próprio *ZTP* como um todo. O caso ao qual dedicamos a maior parte da nossa atenção é, obviamente, o aprovisionamento de novos equipamentos. No entanto, podem existir outros cenários: a utilização dos *playbooks* para atualizações de determinados *firmwares* em equipamentos específicos. É também possível que se pretenda aprovisionar os serviços de cliente necessários e para isso é necessário o aprovisionamento de diversos serviços de rede, por exemplo. Desta forma, fica claro que os cenários de utilização são diversos e, por isso, faz sentido que o modo como especificamos quais os equipamentos em que deve ser executado cada *playbook* deve ser o mais flexível possível. Mais à frente neste Capítulo serão elencados alguns cenários de utilização que deixam patente a flexibilidade do módulo que aqui se descreve a maneira como este serve os cenários descritos e está preparado para cenários mais inesperados.

Esta multiplicidade de cenários possíveis leva a concluir que é necessário relacionar os *templates* existentes com os parâmetros que caraterizam o ambiente em que um aparelho se encontra, para que este possa ser aprovisionado de acordo com o mesmo, caso seja esse o cenário pretendido. Deste modo, surgiu o conceito de regra que essencialmente é a associação de um *template* a parâmetros como versão de *firmware*, endereço de *ip* e os restantes parâmetros.

Com efeito, este serviço tem a função de receber um pedido e o contexto do equipamento responsável pelo mesmo e retornar o *template* ou *templates* a serem executados por forma a satisfazê-los.

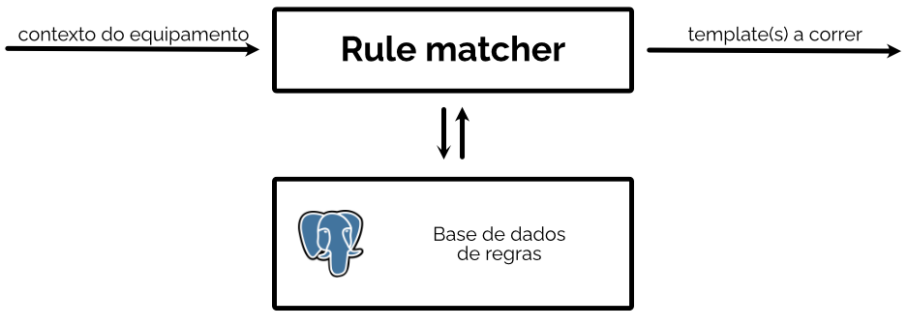


Figura 18: Esquema representativo das funções do *rule matcher*

5.2 Documentação do serviço *rule Matcher*

A implementação deste módulo assenta, mais uma vez, na criação de um microsserviço em *Spring Boot*. É importante, no entanto, definir um conceito que se utiliza neste componente, o conceito de **regra**. Uma **regra** é um conjunto de parâmetros que definem o ambiente, ou seja, o conjunto de condições que se têm de verificar para que determinada regra seja aplicada no aprovisionamento ou qualquer outra ação que seja configurada no *playbook* relativa a um equipamento.

Conforme ilustrado na Figura 18 existe uma base de dados em que são armazenadas as regras. Como tal, é útil perceber quais são exatamente os parâmetros que preconizam uma regra. Essa informação fica patente de forma simples na Tabela 8.

Parâmetro	Tipo de dados	Caráter
token	inteiro	metadado
firmwareVersion	string	parâmetro
ipAddress	string	parâmetro
olt	string	parâmetro
card	string	parâmetro
interface	string	parâmetro
equipmentId	string	parâmetro
password	string	parâmetro
vendor	string	parâmetro
parameters	array	parâmetro
priority	integer	metadado
template	string	metadado

Tabela 8: Parâmetros que integram as regras

Cada um dos parâmetros representa uma condição que deve ser encontrada no ambiente de um novo aparelho a ser aprovisionado com a utilização desta regra, enquanto que os metadados servem para caraterizar a regra. Os parâmetros apresentados são aqueles que são mais comuns para escolher qual a regra a utilizar em cada caso, no entanto, a necessidade de precaver casos em que os mesmos não são suficientes levou à necessidade de implementação de um mecanismo que permita adicionar parâmetros novos que não são formalmente suportados e validados. Neste sentido, criou-se um novo parâmetro na regra que é *parameters*. Este último permite adicionar uma lista de parâmetros com um nome e um valor que em *runtime* são também testados e, por isso, fazem parte do processo de *match*.

Um dos requisitos é também conferir flexibilidade máxima a cada parâmetro configurável tendo em conta os possíveis cenários de utilização. O que isto quer dizer é que se devem considerar mais casos do que a simples igualdade em todos os casos em que isso fizer sentido. Com efeito, analisa-se para cada parâmetro o tipo de casos de utilização possíveis e deriva-se assim quais as opções de configuração que devem ser garantidas para cada campo.

O *token* não é um campo de configuração da regra, é apenas o inteiro que a identifica na tabela da base de dados onde estas são guardadas, pelo que não se justifica a sua análise neste caso.

No caso da *firmwareVersion* existem diversas situações que devem ser configuradas. Existem pelo menos duas possibilidades que devem ser asseguradas neste parâmetro: o caso de se querer que a versão de *firmware* do equipamento seja igual à especificada na regra e o caso de se querer que a versão de *firmware* do equipamento seja inferior ou superior à especificada na regra. No entanto, na especificação da versão de *firmware* do equipamento é utilizado *Semantic Versioning* pelo que faz todo o sentido adicionar a possibilidade de comparar versões *major*, *minor* e *patch*. Por último faz sentido que todas estas opções possam ser combinadas com operações que preconizem o 'e' e o 'ou' lógicos de modo a combinar tantas operações quanto necessárias de modo a conseguir a filtragem pretendida. Com efeito, foi implementado no módulo o suporte a *tokens* que configuram estas ações: `gt`, `eq`, `lt`, `major`, `minor` e `patch`. Estes *tokens* devem sempre ser seguidos da respetiva versão de modo a preconizarem uma configuração válida. Por último, é ainda possível combinar estas expressões com os operadores `and` e `or`.

O *ipAddress* é um outro parâmetro que deve conferir grande flexibilidade e consequentemente os seus casos de uso devem ser bem analisados. Após análise com colaboradores da empresa, a conclusão a que se chegou é que é extremamente importante fazer com o *match* de *IP* seja possível atráves da sua comparação integral mas também que permita adiconar nas regras o *match* de endereços que começam ou acabam com a expressão fornecida, configurando uma comparação parcial do endereço. Com efeito, a configuração destas possibilidades foi feita mais uma vez com recurso aos seguintes *tokens*: `startswith`, `endswith` e `eq`. Novamente estas expressões podem ser combinadas com recurso aos *tokens*: `and` e `or`.

Os *tokens*: `startswith`, `endswith` e `eq` estão disponíveis para todos os restantes parâmetros cujo tipos de dados é *string*, à exceção de *template*. Com efeito todos os tipos de operações descritas, incluindo os operadores `and` e `or` estão também disponíveis para estes parâmetros estão também disponíveis para estes parâmetros.

O parâmetro `priority` permite dotar as regras do *rule matcher* de uma prioridade, ou seja, um valor inteiro que faz com que seja possível decidir qual a regra a utilizar em casos de empate (casos em que mais de uma regra é respeitada pelo equipamento). Por exemplo, quando um ambiente de um equipamento respeita exatamente o mesmo número de parâmetros de uma ou mais regras é possível decidir qual delas utilizar com recurso a este parâmetro. É importante referir que a prioridade é tanto maior quanto menor for o valor do parâmetro, isto é, uma regra com o valor 1 no parâmtetro tem maior prioridade que uma regra com o valor 2. No caso em que a prioridade é igual e, por isso, não permite o desempate ambas as regras são retornadas como resultado do *rule matcher*. Em casos em que mais que uma regra com o mesmo nível de prioridade é encontrada, então o critério utilizado para um possível desempate é o número de parâmetros *matched*.

Esta linguagem específica utilizada na definição dos parâmetros da regras despoletou uma pesquisa sobre qual a melhor e mais económica maneira de a implementar e se, por ventura, existe algum *software* que facilite a sua configuração. A pesquisa por algum pacote que facilitasse que implementasse diretamente algum tipo de *query language* revelou-se infrutífera. No entanto, conseguiu-se perceber que uma

boa alternativa para implementação desta *Domain Specific Language* é a elaboração de uma gramática, que permite em conjunção com as ações configuradas, ao mesmo tempo validar as regras que se tentam criar assim como ajudar no processo de *matching*. A pesquisa por ferramentas para a implementação de gramáticas em *Java* revelou diversas alternativas: *ANTLR* [12], *APG* [13] e *BYACC* [15]. Todas estas alternativas serviriam para implementar a gramática descrita, no entanto, a popularidade e simplicidade de utilização da ferramenta *ANTLR* fizeram que recaísse sobre ela a escolha.

5.3 Cenários de utilização

Esta Secção tem como objetivo fornecer alguns exemplos de regras que poderiam ser utilizadas para configurar as ações a realizar em determinados cenários. Em cada um dos cenários é descrita a condição em que a regra deve ser aplicada e de seguida apresenta-se um exemplo de como a regra poderia ser configurada de modo a conseguir o efeito pretendido.

As ações mais comuns que um novo equipamento pode necessitar prendem-se com o aprovisionamento de serviços, instalação de *software* e atualizações de *firmware*. No entanto, a riqueza do *rule matcher* prende-se com o facto de se poder estabelecer as regras de modo a tornar os *playbooks* apenas aplicáveis quando as condições que nelas especificamos se verificarem. Sendo assim, o exercício que agora se faz é elencar alguns casos hipotéticos que demonstram que a implementação atual deste módulo torna possível a filtragem dos cenários e o estabelecimento das regras é intuitivo e não implica uma grande curva de aprendizagem.

Consideremos o caso de um *ONU* num cenário de cliente que é conectado na rede e, por isso, necessita de ser aprovisionado. Neste caso, demonstremos a possibilidade de aplicar esta regra apenas para equipamentos cujo endereço de *ip* começa por `192.168` e a versão atual de *firmware* é *major* 2, no âmbito de *Semanting Versioning*. Uma implementação da regra que preconiza o cenário descrito é:

regra1.json

```
{
  "token": 1,
  "firmwareVersion": "major 2",
  "ipAddress": "startswith 192.168",
  "olt": null,
  "card": null,
  "interface": null,
  "equipmentId": null,
  "password": null,
  "vendor": null,
  "parameters": [],
  "priority": 1,
```

```
  "template": "provision_client_services.yml"
}
```

Pode, no limite, existir um caso em que se pretende realizar uma ação num *ONU* muito específico. Convém deixar claro que este é uma caso extremamente raro e que não se integra nos casos que foram considerados no desenho da solução deste módulo. No entanto, é possível realizar esta ação através de uma regra que especifique o identificador único do *ONU*, o parâmetro *equipmentId*.

regra2.json

```
{
  "token": 2,
  "firmwareVersion": null,
  "ipAddress": null,
  "olt": null,
  "card": null,
  "interface": null,
  "equipmentId": 6772306984,
  "password": null,
  "vendor": null,
  "parameters": [],
  "priority": 1,
  "template": "specific_action_for_equipment.yml"
}
```

Podemos imaginar ainda uma caso em que é detetado um erro crítico numa versão de *firmware* e que este erro provoca graves problemas de funcionamento em equipamentos de um *vendor* específico. Para se conseguir corrijir esse erro através de um *upgrade* de *firmware* pode ser criada uma regra que identifique estes mesmos aparelhos.

regra3.json

```
{
  "token": 3,
  "firmwareVersion": "eq 2.0.1",
  "ipAddress": null,
  "olt": null,
  "card": null,
  "interface": null,
  "equipmentId": null,
  "password": null,
  "vendor": "PT",
```

```
    "parameters": [],
    "priority": 0,
    "template": "firmware_upgrade.yml"
}
```

5.3.1 Estado de execução de cada regra

É importante também saber a cada momento qual o estado de execução de cada regra. É evidente que cada regra, por si própria, não possui estado dado que é destinada a ser utilizada por vários equipamentos. Assim sendo, só faz sentido pensar no estado de uma regra quando associada ao aprovisionamento de um equipamento. Na arquitetura apresentada para o *ZTP* faz sentido pensar em três tipos de estados: *QUEUED*, *IN COURSE* e *PROVISIONED*. Com efeito, este requisito pode ser assegurado com recurso a uma nova tabela na base de dados com o seguinte formato:

Identificador da regra	**Identificador do equipamento**	**Estado de execução**
string - foreign key	string - foreign key	string

Tabela 9: Tabela do estado do aprovisionamento de cada equipamento

De acordo com a arquitetura demonstrada e o processo através do qual se aprovisionam os equipamentos é importante estabelecer em que etapas do processo os estados apontados acima são atribuídos ao processo de aprovisionamento. Quando um pedido chega ao módulo *ZTP* é recebido pelo *broker* e é criada a entrada na tabela de estados, com o estado *QUEUED* para o aprovisionamento do equipamento. Eventualmente, um dos *workers* receberá o pedido e encaminha-lo-á para uma das *OLT's*, momento em que o estado do aprovisionamento é atualizado para *IN COURSE*. Finalmente, quando o *worker* recebe a resposta por parte da *OLT* o estado do aprovisionamento é atualizado para *PROVISIONED*.

5.3.2 Considerações finais

A implementação considerada deste serviço não é ótima e não tem como objetivo ser encarada como a implementação final que deve fazer parte do *ZTP*. Esta versão do *rule matcher* é uma prova de conceito e serve para deixar patentes as funcionalidades a serem implementadas na versão final. Obviamente que a implementação do serviço que fará parte do *ZTP* terá de ter em conta a estrutura dos dados fornecidos pelos módulos com os quais comunica, ao passo que esta implementação é um serviço *stand-alone*.

6

Segurança

Conforme foi já mencionado no Capítulo 2 a arquitetura do serviço *ZTP* é assente no conceito de microsserviços. As razões que sustentam esta opção foram também já analisadas pelo que neste Capítulo se desenvolve uma pesquisa alargada sobre quais as principais preocupações que devem ser tidas assim como quais as soluções de mitigação das mesmas que, de momento, se configuram como o *standard* na indústria. Esta análise justifica-se na medida em que uma arquitetura baseada em microsserviços tem bastantes pontos de exposição que podem ser explorados e ter consequências catastróficas para os serviços que se pretende assegurar.

6.1 Segurança em arquiteturas de microsserviços

Tecer considerações sobre segurança em microsserviços é uma tarefa inerentemente complicada e em relação à qual é dificil atingir a completude. Os objectivos clássicos de segurança são a confidencialidade e integridade dos dados, autenticação de entidades e mensagens, autorização e disponibilidade do sistema [40].

A segurança dos microsserviços é um problema multifacetado e depende fortemente das tecnologias subjacentes e do ambiente. Para chegar ao fundo do problema, precisamos de decompor a segurança dos microsserviços nas suas diversas componentes [40].

6.1.1 Componentes da segurança em microsserviços

As componentes que se distinguem nos microsserviços em termos das ameaças à segurança são: *hardware*, virtualização, *cloud*, comunicação, serviço e a orquestração ou *deployment*.

O *hardware* está normalmente escondido pelas camadas de abstração, no entanto, a sua confiabilidade e segurança não podem ser descuradas. Problemas e mau funcionamento de *hardware* são extremamente perigosos porque comprometem a disponibilidade dos mecanismos implementados em todos os outros componentes. Uma falha de segurança que pode ocorrer é a instalação de *backdoors* durante a manufaturação do componente de *hardware* [40].

A virtualização configura-se um pouco como um meio de mitigação de potenciais problemas de segurança. O ambiente em que uma aplicação ou serviço é lançado naturalmente afeta as condições de segurança da mesma. O sistema operativo oferece pouca separação entre serviços e processos pelo que máquinas virtuais ou *containers* conferem uma muito maior proteção relativamente a serviços mal comportados ou, porventura, comprometidos. Os tipos de ameaças mais comuns, mesmo até em ambientes virtualizados são: comprometimento do *hypervisor* e ataques de memória partilhada [40]. Uma vertente que gera também preocupação é a proveniência das imagens utilizadas quer em máquinas virtuais quer em *containers* que podem, se não se tiver o cuidado de verificar a origem, ser provenientes de um fornecedor malicioso [40].

O ambiente de *cloud* apresenta-se também como uma preocupação dado que representa também um conjunto de preocupações incluíndo o controlo alargado que os fornecedores deste tipo de serviço tendem a ter sobre tudo o que corre nos serviços que albergam [40]. Este tipo de controlo implica uma grande relação de confiança que quando quebrada pode ter graves consequências [40].

A comunicação entre serviços é outra componente da segurança em microsserviços que inspira grandes preocupações dado que informações vitais do serviço e dos utilizadores podem ser subvertidas ou manipuladas. Neste departamento os ataques mais comuns são os ataques à pilha protocolar da rede e aos protocolos utilizados para a comunicação (*SOAP*, *RESTful*). Os tipos de ataques mais comuns são *sniffing* de pacotes, *spoofing* de identidades, *Denial of Service (DoS)* e *Man-in-the-Middle (MITM)* [40].

Um outro potencial ponto de falha é o serviço ou a aplicação propriamente dita. Existem problemas que ainda são perigosamente comuns como falhas relacionadas com *SQL injection*, controlo de acesso e autenticação mal configuradas, exposição de informação sensível, *Cross-Site Scripting (XSS)* e uma má configuração de forma geral no que à segurança diz respeito [40].

Por último, é importante perceber quais os principais perigos de um *deployment* pouco cuidadoso e mal gerido, ou seja, a coordenação e automatização de tarefas relacionadas com o lançamento dos serviços pode ser um potencial ponto de comprometimento da segurança do sistema como um todo. As redes de microsserviços necessitam de serviços de *DNS* para que determinados serviços possam localizar outros - resolução de nomes. Ataques neste componente normalmente configuram-se como comprometimento do serviço de resolução de nomes através da introdução de entradas relativas a nós maliciosos do sistema forçando a que a comunicação seja estabelecida com estes [40].

A decomposição apresentada demonstra que existem diversas decisões relacionadas com segurança que devem ser feitas em cada um dos componentes da estrutura elencados. Desta forma, a arquitetura do sistema não deve ser vista como um componente singular que se pretende que seja impenetrável, deve assegurar-se que cada um dos componentes constituintes implementa todos os mecanismos de proteção desejáveis. É, no entanto, percetível que o esforço na proteção de cada um dos componentes não seja simétrico uma vez que para um desenvolvedor é mais efetivo em termos de tempo tentar implementar proteções ao nível aplicacional e de comunicação com outros serviços do que ao nível do sistema operativo, por exemplo.

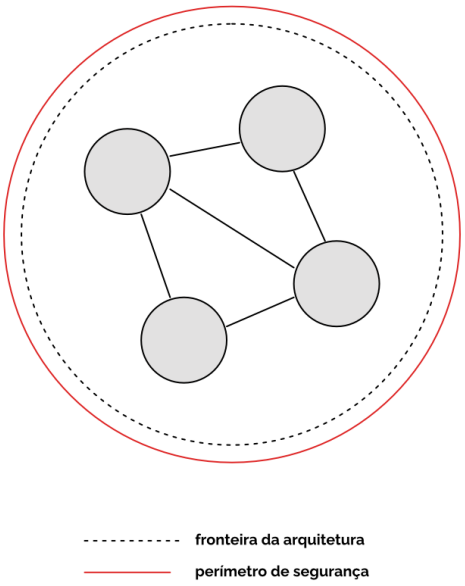


Figura 19: *Perimeter defense*

6.1.2 Definição e redefinição do perímetro de segurança

Até à relativamente pouco tempo *perimeter defense* era a abordagem mais comum no desenvolvimento de mecanismos de segurança em arquiteturas baseadas em microsserviços. No entanto, numa perspetiva mais moderna esta abordagem é considerada insuficiente - é mais útil e promove uma melhor cultura uma abordagem em que se assume que cada um dos serviços com os quais um serviço comunica está comprometido ou é mal intencionado, dado que assim temos um modelo em que o comprometimento de um serviço não implica necessariamente o comprometimento dos restantes. Desta forma assistiu-se a uma mudança no conceito de defesa: passou-se de *perimeter defense* para *defense in depth* - defesa em profundidade. A defesa em profundidade é, no fundo, um conceito que defende aprovisionar diversos mecanismos de defesa e que, por vezes, se sebropõe em diferentes níveis e que fortificam a arquitetura [40].

A segurança em microsserviços é, no fundo, um compromisso entre minimizar o orçamento e cobrir o máximo de superfícies possíveis que estão possivelmente expostas a ataques [40]. No fundo, assumir que quando desenhamos uma arquitetura de microsserviços temos controlo sobre todos os componentes abordados é irreal, quer por razões de conhecimento quer por razões de tempo. Por isto mesmo, ou seja, por uma questão de simplicidade tende a assumir-se que, por exemplo, o *hardware* e os fornecedores de serviços de *cloud* são confiáveis e não estão comprometidos [40]. Deste modo, a discussão das implicações de segurança do desenho de microsserviços centra-se nas categorias que são aceitáveis para o típico engenheiro de *software*.

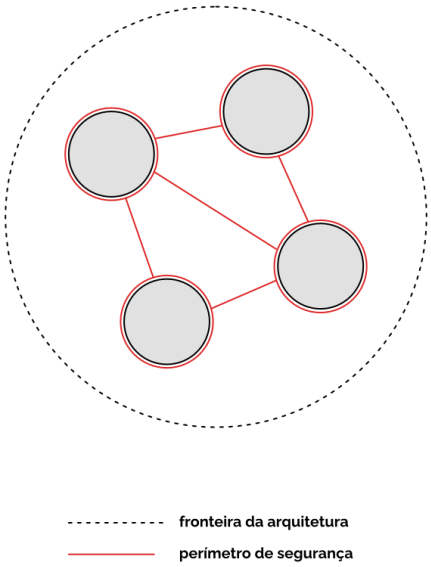


Figura 20: *Defense in depth*

6.1.3 Implicações de segurança no desenho de microsserviços

A revisão de literatura relativa a práticas de segurança em microsserviços revela que após a análise cuidadosa de diversas arquiteturas de microsserviços, é possível estabelecer uma série de princípios que devem ser respeitados no desenho destas arquiteturas de modo a reduzir a probabilidade de ocorrência de falhas de segurança. De seguida enumeram-se os princípios mencionados e explica-se as razões pelas quais configuram práticas desejáveis no desenho de *deployments* de microsserviços.

a. *Do one thing and do it well:* as propriedades do limite de contexto, da conceção em torno dos conceitos do modelo de negócio aliados ao *Domain Driver Design (DDD)* fazem com que se tenham, normalmente, em *codebases* mais pequenas. Consequentemente temos aplicações com menos linhas de código. Aliando isto ao facto de o número de falhas de segurança em aplicações ser diretamente proporcional ao número de linhas de código, percebemos que estamos a minimizar a probabilidade de ocorrência de erros que originem falhas de segurança. Uma aplicação mais pequena e mais simples faz com que seja mais fácil de compreender por todos aqueles que a mantêm, minimizando, deste modo, a probabilidade de ocorrência de falhas de segurança [40].

b. *Automated, immutable deployment:* os serviços devem ser imutáveis, isto é: alterações nos serviços devem implicar que seja criada uma nova versão dos mesmos e devem ser lançados novamente. A imutabilidade destes serviços aumenta a sua segurança dado que alterações maliciosas a um dos serviços é extremamente improvável de se propagar. A automatização de processos tende a ajudar a manter as propriedades elencadas [40].

c. *Isolation through Loose Coupling:* tanto arquiteturas *SOA - Service Oriented Architectures* como

orientadas a microsserviços são construídas com base no conceito de *loose coupling*, ou seja, os componentes dependem tão pouco dos outros quanto possível. Os microsserviços devem levar este conceito ainda um pouco mais longe através do princípio de não partilha de dados e posse total dos mesmos. Isto implica que cada serviço pode ser completamente isolado e comunicar com os outros de acordo com a sua necessidade. Mais uma vez, esta medida limita o dano no caso de um componente. ser comprometido [40]

d. *Diversity through System Heterogeneity:* a arquitetura em microsserviços permite que cada serviço possa ser escrito numa linguagem de programação diferente e que se recorra às tecnologias que melhor servem cada componente. Estas possibilidades levam a que normalmente se assista a uma grande diversidade de tecnologias num ambiente de microsserviços [40]. No fundo, a heterogeneidade que é possibilitada pelo ambiente de microsserviços faz com que a possibilidade de uma intrusão que funcionou para determinado microsserviço não funcione para todos os outros [40].

e. *Fail fast:* Em contraste com arquiteturas monolíticas em que as falhas são muitas vezes totais, sistemas distribuídos podem ser caraterizados por falhas parciais que apenas acontecem em alguns nós. Um microsserviço deve tolerar a presença de falhas parciais e limitar a sua propagação [40].

6.1.4 Práticas de segurança emergentes em microsserviços

Ainda que existam poucas práticas de segurança na indústria existem algumas tendências que se destacam. A primeira é a utilização de *Mutual Transport Layer Security (MTLS)* com a utilização de chaves públicas como método de proteção de todas as comunicações entre serviços. A segunda tendência é a utilização de *tokens* e autenticação local. Ambas as abordagens no contexto de *defense in depth* [40].

a. **Autenticação mútua dos serviços utilizando *MTLS*** - a implementação desta prática que aqui é analisada é a da solução *Docker Swarm* que é uma ferramenta de orquestração de *containers* que permite construir sistemas distribuídos. Neste caso *MTLS* é utilizado por todos os nós do *swarm* para se autenticarem entre si, encriptar todo o tráfego das suas comunicações e distinguir entres nós *slave* e *master*. Um novo nó *master* gera uma *Certificate Authority (CA)* juntamente com um par de chaves. Um *token* é gerado a partir de uma *hash* da *Certificate Authority (CA)* e de um segredo aleatório gerado. Este *token* tem de ser fornecido a todos os nós aquando do *deployment*. Para se juntar a um *swarm* um nó verifica a identidade do *master* com base no *token*, gera um certificado preliminar e manda um pedido de verificação desse certificado ao *master* juntamente com o *token*. Depois de verificar o segredo no *token* o *master* emite o certificado permamente para o novo nó. Autenticados os nós, quando precisam de se conectar com outros nós autenticam-se entre si criam um tunel *TLS* utilizando os certificados que a esta altura ambos possuem [40].

b. **Propagação da autentição através de *tokens*** - normalmente um utilizador autentica-se na *gateway* da aplicação, isto é, na faceta com a qual ele comunica. No entanto, uma aplicação é composta por diversos microsserviços e estes devem estar cientes do estado de autenticação do utilizador, isto é, se está ou não autenticado e quais as autorizações que possui no contexto de autenticação [40].

   1. *Security tokens and relevant standards*: A autenticação baseada em *tokens* é um mecanismo muito bem conhecido na engenharia de *software* que se baseia em mecanismos criptográficos que contém objetos chamados *tokens* de segurança. Quando um cliente valida com sucesso a sua sessão é criado um *token* do lado do servidor e fornecido ao cliente para uso posterior. A partir deste momento, e durante o período de validade do *token*, este substitui as credenciais do cliente e autentica os seus pedidos [40].
   2. *Reverse Security Token Service*: Uma prática de tendência crescente é a utilização de *tokens* para a autenticação de pedidos dentro da rede de microsserviços. Este tipo de abordagem em que cada serviço está preparado para testar a validade dos *tokens* permite transportar a identidade do utilizador assim como a sessão pelo sistema de uma maneira segura e descentralizada. A ideia principal deste mencanismo de autenticação é que depois de o utilizador se autenticar um *token* é gerado para uso interno na rede de microsserviços [40].

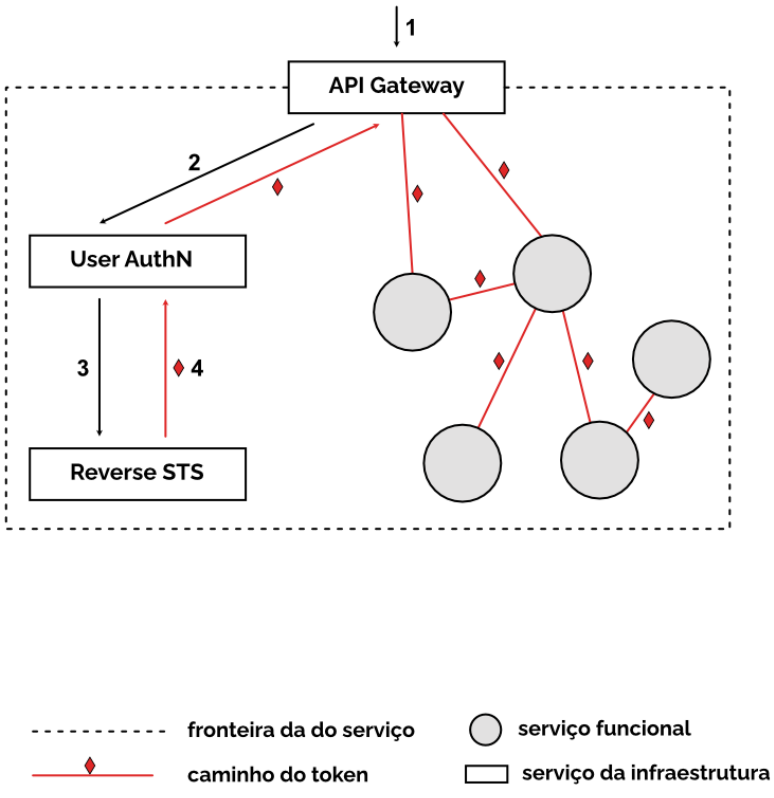


Figura 21: Esquema *Reverse STS*

c. **Fine-Grained Authorization**

1. *Security Tokens for User Authentication*: Existem vários mecanismos de controlo de acesso como *Role Based Access Control (RBAC)* e *Attribute Based Access Control (ABAC)*. O primeiro, bem como os seus antecessores, são modelos de controlo de acesso que se centram somente no utilizador e, consequentemente, não contemplam a análise entre a entidade que requisita determinado recurso e o recurso. Para um acesso mais refinado o modelo *ABAC* deve ser usado [40].
2. *Inter-Service Authorization Based on Certificates*: a utilização de certificados ao invés da autenticação foi pela primeira vez sugerida em 1999. Mais tarde, foi proposta uma alternativa para um sistema de permissão baseado em certificados para *SOA - Service Oriented Architectures*. A ideia principal é que apenas serviços que dependem uns dos outros devem conseguir comunicar entre si. Se a rede através da qual os serviços se comunicam já implementam *MTLS*, então pode ser criado um novo certificado assinado por cada tipo de serviço. Estes certificados serão utilizados para assinar os certificados das instâncias de cada tipo de serviço [40].

6.2 Aplicação das soluções emergentes no *ZTP*

Com todas as preocupações de segurança elencadas fica bastante óbvio que o *ZTP* deve considerar a implementação de, pelo menos, alguns dos mecanismos de segurança elencados.

Relembremos que a não implementação de mecanismos de segurança incrementa de forma praticamente incomensurável o risco de ocorrerem ataques que provoquem indisponibilidade do serviço, um funcionamento incorreto do mesmo ou até mesmo a divulgação de informação sensível guardada na base de dados ou relativa ao modo de funcionamento do serviço.

É útil conjeturar o tipo de ataques a que o serviço estaria sujeito no caso de não serem implementados mecanismos de segurança. Um utilizador mal intencionado poderia, por exemplo, criar uma série de pedidos que manteriam o sistema ocupado e indisponível para tratar os pedidos reais que dizem respeito a dispositivos que existem, de facto, na rede. Os ataques poderiam também ter a intenção de simplesmente fazer *trace* do caminho percorrido pelos pedidos e assim perceber qual a arquitetura em que sustenta o funcionamento do serviço de modo a prepara futuros ataques.

Deste modo, de acordo os resultados da pesquisa apresentados em 6.1.4 a sugestão apresentada para implementação de segurança no *ZTP* é a utilização de um mecanismo de autenticação baseado em serviços de fornecimento e validação de *tokens*. A sugestão apresentada é claramente inspirada no mecanismo de *Reverse STS*. Fica também claro que a filosofia de estabelecimento do perímetro de segurança utilizada é a defesa em profundidade que, conforme explicado em 6.1.2 é mais atualizada e, de um modo geral, mais segura.

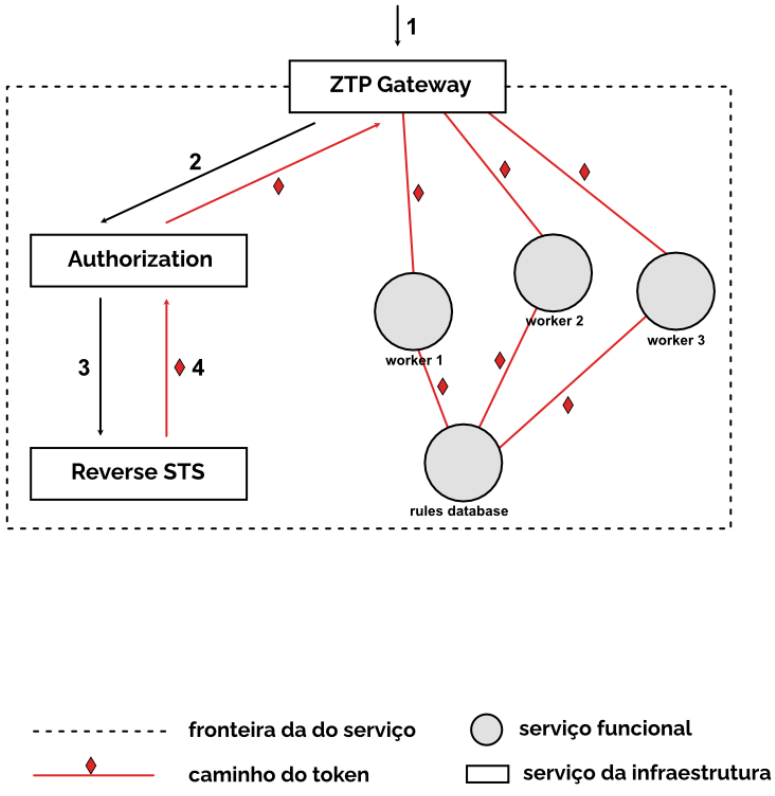


Figura 22: Esquema *Reverse STS* no *ZTP*

7

Monitorização

A insuficiência de registos de ações tem sido um grande problema de segurança nas aplicações. A monitorização adequada de eventos críticos de segurança numa aplicação pode potencialmente mitigar e até prevenir grandes violações de dados. Além disso, a eficiência do registo dos dados desempenha um papel crucial no desempenho do sistema. A eficiência dos registos implica registar apenas o necessário para análise e recuperação posteriores, em vez de recolher ingenuamente informações excessivas que impedem uma resposta atempada a incidentes de segurança [11].

As abordagens existentes para a localização de falhas em microsserviços com muita frequência apenas utilizam uma fonte de dados como, por exemplo, os *logs* do sistema operativo ou métricas de monitorização. No entanto, os sistemas podem exibir diferentes pistas em diferentes fontes de dados de monitorização. Por exemplo, altas cargas de sistema podem causar uma utilização anormalmente alta da unidade de processamento central. Para além disso, um erro num pedido a uma base de dados pode ser refletido nos *logs* do sistema enquanto que não será percetível nas métricas de monitorização [41].

Consequentemente, a monitorização é uma prática de extrema importância em engenharia de *software*. As estruturas de microsserviços existentes atualmente tendem a criar *logs* distintos por cada instância de microsserviço; para além disso, diferentes microsserviços podem encorporar diferentes formatos ou até diferentes semânticas [26].

Existem diversos trabalhos realizados com o intuito de investigar quais as melhores formas de produzir *logs* num ambiente de microsserviços e as melhores de maneiras de fazer com que estes sejam uma mais valia para o desenvolvedor e gestor da infraestrutura. **Singh** [35] desenvolveu trabalho ao nível dos *logs* em ambientes de *cluster*. Ele afirma que reunir e analisar informações dos *logs* é um fator fundamental de gerir um ambiente de produção e garantir a sua fiabilidade. O autor refere também que recolher os *logs* de componentes utilizando *containers* ou *Kubernetes* é um desafio [31].

7.1 Monitorização e *logs* no ambiente do *ZTP*

O ambiente escolhido para o *deployment* do *ZTP* foi o de microsserviços que serão corridos em *containers* num ambiente de *cluster Kubernetes*. No entanto, segundo **Satnam Singh** [35], a recolha de

logs relativos aos componentes de um *cluster* é mais desafiante porque já não existe um programa ou servidor específico que implemente toda a lógica de realização desta tarefa. Isto acontece dado que cada componente da estrutura consiste num número dinâmico de instâncias anónimas cujo número depende da carga no sistema a cada altura. Assim sendo, o artigo referenciado serve de orientação à conceção de uma solução de *logging* para o *ZTP*.

7.1.1 Kubernetes

O entendimento da solução descrita no artigo referenciado é dependente do entedimento de alguns conceitos básicos de *Kubernetes*. Com efeito, efetuou-se alguma pesquisa no que diz respeito aos conceitos fundamentais da ferramenta e deixam-se aqui os seus resultados como auxílio à compreensão da solução apresentada.

A responsabilidade primária do *Kuberntes* é a orquestração de *containers*. Isto significa garantir que todos os *containers* que realizam as mais variadas tarefas estão escalonados para correr em máquinas físicas ou virtuais. (...) Para além disto, tem de monotorizar todos os *containers* que estão a correr a determinado momento e subsituir os mortos, os que não respondem ou, por algum motivo, falharam, por novas instâncias [32].

Seguem-se agora alguns conceitos da arquitetura de *Kubernetes* juntamente com a sua explicação:

1. **Cluster** - é uma colecção de recursos de armazenamento e de rede que o *Kubernetes* utiliza para executar várias cargas de trabalho que compõem o sistema. Note que todo o sistema pode consistir em múltiplos *clusters* [32].

2. **Pod** - é uma unidade de trabalho em *Kubernetes*. Cada *pod* contém um ou mais *containers*. Os *pods* são sempre programados em conjunto (funcionam sempre na mesma máquina). Todos os *containers* de um *pod* têm o mesmo endereço IP e porta; podem comunicar utilizando a comunicação `localhost` ou inter-processo padrão. Além disso, todos os *containers* de um *pod* podem ter acesso ao armazenamento local partilhado no nó que aloja o *pod*. O armazenamento partilhado será montado em cada *container*. Os *pods* são um conceito importante de *Kubernetes*. É possível executar múltiplas aplicações dentro de um único *container Docker*, tendo algo como supervisor como principal aplicação *Docker* que executa múltiplos processos, mas esta prática é muitas vezes reprovada, pelas seguintes razões:

   - **Transparência** - Tornar os *containers* dentro do *pod* visíveis para a infra-estrutura permite que a mesma preste serviços a esses *containers*, tais como gestão de processos e monitorização de recursos. Isto facilita uma série de conveniências para os utilizadores.
   - **Desacoplamento de dependências de *software*** - Os *containers* individuais podem ser versionados, reconstruídos e redistribuídos de forma independente. O *Kubernetes* pode mesmo vir a suportar actualizações ao vivo de *containers*.

- **Facilidade de utilização** - Os utilizadores não precisam de executar os seus próprios gestores de processos, preocupar-se com a propagação de sinais e códigos de saída.
- **Eficiência** - Como a infra-estrutura assume maior responsabilidade, os *containers* podem ser mais leves.

Os *pods* fornecem uma solução para gerir grupos de *containers* estritamente relacionados que dependem uns dos outros e precisam de cooperar com o mesmo *host* para cumprirem o seu propósito. É importante lembrar que os *pods* são considerados entidades efémeras e descartáveis que podem ser descartadas e substituídas a qualquer momento. Qualquer armazenamento de *pods* é destruído com o seu *pods*. Cada *pod* recebe uma identificação única (UID), pelo que ainda se pode distinguir entre eles, se necessário [32].

3. **Service** - Os serviços são utilizados para expor alguma funcionalidade aos utilizadores ou outros serviços. Normalmente englobam um grupo de *pods*, normalmente identificados por uma *label*. Existem serviços que fornecem acesso a recursos externos, ou a *pods* que controla directamente ao nível de IP virtual. Os serviços *Kubernetes* nativos são expostos através de *endpoints* convenientes. Os serviços são publicados ou descobertos através de um de dois mecanismos: DNS, ou variáveis de ambiente [32].

Desta forma, quando trabalhamos com *Kubernetes* não devemos focar-nos em parâmetros como o nome da máquina virtual em que um determinado serviço é colocado visto que isso pode, muito rapidamente, deixar de se verificar. Para além disso, os componentes são escaláveis através da especificação do número de réplicas pelo que determinado componente pode estar em mais que uma máquina virtual [35].

Em *Kubernetes* é muito mais significativo e prático identificar partes do sistema fazendo *queries* sobre os *labels* que estão associados às entidades anónimas criadas pelo orquestrador do *Kubernetes* que retornem as entidades que satisfazem as condições. Deste modo, temos condições para pensar sobre uma infra-estrutura dinâmica sem termos de mencionar nomes de recursos específicos [35].

7.1.2 Monitorização no *ZTP*

A aplicação em questão tem uma arquitetura constituída por quatro componentes funcionais diferentes: *broker*, *worker*, *rule matcher* e *rules database*, tal como fica claro na Figura 23.

Conforme foi já explicado em 7.1.1 os *pods* são a unidade básica de um *deployment* em *Kubernetes*. Sendo assim é necessário definir os *pods* necessários para o funcionamento da nossa aplicação.

Para começar, a nossa aplicação utiliza um *pod* para o *deployment* da instância que albergará o nosso *container* de *PostgreSQL* que armazena as regras do *ZTP*.

Esta é a especificação desse *pod*:

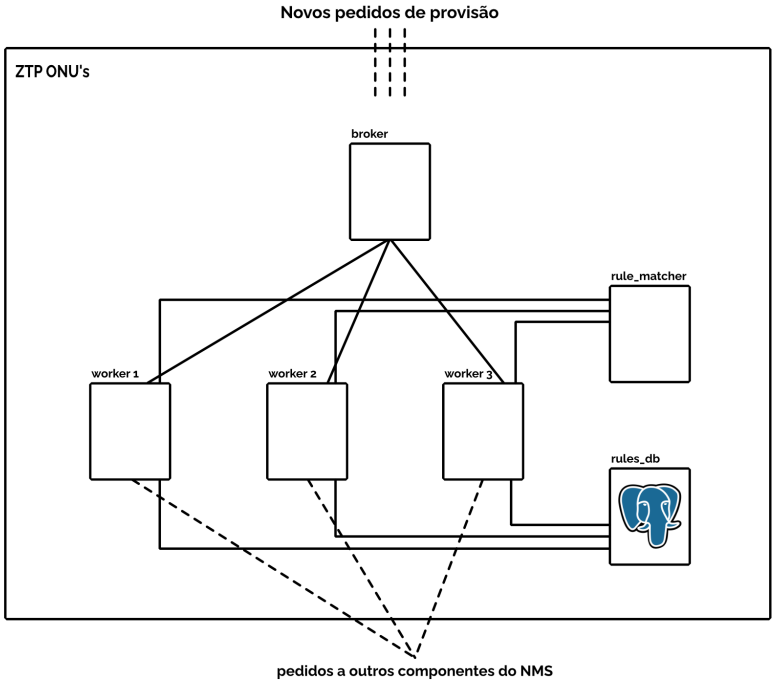


Figura 23: Arquitetura do *ZTP*

rules-db-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: rules-db
  labels:
    app: ztp
    role: db
    tier: backend
spec:
  containers:
  - name: rules-db
    image: private-domain:private-port/ztp-onu-rules-database:1.0
    ports:
    - containerPort: 5432
    volumeMounts:
    - name: psql-persistent-storage
```

```
      mountPath: /var/lib/psql
  volumes:
    - name: psql-persistent-storage
      gcePersistentDisk:
        pdName: rules-disk
        fsType: ext4
```

Neste momento os *logs* gerados pelo *pod* dizem respeito ao *container* que estiver a correr no momento em que os mesmos forem consultados. O que é necessário é providenciar um mecanismo que permita agregar os *logs* de todos os *containers* que correm como parte do ciclo de vida de um *pod*. Não esqueçamos também que é comum em *Kubernetes* especificar um controlador de réplicas que cria réplicas de um *pod*. A consistência e agregação dos *logs* tem de ser garantida também nesses casos.

Especifiquemos então a especificação de um controlador de réplicas para as réplicas de *workers* no nosso *deployment*:

workers-replication-controller.yaml

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: worker-slave
  labels:
    app: ztp
    role: slave
    tier: backend
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: ztp
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: private-domain:private-port/ztp-onu-worker:1.0
        resources:
          requests:
            cpu: 400m
```

```
            memory: 250Mi
        ports:
        - containerPort: 8080
```

Precisamos agora de definir um serviço para os *workers* por forma de termos uma forma perene de os identificar.

workers-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: workers-slave
  role: slave
  tier: backend
spec:
  ports:
  - port: 8080
  selector:
    app: ztp
    role: slave
    tier: backend
```

No entanto, mesmo que consigamos recolher os *logs* de todos os *containers* num *pod* continuamos com o problema de os mesmos serem dinâmicos. Por isso precisamos de recolher *logs* com base no ciclo de vida do controlador e não de um *pod* específico. Neste cenário a utilização de *Fluentd* [20] configura-se como uma solução para o problema.

Fluentd é utilizado para recolher os *logs* de todos os *containers Docker* que correm num determinado nó. Os coletores do *Fluentd* não guardam eles próprios os *logs*. Em vez disso, enviam-nos para um *cluster* de *Elasticsearch* [17] que guarda a informação num conjunto de nós replicados.

Fluentd é um coletor de dados *open source*, que permite unificar a recolha e o consumo de dados para uma melhor utilização e compreensão dos dados obtidos [20].

Desta forma, a especificação do *Fluentd* ao nível dos *pods* é a seguinte:

elasticsearch-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
spec:
  containers:
  - name: fluentd-elasticsearch
    image: gcr.io/google_containers/fluentd-elasticsearch:1.11
    resources:
      limits:
        cpu: 100m
    args:
    - -q
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
  terminationGracePeriodSeconds: 30
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: varlibdockercontainers
    hostPath:
      path: /var/lib/docker/containers
```

O esquema da Figura 24 é uma representação de um *deployment* que garanta os princípios de *logging* e monotorização que se pretende garantir. Essencialmente o objetivo é ter *Fluentd* a correr em cada *container*, dentro de cada *pod*, que reporta à instância de *Elasticsearch*. Deste modo, os *logs* de cada uma das aplicações ficam salvaguardados e centralizados com uma *interface* de consulta que facilita quer a sua agregação quer a interpretação e consulta dos mesmos.

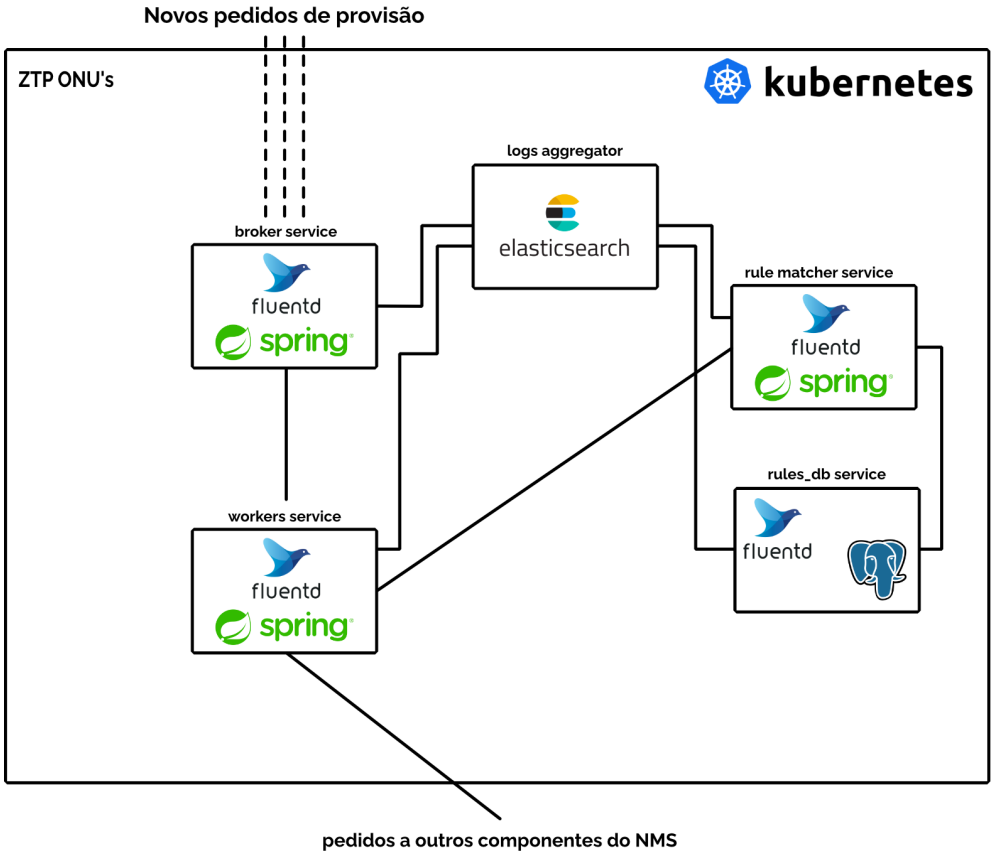


Figura 24: Arquitetura de *logging* do *ZTP*

Este capítulo visa propor uma arquitetura que permitisse uma monitorização consistente e coerente dos *logs* produzidos pelos serviços. O propósito não é propor uma implementação, razão pela qual as especificação dos serviços, *pods* e controladores de *Kubernetes* são meramente representativos. Com a arquitetura proposta garantimos as propriedades referidas dos *logs* e uma maneira centralizada de os consultar através da instância de *Elasticsearch*.

8

Conclusão

O trabalho realizado incidiu essencialmente sobre o estudo da arquitetura atual das redes de acesso, as ferramentas que existem para realizar a manutenção das mesmas e ter acesso ao seu estado a cada momento. Deste modo, ficou claro qual o contexto em que se insere este trabalho e qual a mudança positiva que ele representa como uma solução desenvolvida num contexto específico mas que é facilmente aplicável a qualquer sistema de manutenção e gestão de redes. Toda esta informação foi obtida com recurso a artigos científicos que foram referenciados, documentação pública do **AGORA**, informação disponível em *blogs* e artigos divulgados por vendedores de equipamentos tipicamente focados nas redes de acesso. Foi feita uma pequena introdução ao tema das **Redes PON** como forma de proporcionar ao leitor um eventual primeiro contacto com a realidade dos sistemas de gestão de redes que são mais comuns atualmente. No entanto, um entendimento profundo desse tema não foi necessário quer para o desenvolvimento deste trabalho quer para o entendimento deste documento, uma vez que o foco é claramente na arquitetura do *software* a desenvolver assim como as ferramentas a utilizar para a sua implementação. Seguidamente, foi criada uma proposta de arquitetura do módulo de *software* a estudar. Apesar de tudo, esta proposta não ilustrava preocupações em termos de escalabilidade e segurança em virtude de apenas visar apresentar uma visão geral de como o novo componente (o *ZTP*) comunicará com os componentes de *software* já existentes no sistema de gestão de redes da **AlticeLabs**.

A proposta de arquitetura mencionada foi depois desenvolvida e concretizada em quatro variações por forma a se poder aferir qual delas representa a opção ótima num cenário com as restrições descritas. Com o objetivo de fazer esta análise foi desenvolvido um ambiente de simulação que permitiu obter resultados em termos do tempo de aprovisionamento e os *timeouts* ocorridos durante os aprovisionamentos simulados. Este ambiente de simulação foi, de facto, a componente mais prática do trabalho desenvolvido e foi o que permitiu alcançar o maior objetivo deste trabalho: recomendar uma arquitetura para o módulo *ZTP ONU's* com base numa investigação minuciosa que garanta o cumprimento dos requisitos de escalabilidade e implementação da concorrência no aprovisionamento.

Seguidamente, foi pensado um sistema que permitisse associar um equipamento a uma das regras de aprovisionamento disponíveis no momento em que ele aparece no *NMS*. O resultado desta componente foi o esboço de um sistema de filtros que permite fazer uma correspondência, ainda que parcial, das regras

com o novo equipamento. Esta componente do *ZTP ONU's* é crucial dado que implementa automatização na escolha do *playbook Ansible* a correr no momento do aprovisionamento.

Como o módulo a desenvolver lida com informação sensível relativa a serviços de cliente entre outros tipos de informação, a segurança foi uma preocupação. Neste sentido, foi realizado um estudo das práticas de segurança que devem ser implementadas em ambientes de microsserviços e, consequentemente, apresentada uma proposta de arquitetura do módulo que conta já com essas preocupações.

Por fim, como a *traceability* e monitorização são preocupações igualmente importantes foi realizada pesquisa nesse âmbito e, mais uma vez, apresentada uma proposta de arquitetura que permite implementar *logging* no módulo, ou seja, contar com um mecanismo que assegura a persistência desses *logs* assim como a delegação da função recolha dos mesmos a *software* especializado.

Um dos próximos passos no contexto do *ZTP ONU's* é desenvolver, de facto, os microsserviços que asseguram as funções delegadas a cada um dos componentes da arquitetura apresentada. Foram já estudadas todas as componentes que se apontaram como objeto de estudo dado que este é um dos primeiros projetos num ambiente de microsserviços no contexto da empresa, razão que justificou a necessidade de todo este trabalho de investigação.

Como fica claro, este trabalho tem diversas vertentes que podem ser exploradas, mas aqui o principal foco foi desenhar um sistema que esteja preparado para receber novas funcionalidades e novos tipos de pedidos ao longo do tempo e esteja bem preparado para essa evolução.

8.1 Trabalho futuro

Conforme foi já estabelecido, a incidência deste trabalho foi essencialmente no desenho da arquitetura da solução a desenvolver e no desenvolvimento da pesquisa necessária para tornar essa solução segura e cumpridora dos requisitos de desempenho. No entanto, este foco mais teórico e de investigação neste trabalho faz com que as tarefas que restam no desenvolvimento do módulo *ZTP ONU's* sejam do domínio mais prático, ou seja, a implementação propriemente dita dos módulos integrantes da solução que foram sendo referidos e que se especificam a seguir.

No entanto, a implementação destes módulos requer ela própria pesquisa no sentido em que estes comunicam com outros módulos existentes no *NMS*, nomeadamente através de *REST API's* que devem ser estudadas. Mantenha-se também presente que até agora foi apenas considerada a notificação respetiva a um novo *ONU*, no entanto, o módulo foi pensado para ter maior versatilidade e, por isso, receber notificações de mais tipos. Estes diferentes tipos de alarmes darão origem a tipos de ações diferentes, podendo até ser configurados diferentes conjuntos de *playbooks* para cada um dos diferentes tipos de notificações, por exemplo. Com efeito, faz todo o sentido, realizar um levantamento de todos os eventos que apresentem relevância e que possam despoletar ações no módulo desenvolvido.

O processo que foi sendo mencionado como *broker*, tendo em conta a arquitetura que foi escolhida, tem a função de encaminhar mensagens para um dos *workers* de forma a minimizar o seu tempo de

processamento e a probabilidade de que a mesma incorra num *timeout*. Desta forma, é necessário implementar este módulo e incluí-lo no fluxo das mensagens do *ZTP*, isto é, as mensagens começarem a serem enviadas para o *broker* que posteriormente se encarrega do seu encaminhamento. De igual modo, o processo que se domina de *worker* tem também de ser consubstanciado num serviço que se encarrega de levar a cabo as ações que asseguram a aplicação de todas as ações que compõem os *playbooks*. Conforme foi já deixado claro em várias ocasiões ambos os serviços serão implementados em *Java* com recurso à *framework Spring Boot*.

Após o desenvolvimento e codificação de cada um destes serviços é necessário implementar a arquitetura total do sistema que engloba a implementação dos mecanismos de mitigação de possíveis problemas de segurança abordados no Capítulo 6, assim como assegurar os mecanismos de *logging* com o *deployment* apresentado no Capítulo 7 num ambiente de *deployment* em microsserviços como, por exemplo, um *cluster Kubernetes*.

Depois de implementados os serviços e a arquitetura escolhida é necessário validar quer a funcionalidade quer o desempenho do sistema. Devem ser desenvolvidos testes que permitam validar a funcionalidade do *ZTP* quer o desempenho perante diferentes cenários de carga antes de ser lançada uma versão para produção.

Bibliografia

[1] url: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (ver p. xvi).

[2] url: https://docs.rudder.io/reference/6.1/index.html (ver p. 8).

[3] url: https://puppet.com/use-cases/continuous-configuration-automation/ (ver p. 9).

[4] url: https://puppet.com/docs/puppet/7/puppet_language.html (ver p. 9).

[5] url: https://github.com/asim/go-micro (ver p. 13).

[6] url: https://moleculer.services/ (ver p. 13).

[7] H. Abbas e M. Gregory. "The next generation of passive optical networks: A review". Em: *Journal of Network and Computer Applications* 67 (mar. de 2016). doi: 10.1016/j.jnca.2016.02.015 (ver p. 4).

[8] "ABC of PON: Understanding OLT, ONU, ONT and ODN". Em: url: https://community.fs.com/blog/abc-of-pon-understanding-olt-onu-ont-and-odn.html (ver pp. xv, 6).

[9] "About the Charmed Operator Framework and Charmhub". Em: url: https://juju.is/about (ver p. 10).

[10] "Agora - Network Managment System". Em: url: https://www.alticelabs.com/site/gpon-solutions/network-management-system/ (ver p. 5).

[11] S. Amir-Mohammadian e A. Y. Zowj. "Towards Concurrent Audit Logging in Microservices". Em: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2021, pp. 1357–1362. doi: 10.1109/COMPSAC51774.2021.00191 (ver p. 57).

[12] "ANTLR". Em: url: https://www.antlr.org/ (ver p. 46).

[13] "APG". Em: url: https://https://sabnf.com/ (ver p. 46).

[14] A. Barabanov e D. Makrushin. *Security audit logging in microservice-based systems: survey of architecture patterns*. Fev. de 2021 (ver p. 14).

[15] "BYACC". Em: url: http://byaccj.sourceforge.net/ (ver p. 46).

[16] O. Al-Debagy e P. Martinek. "A Comparative Review of Microservices and Monolithic Architectures". Em: mai. de 2019 (ver p. 11).

[17] "Elasticsearch". Em: url: https://www.elastic.co/?ultron=B-Stack-Trials-EMEA-S-Exact&gambit=Stack-Core&blade=adwords-s&hulk=paid&Device=c&thor=elasticsearch&gclid=Cj0KCQjwwJuVBhCAARIsAOPwGAQkybNJX7wnT2DLrt_PKpMV6qgjIYjtN2R0p2nMi296TOXx5X2-cWYaAlgLEALw_wcB (ver p. 62).

[18] *Encyclopaedia Britannica*. 2017-12-27 (ver p. xv).

[19] "FCAPS". Em: Wikipedia. url: https://en.wikipedia.org/wiki/FCAPS (ver pp. vii, ix).

[20] "Fluentd". Em: url: https://fluentd.org (ver p. 62).

[21] M. Fowler. "Microservices". Em: url: https://martinfowler.com/articles/microservices.html (ver p. 11).

[22] *Getting Started*. url: https://www.dropwizard.io/en/latest/getting-started.html (ver p. 13).

[23] R. Gnatyk. "Microservices vs Monolith: which architecture is the best choice for your business?" Em: url: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (ver p. 11).

[24] P. Goyal, R. Mikkilineni e M. Ganti. *FCAPS in the Business Services Fabric Model*. Jun. de 2009. doi: 10.1109/WETICE.2009.21 (ver p. 1).

[25] J. Hehmann e T. Pfeiffer. "New monitoring concepts for optical access networks". Em: vol. 13. 1. 2008, pp. 183–198. doi: 10.1002/bltj.20290 (ver pp. 5, 21).

[26] J. Hehmann e T. Pfeiffer. "New monitoring concepts for optical access networks". Em: *Bell Labs Technical Journal* 13.1 (2008), pp. 183–198. doi: 10.1002/bltj.20290 (ver p. 57).

[27] "Juju". Em: url: https://juju.is/ (ver p. 9).

[28] S. M. .-. S. A. Manager. "White Paper - Gigabit Passive Optical Networks". Em: (ver p. 5).

[29] N. Mateus-Coelho. "Security in Microservices Architectures". Em: jan. de 2020 (ver pp. 11, 12).

[30] "Red Hat Ansible Automation Platform". Em: url: https://www.redhat.com/en/technologies/management/ansible (ver p. 9).

[31] B. Saman. "Monitoring and analysis of microservices performance". Em: *Journal of Computer Science and Control Systems* 10.1 (2017), p. 19 (ver p. 57).

[32] G. Sayfan. *Mastering kubernetes*. Packt Publishing Ltd, 2017 (ver pp. 58, 59).

[33] "Scaling Horizontally vs. Scaling Vertically". Em: url: https://www.section.io/blog/scaling-horizontally-vs-vertically/ (ver p. 15).

[34] K. Sen. “Top 10 configuration management tools you need to know about”. Em: ago. de 2021. url: https://www.upguard.com/blog/configuration-management-tools (ver p. 8).

[35] S. Singh. “Cluster-level Logging of Containers with Containers: Logging Challenges of Container-Based Cloud Deployments”. Em: *Queue* 14.3 (2016), pp. 83–106 (ver pp. 57, 59).

[36] A. Valdar. *Understanding Telecommunications Networks*. 2006 (ver p. xvi).

[37] *What is Multiprotocol Label Switching (MPLS)?* url: https://www.forcepoint.com/cyber-edu/mpls-multiprotocol-label-switching (ver p. xv).

[38] “What is Network Managment?” Em: CISCO. url: https://www.cisco.com/c/en/us/solutions/enterprise-networks/what-is-network-management.html (ver p. 1).

[39] “Why Spring?” Em: url: https://spring.io/why-spring (ver p. 12).

[40] T. Yarygina e A. H. Bagge. “Overcoming Security Challenges in Microservice Architectures”. Em: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 2018, pp. 11–20. doi: 10.1109/SOSE.2018.00011 (ver pp. 49–55).

[41] Q. Zhang et al. “Fault Localization for Microservice Applications with System Logs and Monitoring Metrics”. Em: *2022 7th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. 2022, pp. 149–154. doi: 10.1109/ICCCBDA55098.2022.9778893 (ver p. 57).