Mestrado em Engenharia Informática

Estágio

Relatório Final

2014/2015

Implementação de algoritmo de escalabilidade para DNS em contexto *cloud*

João Manuel Marques do Santos
josantos@student.dei.uc.pt

Orientador DEI:
Professor Doutor Edmundo Monteiro

Orientadores OneSource:
Doutor Bruno Sousa
Doutor David Palma

2 de setembro de 2015

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Implementação de algoritmo de escalabilidade para DNS em contexto cloud

Relatório submetido no âmbito do estágio curricular do Mestrado em Engenharia Informática

2014/2015

Autor: João Manuel Marques dos Santos

Orientador DEI: Professor Doutor Edmundo Monteio

Orientadores OneSource: Doutores Bruno Sousa e David Palma

Júri Arguente: Professor Doutor Fernando Boavida

Júri Vogal: Professor Doutor Fernando José Barros

Agradecimentos

O presente relatório representa o culminar de um percurso de estudos em Engenharia Informática, aqui como forma de cumprimento dos requisitos necessários à obtenção do grau de Mestre e para o qual foram essenciais todos os conhecimentos adquiridos ao longo deste ciclo de estudos na Faculdade de Ciências e Tecnologia da Universidade de Coimbra, mas também, desde logo, durante a Licenciatura na Escola Superior de Tecnologia e Gestão de Oliveira do Hospital do Instituto Politécnico de Coimbra. É neste momento que importa também, para além do esforço de enriquecimento pessoal e académico, demonstrar o meu reconhecimento para com todos os que contribuíram para a materialização deste Relatório de Estágio.

Expresso o meu agradecimento ao Professor Doutor Edmundo Monteiro e aos Doutores Bruno Sousa e David Palma pela orientação deste trabalho, pela competência, exigência e a compreensão do desafio, mas também pela disponibilidade, observações e sugestões que me incentivaram a trabalhar para alcançar um resultado que é o aproximado às nossas expectativas.

Agradeço também à OneSource por ter criado as condições ao Estágio que serviu de enquadramento a este Relatório, pela confiança demonstrada e a oportunidade em trabalhar num projeto desafiante com uma equipa dedicada, disponível e cujo apoio foi muito importante para este projeto.

Este percurso não seria possível sem o contínuo esforço e dedicação da pessoa a quem devo muito do que sou hoje: a minha Mãe. A ela, com um grande e especial agradecimento pelo apoio em todas as ocasiões, dedico este trabalho por 23 anos de contínua dedicação. Ao meu Pai pela força e coragem inspirada nos momentos mais adversos. À Irmã Ana por ser uma constante inspiração e por me compreender melhor do que ninguém. Ao Tio José pelo companheirismo e conselhos pertinentes. À Tia Conceição pela disponibilidade e apoio dados. Um agradecimento também à restante família, demasiado extensa para ser particularizada mas que, à sua maneira, soube contribuir para este percurso.

Resumo

O *Domain Name System* (DNS) é um protocolo de extrema importância na Internet, uma vez que muitos serviços e utilizadores dependem dos mecanismos de traduções entre nomes e endereços. Dado o seu vasto uso, são utilizadas técnicas avançadas, como distribuição de carga para maximizar a sua *performance* e possibilitar outros serviços como redes avançadas de distribuição de conteúdo

Com o paradigma *cloud computing* a assumir uma crescente relevância na gestão de serviço - permitindo ter serviços mais robustos, escaláveis e com mecanismos de *failover* – começa a verificar-se uma clara migração de alguns serviços para a *cloud* e o DNS não é exceção.

Este relatório apresenta uma proposta de solução de DNS as a Service (DNSaaS), validada no âmbito do projeto Mobile Cloud Networking. É definida e apresentada, a especificação da sua arquitetura, seguida pelo processo de desenvolvimento de um algoritmo que visa a escalabilidade do serviço. Este algoritmo assegura o desempenho do próprio serviço dentro dos níveis pré-definidos, otimizando a alocação dos recursos necessários em tempo real.

Palavras-chave: *Cloud Computing,* DNSaaS, Escalabilidade, Mobile Cloud Networking, OpenStack.

Abstract

Domain Name System (DNS) is an extremely important protocol on the Internet, since many services and users rely on translation mechanisms between names and addresses. Given its wide use, advanced techniques are used, such as load distribution to maximize its performance and enable other services such as advanced content delivery networks.

With cloud computing paradigm playing an increasing importance in service management – allowing more powerful services, scalable and failover mechanisms – it begins to notice a clear migration of some services to the cloud, and DNS is no exception.

This report proposes a DNS solution as a Service (DNSaaS), validated under the Mobile Cloud Networking. It is defined and displayed the specification of its architecture, followed by the development process of an algorithm which seeks for service scalability. This algorithm ensures the performance of the service itself within the predefined levels, optimizing the resources allocation in real time.

Keywords: Cloud Computing, DNSaaS, Scalability, Mobile Cloud Networking, OpenStack.

Índice

Capítulo 1 : Introdução ... 1

1.1. Contexto ... 1
Telecomunicações móveis ... 2
Cloud Computing ... 3
Domain Name System ... 4

1.2. Objetivos ... 5

1.3. Metodologia ... 5

1.4. Planeamento ... 6

1.5. Trabalho realizado ... 7

1.6. Estrutura do documento ... 8

Capítulo 2 : Estado da Arte ... 9

2.1. *Cloud Computing* ... 9
Modelos de Implementação e Serviços ... 10

2.2. Mobile Cloud Networking ... 11
Serviços ... 11
Ciclo de vida dos serviços ... 12
Elementos ... 13

2.3. Soluções IaaS ... 14
Amazon Elastic Compute Cloud ... 14
Microsoft Azure ... 15
Rackspace Cloud ... 16
LunaCloud ... 17
OpenStack ... 18
Comparação ... 19

2.4. OpenStack ... 20
Horizon ... 21
Keystone ... 21
Nova ... 22
Neutron ... 23
Swift ... 23
Cinder ... 23
Glance ... 24
Ceilometer ... 24
Heat ... 24
Designate ... 25

2.5. Soluções DNS ... 25
PowerDNS ... 26
BIND ... 26

2.6. Escalabilidade ... 27
Escalabilidade de serviços na cloud ... 27
Escalabilidade de DNS ... 28

2.7. Algoritmo de decisão ... 28

Capítulo 3 : *DNS as a Service* **31**

3.1. Arquitetura 32

Capítulo 4 : Validação da arquitetura **35**

4.1. Avaliação 38
BD Distribuída vs BD Centralizada 38
Número de Servidores 41

Capítulo 5 : Algoritmo de Escalabilidade **43**

5.1. Desenvolvimento 46
Características 49
Atividade do algoritmo 50

5.2. Testes 51

5.3. Análise 54
Teste 1 54
Teste 2 62
Conclusões 70

Capítulo 6 : Conclusão **73**

6.1. Trabalho Futuro 74

Apêndice A – *Template* e *Stack* **77**

Apêndice B – MeTHODICAL **81**

Lista de Figuras

Figura 1 - Arquitetura conceptual do projeto MCN ... 2

Figura 2 - Previsão do mercado dos dispositivos ... 3

Figura 3 - Previsão dos gastos nos serviços de *cloud computing* ... 4

Figura 4 - Metodologia de Projetos de Investigação ... 6

Figura 5 - Planeamento inicial ... 7

Figura 6 - Planeamento executado ... 7

Figura 7 - Componentes presentes durante o ciclo de vida do DNSaaS ... 8

Figura 8- Serviços vs Implementação ... 11

Figura 9 - Fase de negócio ... 12

Figura 10 - Fase técnica ... 13

Figura 11 - Comunicação entre os elementos e serviços no MCN ... 14

Figura 12 - Arquitetura conceptual do Openstack ... 21

Figura 13 - Arquitetura lógica do Keystone ... 22

Figura 14 - Arquitetura lógica do Nova ... 22

Figura 15 - Arquitetura lógica do Heat ... 24

Figura 16 - Arquitetura lógica do Designate ... 25

Figura 17 – Dependências do DNSaaS ... 31

Figura 18 - Arquitetura com base de dados centralizada ... 33

Figura 19 - Arquitetura com base de dados distribuída ... 34

Figura 20 - Testes com base de dados centralizada ... 35

Figura 21 - Testes com base de dados distribuída ... 36

Figura 22 - Latência no teste distribuída vs centralizada no cliente ... 39

Figura 23 - QPS no teste distribuída vs centralizada no cliente ... 40

Figura 24 - *Queries* recebidas no teste distribuída vs centralizada no cliente ... 40

Figura 25 - Latência no teste de número de servidores no cliente ... 41

Figura 26 - Estrutura do algoritmo ... 43

Figura 27 - Diagrama de classes do SO ... 47

Figura 28 - *Array* cumulativo de duas posições ... 49

Figura 29 - Falhas na métrica ... 50

Figura 30 - Diagrama de atividade do algoritmo de escalabilidade ... 51

Figura 31 - Estrutura do serviço durante os testes. 53
Figura 32 - Ramificação de SLO 54
Figura 33 - Comparação dos algoritmos durante o teste 1, relativamente ao número de PDNS 56
Figura 34 - Comparação dos algoritmos durante o teste 1, relativamente à latência 57
Figura 35 - Comparação dos algoritmos durante o teste 1, à percentagem de *answers slow* 59
Figura 36 - Comparação dos algoritmos durante o teste 1, relativamente ao CPU utilizado 60
Figura 37 - Comparação dos algoritmos durante o teste 1, relativamente às violações de SLO 62
Figura 38 - Comparação dos algoritmos durante o teste 2, relativamente ao número de PDNS 64
Figura 39 - Comparação dos algoritmos durante o teste 2, relativamente à latência 65
Figura 40 - Comparação dos algoritmos durante o teste 2, à percentagem de *answers slow* 67
Figura 41 - Comparação dos algoritmos durante o teste 2, relativamente ao CPU utilizado 68
Figura 42 -Comparação dos algoritmos durante o teste 2, relativamente às violações de SLO 70
Figura 43 - Latência nos clientes durantes testes de *performance* do serviço 71
Figura 44 – *Queries* por segundo nos clientes durantes testes de performance do serviço 72
Figura 45 - Infraestrutura do *template* submetido 80
Figura 46 - Passos do Algoritmo MeTHODICAL 81
Figura 47 - Distância de Benefícios 82
Figura 48 - Distância de Custos 82

Lista de Tabelas

Tabela 1 - Preços de instâncias na EC2 15

Tabela 2 - Preços de instâncias no Azure 16

Tabela 3 - Preços de instâncias no Rackspace 17

Tabela 4 - Preços de instâncias na LunaCloud 18

Tabela 5 - *Flavors* das instâncias do Openstack 19

Tabela 6 - Comparação entre soluções IaaS 19

Tabela 7 - Características das soluções IaaS 20

Tabela 8 - Diferenciação de técnicas de escalabilidade 28

Tabela 9 - Algoritmos de decisão 29

Tabela 10 - Configuração dos testes 37

Tabela 11 - Métricas dos componentes 38

Tabela 12 - Métricas enviadas para o algoritmo MeTH 44

Tabela 13 - Peso das métricas 46

Tabela 14 - Comparação entre decisões dos algoritmos 51

Tabela 15 - *Thresholds* atualizados 52

Tabela 16 - Configuração dos testes do algoritmo 53

Glossário

API	Grupo de métodos ou pontos de acesso que um *software* permitindo o acessp a um conjunto de funcionalidade por um utilizador, sem que esteja implícito o conhecimento intrínseco dos processos
DNSaaS	Serviço de Domain Name System disponibilizado na cloud
EBS	Armazenamento em bloco utilizado em instâncias
ECU	Medida relativa da capacidade total de processamento de uma instância
Flavor	Combinação de recursos (processamento, memória, armazenamento e rede) relativos a uma instância
FP7	Programa criado para suportar e fomentar projetos de pesquisa tecnológica
Multi-tenant	Princípio segundo o qual uma instância de um determinado software serve diversos grupos isolados de tenants
OCCI	Conjunto de especificações e standards para a cloud
OpenShift	Solução Platform as a Service, com suporte a uma grande variedade de linguagens de programação, base de dados e frameworks
OpenStack	Solução Infrastructure as a Service, open-source
Pooling	Técnica de partilha de recursos de computação, possibilitando que nas máquinas virtuais os recursos sejam atribuídos de forma dinâmica
Service Level Agreement	Definição dos aspetos (âmbito, qualidade e responsabilidade) em que o serviço é acordado entre o utilizador e o fornecedor do serviço.
Snapshot	Cópia do estado (software instalado e configurações realizadas) de uma instância.
Tenant	Entidades (utilizadores ou serviços) que acedem aos serviços
Threshold	Limite de métricas
Token	Responsável pela autenticação de um tenant num serviço. Os tokens são constituídos por uma identificação e tempo de expiração.
vCPU	Processador virtual atribuído a uma instância.

Acrónimos

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CC	Cloud Controller
CDNaaS	Content Delivery Network as a Service
CPU	Central Processing Unit
CRUD	Create, Read, Update and Delete
DNS	Domain Name System
DNSaaS	Domain Name System as a Service
EBS	Elastic Block Store
ECU	Elastic Compute Unit
FP7	Framework Programmes 7
GPS	Global Positioning System
GPRS	General Packet Radio Service
IaaS	Infrastructure as a Service
IP	Internet Protocol
IT	Information Technology
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MaaS	Monitoring as a Service
MADM	Multiple Attribute Decision Making
MCN	Mobile Cloud Networking
NaaS	Networking as a Service
OCCI	Open Cloud Computing Interface
PaaS	Platform as a Service
QPS	Queries per Second
REST	Representational State Transfer
SaaS	Software as a Service
SDK	Software Development Kit
SLA	Service Level Agreement
SLO	Service Level I
SM	Service Manager
SO	Service Orchestrators
SQL	Structured Query Language
VNC	Virtual Network Computing
YAML	YAML Ain't Markup Language

Capítulo 1 : Introdução

O presente relatório tem como objetivo apresentar o trabalho desenvolvido ao longo de um semestre curricular, na empresa OneSource Consultoria Informática no âmbito do projeto MCN, pelo estagiário João Manuel Marques dos Santos. Este estágio teve como orientadores da empresa os Doutores Bruno Sousa e David Palma e como orientador do Departamento de Engenharia Informática o Professor Doutor Edmundo Monteiro.

A OneSource é uma empresa IT, sediada em Coimbra, especializada nas áreas de comunicação de dados, segurança, gestão de sistemas e redes, incluindo também consultoria, auditoria, *design*, desenvolvimento e administração de soluções IT. Desde 2012 que a OneSource é parceira do projeto Mobile Cloud Network, estando o módulo de DNSaaS sob a sua alçada.

1.1. Contexto

Cloud computing e *mobile computing* são paradigmas em franca expansão. Com a sua conjugação desenvolve-se um novo conceito – o *mobile cloud computing*, que pretende oferecer um conjunto mais alargado de recursos aos seus utilizadores, tendo em conta o crescente número de utilizadores com dispositivos móveis, mas que por vezes não dispõem desses recursos. Considerando que muitas aplicações desenvolvidas para dispositivos móveis requerem bastante poder computacional e são caracterizadas por algumas dependências a nível de *software*, a introdução do *mobile cloud computing* vem permitir que os recursos necessários para utilizar essas aplicações fiquem disponíveis através da "cloud". Isto possibilita o desenvolvimento das aplicações sem grandes limitações graças aos recursos disponíveis na *cloud*, tornando assim possível o desenvolvimento de aplicações mais complexas. Não obstante a tentativa para a sua clarificação, o conceito de *mobile cloud computing* carece de uma definição técnica, sob pena de uma utilização incorreta, desvalorizando o seu potencial.

O projeto europeu Mobile Cloud Networking [1] vem colmatar esta indefinição, ao elucidar sobre o conceito de *mobile cloud computing*, e permitindo também que as organizações de telecomunicações europeias liderem este paradigma, ainda com um vasto potencial para

exploração. Uma vez que *cloud computing* se integra na área de *software*, levando com isso a algumas dificuldades de interpretação por parte de especialistas de telecomunicações, é também objetivo deste projeto inverter essa situação. Este projeto foi iniciado em novembro de 2012 e estende-se por um período de 36 meses, estando inserido no programa do FP7, sob coordenação da SAP e liderança/orientação técnica da ZHAW (Universidade de Zurique) e do seu laboratório ICCLAB que abrange a área de *cloud computing*. São parceiras deste projeto algumas organizações como a Orange, Portugal Telecom, Intel, CloudSigma, OneSource entre outras organizações e universidades.

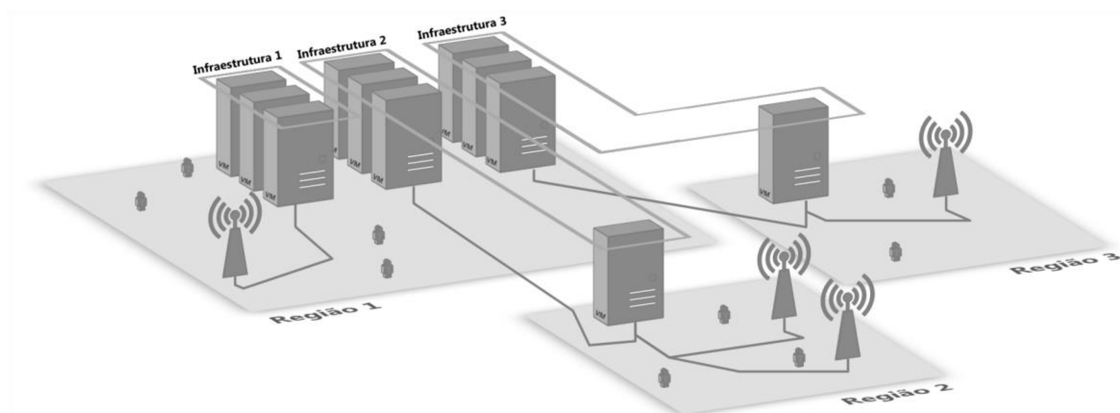


Figura 1 - Arquitetura conceptual do projeto MCN [2]

Telecomunicações móveis

Em março 1984 foi lançado pela Motorola, o primeiro telemóvel para as grandes massas [3]. Este acontecimento veio revolucionar a forma como as pessoas comunicam, permitindo-lhes comunicar em movimento. Nos seguintes anos, estes dispositivos evoluíram, tornando-se mais pequenos, finos, leves e com mais recursos computacionais. Com a evolução da sua fisionomia, mas acima de tudo dos recursos que estes dispositivos possuem, começaram a emergir há cerca de uma década atrás, os *smartphones*. Esta nova geração de telemóveis com maior poder computacional e conetividade (WiFi, Bluetooth, GPS e GPRS), possibilita uma série de novas funcionalidades e *software* que são suportadas pelos sistemas operativos dos telemóveis. Os dispositivos móveis, devido às suas novas características têm vindo a obter uma grande cotação no mercado, contrastando com os computadores que têm perdido. A figura 2 representa uma previsão da International Data Corporation sobre o peso que os *smartphones* irão assumir na nossa sociedade nos próximos anos [4].

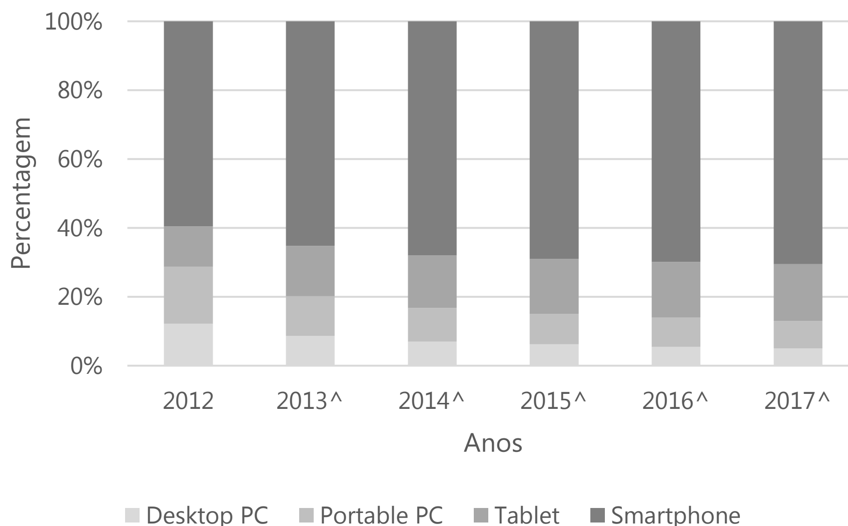


Figura 2 - Previsão do mercado dos dispositivos [4]

Cloud Computing

Mobilidade e escalabilidade são dois conceitos em crescente popularidade. Diariamente as pessoas encontram-se em constante movimento, não prescindindo da disponibilidade e do acesso aos seus ficheiros ou dados em qualquer lugar e altura (paradigma *anywhere, anytime*). Atualmente, basta acederem ao seu repositório *online* através de um dispositivo móvel, dispensando deste modo a utilização do disco rígido externo ou *pen drive*. No decorrer deste processo de consulta de informação cada pessoa utiliza, de forma inconsciente, serviços que o auxiliaram a efetuar a sua tarefa. Estes serviços são utilizados por uma grande quantidade de pessoas e nesse sentido para suportar esta grande quantidade de acesso deve ser escalável, favorecendo e permitindo a existência de um serviço com qualidade a preços competitivos. É esta ideia de permitir qualidade e mobilidade ao utilizador que está na base da tecnologia de *cloud computing*.

Investigada em 1960, esta tecnologia teve como impulsionador John McCarthy [5]. No entanto, só após cerca de 40 anos é que esta tecnologia começa a mostrar o seu potencial, contando com o contributo de várias empresas, entre as quais se destacam por exemplo a Amazon, Google, IBM e Microsoft.

Este paradigma está a revolucionar a forma como os recursos de IT são consumidos, da mesma forma que a Internet revolucionou as comunicações. Com este paradigma as arquiteturas são dinâmicas consoante a utilização do serviço, deixando para trás arquiteturas megalómanas devido a picos ocasionais. A prova do crescimento deste paradigma é ilustrado na figura 3, que de acordo com um estudo levado a cabo pela International Data Corporation mostra uma previsão do que se gastará em 2017 em *cloud computing* [6].

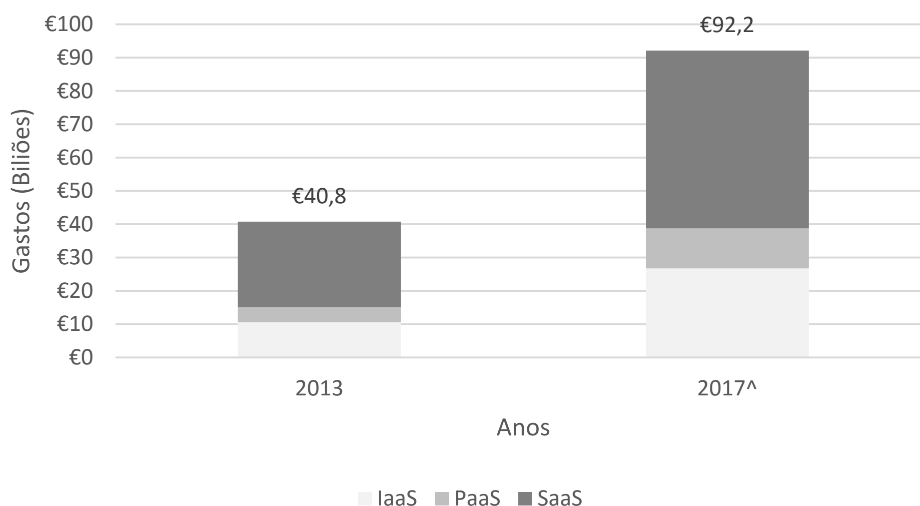


Figura 3 - Previsão dos gastos nos serviços de *cloud computing*. [6]

Este paradigma deve muito à virtualização, que permite a divisão de recursos (CPU, memória e armazenamento), criando desta forma uma instância com os recursos e sistema operativo pretendidos. A virtualização possibilita que um *datacenter* em subutilização possa ter maior aproveitamento, dividindo os recursos de forma a conseguir alocar mais serviços ou prestar melhor qualidade aos existentes. Esta tecnologia deriva de uma investigação levada cabo pela IBM em 1960, tendo como peça principal o *hypervisor*, uma plataforma que cria e executa máquinas virtuais [7].

Domain Name System

Serviço cuja principal finalidade é resolver nomes em endereços IP, no sentido de serem encaminhados no Internet, por exemplo o nome http://www.google.pt pode corresponder ao endereço IP 216.58.208.4. Além desta função o DNS também dispõe da resolução reversa que é utilizada está disponível o endereço IP e se pretende obter o domínio. Uma outra característica a destacar no DNS é o geodns [8] que permite dividir os pedidos geograficamente – consoante a localização do pedido a resposta será diferente. Isto permite que seja acedido ao endereço mais perto, reduzindo o tempo de comunicação.

Teve a sua origem durante o projeto ARPAnet, sendo apenas utilizado um ficheiro de texto com as conversões necessárias para os domínios existentes [9]. Devido seu ao crescimento exponencial de domínios existentes, este serviço sofreu alterações na arquitetura de modo a responder eficientemente a um número exponencial de pedidos de resolução de nomes, dada a proliferação de dispositivos e serviços que assentam no DNS.

Com a *cloud* a desempenhar um papel importante nas novas arquiteturas dos serviços, o DNS inicia a sua migração para esta nova tecnologia permitindo deste modo um serviço elástico. Possibilita

que serviços de DNS como o do Google – que recebe cerca de 70 biliões de pedidos [10] – sejam mais facilmente geridos e obtenham um melhor desempenho. As características que a *cloud* garante ao DNS e que levam a migrar para estes ambientes são a garantia que tem à disposição recursos dos quais se vai servindo, permitindo fazer uma melhor gestão dos recursos e utilizando apenas do que o serviço necessita. Estas características fazem com que o custo de um serviço de DNS seja reduzido.

1.2. Objetivos

Os objetivos deste trabalho podem ser agrupados em torno de dois âmbitos essenciais: Pessoal – com o foco principal na experiência e conhecimentos adquiridos – e Projeto – correspondente à implementação das características propostas. Relativamente à vertente pessoal, os objetivos são tomar contacto e ganhar experiência no mundo empresarial – onde o empenho e o trabalho em equipa são essenciais para garantir que as metas sejam cumpridas com a qualidade exigida –, aprender a linguagem Python (uma vez que todo o projeto é escrito nesta linguagem) e obter conhecimentos em OpenStack, a IaaS escolhida para o projeto MCN. No que diz respeito ao projeto, o objetivo passa por desenvolver um algoritmo de escalabilidade para DNS em contexto *cloud*. O algoritmo de escalabilidade deve ter as seguintes características:

- Racionar da melhor forma os recursos, não descorando para isso a *performance* do serviço.
- Dever ser o mais discreto possível, não tendo impacto negativo no serviço
- Ser ajustável à infraestrutura e às necessidades do serviço

A seguinte lista demonstra as etapas para atingir os objetivos propostos:

1) Familiarização com os sistemas DNS e *cloud computing*: permitindo desenvolver conhecimento sobre uma área ainda não explorada, indo ao encontro dos objetivos pessoais.
2) Especificação da arquitetura do DNSaaS: efetuando testes de *stress* às arquiteturas propostas de maneira a propor a abordagem mais escalável e fiável.
3) Desenvolvimento do algoritmo: permitindo que a arquitetura suporte escalabilidade com o menor custo possível, maximizando o desempenho do serviço. Para validar este algoritmo serão feitos testes demonstrativos da sua *performance* no serviço de DNS.

1.3. Metodologia

No decorrer do estágio foi utilizada a metodologia de projetos de investigação [11], conforme representado na figura 4, pretende auxiliar e compreender se a solução proposta para um problema está correta. Este modelo enconta como ponto de partida, a existência

de um problema, para qual é necessário uma solução, tomando-se durante este processo, decisões com visto à sua resolução. Durante o desenvolvimento, a solução para o problema encontrado é estruturada e implementada para depois ser testada durante a avaliação. Efetuadas todas as avaliações à solução, tiram-se conclusões para tentar perceber se o problema fica resolvido através da proposta de solução encontrada.

Figura 4 - Metodologia de Projetos de Investigação

De referir que esta metodologia é utilizada para garantir o objetivo do projeto, assim como para alguns problemas que foram surgindo no decorrer do projeto. Tendo como exemplo a definição arquitetura de DNSaaS, e partindo do problema/indefinição sobre a estrutura da arquitetura, foram apresentadas duas soluções para o problema. Implementadas as soluções avançou-se para uma etapa de avaliação, que por sua vez permitiu tirar conclusões sobre qual a melhor solução para o problema.

1.4. Planeamento

Neste capítulo é apresentado o plano de trabalhos do estágio, sendo representado através de dois diagramas gantt a Figura 5 que ilustra o planeamento inicial e a Figura 6 que representa o planeamento executado após os atrasos. De seguida será realizada uma descrição do trabalho realizado.

- **Familiarização:** Decorreu o processo de enquadramento e adaptação ao projeto MCN e tecnologias utilizadas.
- **Especificação da arquitetura:** Elaboração de possíveis arquiteturas escaláveis para o serviço de DNS.
- **Testes à arquitetura:** Avaliação das arquiteturas propostas.
- **Escrita de artigo:** *Paper* redigido com a avaliação da arquiteturas escaláveis [12].
- **Escrita do relatório intermédio:** Descrição do estado do trabalho efetuado durante o 1º semestre.
- **Especificação do algoritmo:** Planificação da estrutura do algoritmo de escalabilidade.
- **Implementação do algoritmo:** Desenvolvimento do algoritmo de escalabilidade do serviço.
- **Validação do algoritmo:** Testes ao algoritmo proposto.
- **Escrita do relatório final:** Descrição do estão do trabalho efetuado durante o ano letivo.

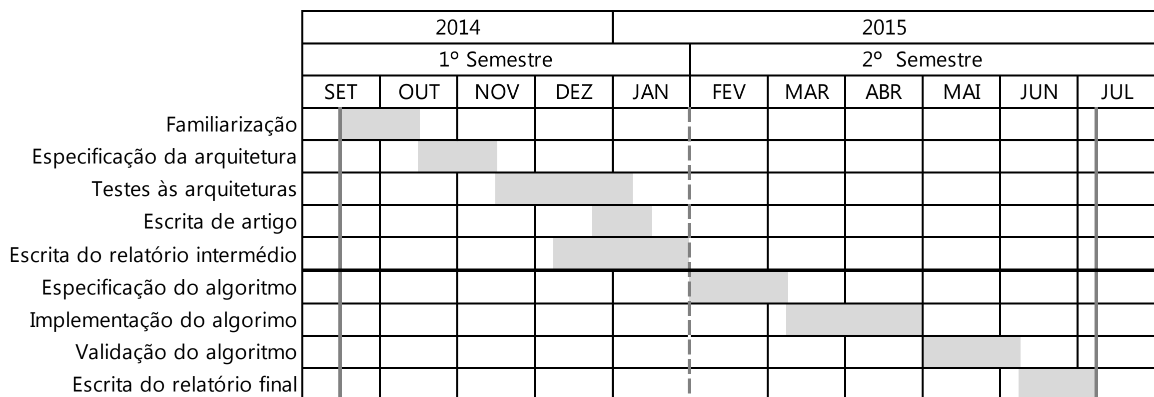


Figura 5 - Planeamento inicial

Relativamente ao 1º semestre não houve atrasos e o planeamento foi cumprido na íntegra, mas no 2º semestre houve um certo desvio do que foi planeado inicialmente. As atualizações dos *testbeds* e problemas de comunicação entre os vários componentes fizeram com que o trabalho sofresse um atraso considerável. Devido aos problemas de comunicação a tarefa de implementação do algoritmo teve de ser retomada de forma a contornar esses mesmos problemas através de codificação.

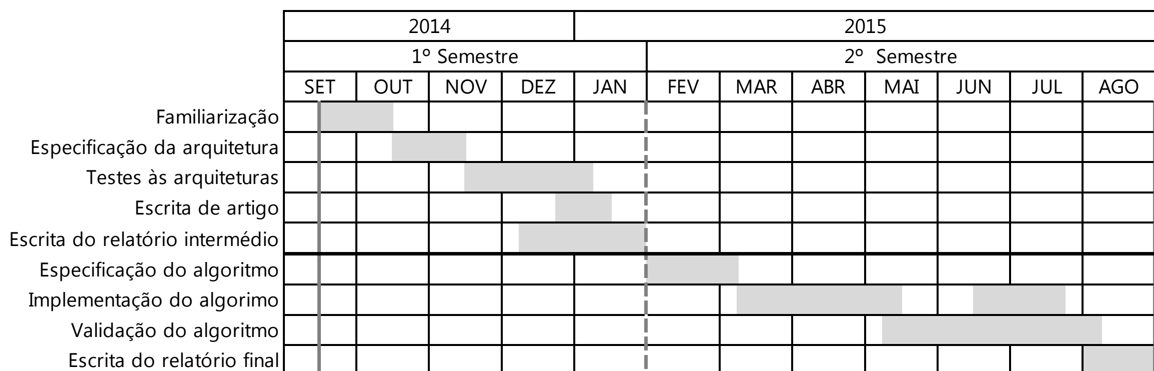


Figura 6 - Planeamento executado

1.5. Trabalho realizado

A Figura 7 representa todos os componentes presentes durante o ciclo de vida do serviço de DNS. Dos componentes presentes na Figura não tive contacto apenas com o CC (*Cloud Controller*) devido ao fato de este componente ser partilhado com os restantes serviços do MCN sendo portanto ser um componente geral do projeto e não um serviço dedicado ao DNS. Relativamente ao componente SM (*Service Manager*) participei no SDK onde contribui com implementação de um método responsável por devolver os *endpoints* do serviço de DNS, caso este não exista o método inicia tudo o processo de criação do serviço e das suas respetivas dependências.

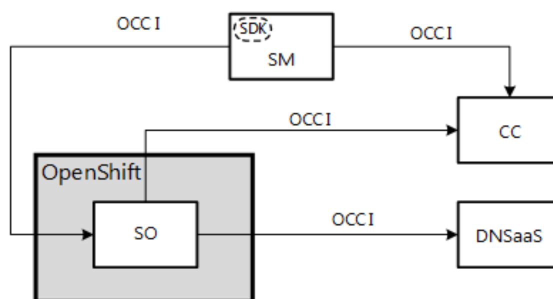


Figura 7 - Componentes presentes durante o ciclo de vida do DNSaaS

No componente de DNSaaS defini uma arquitetura escalável para o serviço, com base em testes de *stress*, garantindo desta forma que a arquitetura não tornava impeditiva a tarefa de escalabilidade dinâmica ao algoritmo.

Por último, o SO (*Service Orchestrator*) – componente onde estive mais envolvido, onde realizei alterações da estrutura das classes conforme os *standards* do OCCI, permitindo interoperabilidade e portabilidade ao componente. Foi ainda neste componente que implementei o algoritmo de escalabilidade.

1.6. Estrutura do documento

Este relatório está divido em 6 capítulos, sendo cada um deles apresentado de seguida:

1. Capítulo introdutório ao relatório, onde é abordado o contexto, os objetivos e planeamento do estágio.
2. Apresentado o estudo da estado da arte uma análise aos componentes do projeto e uma análise às soluções de tecnologias.
3. Explanação das arquiteturas propostas para o serviço DNSaaS e suas características
4. Validação das arquiteturas, analisando testes de *stress* efetuados de forma a aferir a mais escalável.
5. Capitulo que descreve o processo especificação, implementação e análise ao algoritmo desenvolvido.
6. Apresentação das conclusões relativas ao estágio e possíveis medidas futuras a tomar.

O documento ainda contem um conjunto de anexos que contém informação de suporte:

A. Exemplo de um *template* e o resultado no OpenStack

B. Algoritmo MeThodical

Capítulo 2 : Estado da Arte

Neste capítulo pretende-se apresentar o estudo do estado da arte realizado, numa breve reflexão sobre as atuais metodologias, arquiteturas e tecnologias que se enquadram no âmbito deste estágio. Será também realizada uma descrição e análise de soluções, com funcionalidades que se enquadram nos moldes do projeto.

2.1. *Cloud Computing*

Cloud computing é um termo genérico utilizado para descrever um novo paradigma de computação que fornece recursos computacionais (aplicações, serviços, computação) e a infraestrutura (armazenamento, rede) subjacente. O objetivo do *cloud computing* é abstrair toda infraestrutura ao utilizador, que comunica apenas com os serviços que gerem a "cloud". Este paradigma disponibiliza grande poder computacional e permite às organizações crescerem consoante as suas reais necessidades.

Recentemente tem-se assistido a um aumento da utilização do termo *cloud*, especialmente motivado pela crescente migração das organizações para ambientes que permitam uma mais fácil gestão dos seus serviços[13]. Entre as principais razões que levam as organizações a optarem pela *cloud* é possível destacar, por exemplo, as seguintes:

- Utilização mais eficiente dos recursos, economizando tempo na gestão da infraestrutura;
- Redução dos custos de aquisição e manutenção de *hardware*;
- Aumento da agilidade dos negócios, permitindo que as organizações satisfaçam as necessidades em ambientes que mudam rapidamente;
- Redução dos custos associados ao desenvolvimento e administração de serviços.

Com o crescente desenvolvimento de serviços torna-se possível a identificação de um conjunto de características comuns, que o Institute of Standards and Technology distingue em cinco grupos:[14]

- **Serviços *Self-Service*:** O utilizador pode consumir os recursos que mais lhe convêm, podendo aumentar ou diminuir as capacidades computacionais sem necessitar de interação humana com o fornecedor de cada serviço.

- **Amplo acesso à rede:** Todas as funcionalidades estão disponíveis através da rede e são acessíveis através de mecanismos *standard*, que promovem o uso de plataformas-cliente heterogéneas.
- ***Pool* de Recursos:** Garantia de recursos *multi-tenant* e *pooling*, que permitem combinar recursos computacionais (processamento, memória, armazenamento, rede) de forma heterogénea e servir vários clientes, de forma dinâmica.
- **Elasticidade:** Os recursos são maleáveis, permitindo rápida e facilmente a sua escalabilidade, respondendo às necessidades.
- **Serviços Mensuráveis:** A *cloud* efetua uma monitorização da utilização dos recursos possibilitando desta forma uma otimização e gestão automática dos recursos.

Modelos de Implementação e Serviços

O *Cloud Computing* tem três modelos de implementação que informam como o ambiente *cloud* disponibiliza os seus serviços. Estes modelos de implementação podem ser classificados como[15]:

- ***Cloud* Pública:** são da propriedade, administração e operacionalização de organizações que oferecem um acesso rápido a recursos de computação a outras organizações ou utilizadores. Com este modelo os utilizadores não terão de adquirir *hardware*, *software* ou qualquer outra infraestrutura de apoio, sendo o seu fornecimento e gestão garantidos pelo fornecedor do serviço.
- ***Cloud* Privada:** são da propriedade, administração e operacionalizadas por parte de apenas uma organização que controla o modo como os recursos são virtualizados e os serviços são usados por outras organizações. Este modelo aproveita a *cloud* de forma eficiente, oferecendo maior controlo sobre os recursos e serviços.
- ***Cloud* Hibrida:** utiliza a base da *cloud* privada combinada com os serviços da *cloud* pública. Na realidade, uma *cloud* privada não pode existir isolada do resto dos recursos e da *cloud* pública da organização. As *clouds* são interligadas por tecnologia *standard* ou proprietária que permite a portabilidade de dados e aplicações. Este modelo permite que as organizações migrem para ambientes *cloud*, de forma mais fácil e segura, conseguindo manter os seus dados confidencias e aplicações internas ou privadas na *cloud* privada e migrar as restantes aplicações e dados para a *cloud* pública.

Por sua vez os serviços de *cloud computing* podem subdividir em três grandes grupos[16]:

- **Software-as-a-Service:** permite aos utilizadores finais aceder e utilizar aplicações abstraindo o utilizador das camadas de Plataforma e Infraestrutura. A utilização deste tipo de aplicações representa uma alternativa económica para as organizações, uma vez

que reduz os custos com recursos computacionais e humanos, pois as aplicações são executadas na *cloud* e ao adquirirem um serviço não terão de gastar em desenvolvimento e administração da aplicação. O utilizador não tem que se preocupar com a instalação, configuração ou execução da aplicação, tendo apenas que usá-la através de um cliente ou API.

- **Platform-as-a-Service:** é colocada à disposição dos *developers* uma plataforma (ferramentas, bibliotecas e recursos computacionais) de suporte para desenvolvimento e implementação de *software*. Este serviço fornece ainda um ambiente de instalação e implementação de aplicações e serviços sem a complexidade de custos necessários para administrar toda a infraestrutura subjacente.
- **Infrastructure-as-a-Service:** através deste tipo de serviço as organizações têm um controlo sobre a infraestrutura que se encontra na *cloud*. São disponibilizados recursos computacionais (armazenamento, comunicação, computação) virtuais, possibilitando a construção de sistemas. Permite às organizações uma fácil gestão na construção dos seus sistemas, com a vantagem de dispensar custos de aquisição, manutenção e administração do *hardware*.

Web browser, Aplicações mobile

		Pública	Privada	Híbrida
Aplicações, *Web Services*, Multimédia	SaaS	Basecamp	-	-
Frameworks, Armazenamento	PaaS	Google App Engine	OpenShift	Pivotal Cloud Foundry
Computação, virtualização CPU, Memória, Disco, Rede	IaaS	Amazon Elastic Compute Cloud	Openstack	Rackspace

Figura 8- Serviços vs Implementação

2.2. Mobile Cloud Networking

Sendo o projeto MCN um projeto de investigação inovador que pretende implementar um sistema de *Mobile Cloud Computing*, a arquitetura deste novo sistema teve de ser desenhada da raiz. Este capítulo pretende descrever os componentes na arquitetura e como se comportam durante a execução.

Serviços

Na arquitetura do MCN é possível identificar quatro tipos de serviços que distinguem os comportamentos e ações dos serviços [17].

- **Atómico:** serviço nuclear que executa funções técnicas e de negócio, geralmente utilizando recursos (computação, armazenamento e rede) do fornecedor do serviço. Um claro exemplo são os serviços de computação (e.g. OpenStack Nova [38]) ou de armazenamento em bloco (e.g. OpenStack Cinder [41]).
- **Composto:** serviço que combina dois ou mais serviços, incluindo Atómicos ou mesmo outros serviços compostos. No projeto MCN existem duas categorias de serviços Compostos, que são os serviços de Suporte e MCN.
- **Suporte:** serviço de plataforma do MCN que fornece objetivos e funções específicas de utilização de qualquer serviço. Exemplos de serviços de Suporte são MaaS e CDNaaS.
- **MCN:** serviços desenvolvidos no projeto MCN. Este serviço é constituído por um Service Manager, um ou mais Service Orchestrators e as suas funcionalidades.

Ciclo de vida dos serviços

No projeto MCN, o ciclo de vida dos serviços é expresso em duas fases distintas – a fase de Negócio e Técnica [17]. A fase de Negócio contém todas as atividades relacionadas com a conceptualização do serviço, encontrando e pesquisando outros serviços que possam ser combinados no novo serviço. Esta fase é realizada na sua grande maioria por administradores. A figura 9 retrata fase de Negócios, que por sua vez se desdobra em duas etapas.

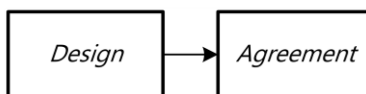


Figura 9 - Fase de negócio [17]

Na primeira etapa, *Design*, corresponde à conceptualização do serviço. Note-se que os serviços que não conseguem ser projetados pela organização são encaminhados para outra organização e os requisitos do serviço são verificados para uma possível combinação com outros serviços. Na etapa de *Agreement*, o preço, o SLA, o acesso e outros condicionantes que envolvam o usufruto do serviço são acordados entre as organizações.

Voltando ao ciclo de vida dos serviços no projeto MCN, para além da fase de Negócio, há a realçar a segunda fase – a fase Técnica, que aborda essencialmente todas as atividades desde a projeção até à eliminação do serviço. Esta fase tem dependências pelas decisões e acordos tomados na fase de Negócio pelos fornecedores do serviço. Conforme demonstrado na figura 10, a fase Técnica desenvolve-se ao longo de seis etapas, a saber *Design; Implementation; Deployment; Provisioning; Operation & Runtime Management* e *Disposal*.

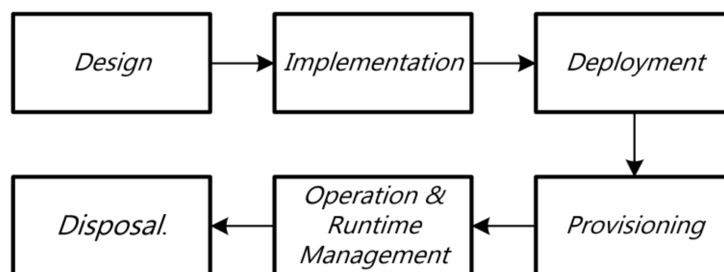


Figura 10 - Fase técnica [17]

Na etapa de *Design* são apresentadas soluções de projeção da arquitetura, *implementation*, *deployment*, *Provisioning* e Runtime, isto é, conceptualiza o serviço e projeta-o para as fases seguintes. Na etapa de *Implementation*, a arquitetura projetada, as funções, interfaces, controladores e APIs do serviço são elaborados. Na etapa de *Deployment*, são estabelecidos os elementos da fase anterior, criando condições para que o serviço possa ser usado, apesar de não lhe proporcionar acesso. Na etapa *Provisioning* são criadas as condições que permitem o acesso ao serviço. Na etapa de *Operation & Runtime Management*, o serviço está pronto e já se encontra em execução. Nesta fase realizam-se operações de escalabilidade e reconfiguração dos componentes da instância do serviço. Na etapa de *Disposal*, os componente do serviço na instância e a instância do serviço são libertados.

Elementos

Na arquitetura do MCN existem três elementos fundamentais que irão estar presentes na gestão dos serviços que estão em execução – *Service Manager* (SM), *Service Orchestrator* (SO) e o *CloudController* (CC) [17].

O SM faculta uma interface externa ao utilizador final sendo responsável não apenas pela gestão do(s) Service Orchestrator(s) mas também pelo acompanhamento dos serviços na sua fase Técnica. O SM tem duas funções essenciais, na perspetiva de negócio trata dos acordos entre os serviços e na perspetiva técnica gere o(s) Service Orchestrator(s) de um *tenant*.

O SO é um serviço chave na arquitetura MCN uma vez que ao supervisionar a orquestração da instância de um serviço, tem a responsabilidade de gerir os seus componentes quando o serviço for criado e estiver em execução. O SO gere todas as instâncias atribuídas a um serviço, sendo portanto um componente específico que contém a lógica do serviço. Está bastante presente na etapa Operação e Gestão do Tempo de Execução de um serviço sendo responsável pela implementar, configurar, migrar, escalar e eliminar os componentes da instância do serviço. Geralmente, por cada instância do serviço é inicializado somente um SO, que por sua vez irá estar sempre associados ao SM que o criou.

O CC tem acesso aos serviços Atómico e de Suporte, tendo a possibilidade de configurar os Atómicos. Está presente nas etapas dos serviços durante a sua *implementation*, *deployment* e *dispose.*

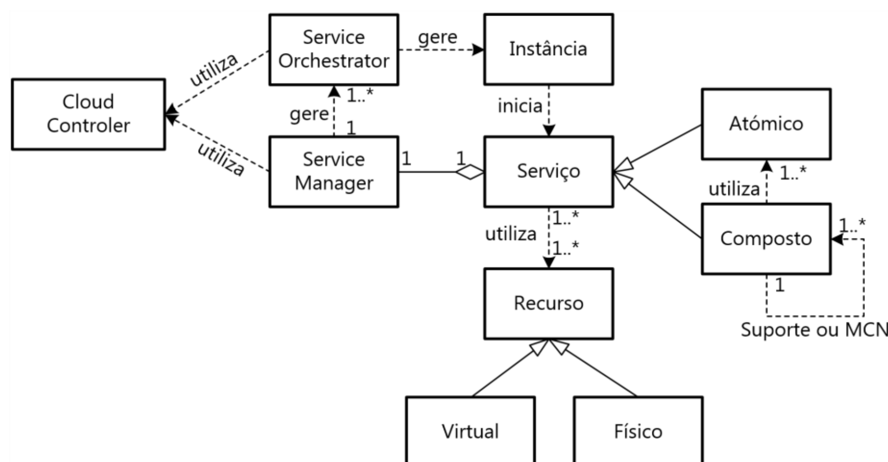


Figura 11 - Comunicação entre os elementos e serviços no MCN [17]

2.3. Soluções IaaS

A infraestrutura é parte essencial de um sistema, no entanto pode criar alguns problemas às organizações se considerarmos a dificuldade da sua manutenção e elevados custos. A *cloud* vem responder a esta dificuldade, com a disponibilização de um serviço que permite a gestão de todos os recursos, de forma simples e económica. Assim, as organizações conseguem tirar proveito de um sistema mais completo, flexível e menos dispendioso.

Perante a necessidade de comparação das soluções IaaS e dos benefícios que oferecem comparativamente entre si, foram selecionadas cinco soluções que serão analisadas de acordo com vários critérios. Foram escolhidos *flavors* predefinidos pelas soluções, podendo os utilizadores fazer a combinação dos seus recursos (sistema operativo, vCPU, memória e armazenamento). Será apresentada também a disponibilidade que cada solução oferece no seu SLA, sendo os utilizadores reembolsados caso o valor da disponibilidade seja inferior ao referido, tentando desta forma garantir qualidade no serviço prestado.

Amazon Elastic Compute Cloud

O Amazon Elastic Compute Cloud [18] também denominado EC2 é uma solução desenvolvida pela Amazon, lançada pela primeira vez, em versão beta, a 25 de agosto de 2006. A EC2 é parte fulcral da plataforma de *cloud computing* da Amazon Web Services, uma vez que é sobre esta plataforma que vão ser executados os serviços. A Amazon ainda disponibiliza PaaS para os utilizadores desta solução, denominado de AWS Elastic Beanstalk (tendo a sua primeira versão sido lançada em 19

de janeiro de 2011), oferecendo aos utilizadores um conjunto de ferramentas e ambientes de desenvolvimento melhorados e com maior extensão.

A EC2 é do tipo *cloud* pública o que significa a existência de uma organização a pagar pelo serviço, consoante as suas necessidades, ou seja do tipo *pay-as-you-go*. Esta solução tem, no seu SLA, um compromisso de disponibilidade em 99,95% do tempo querendo dizer que uma instância passa, no máximo, cerca de 4 horas e 23 minutos inativa num ano. [19]

Esta solução tem à disposição vários tipos de *flavors* que permitem que o utilizador possa escolher os recursos, conforme as suas necessidades e do que pretende ter na sua instância. De seguida, é apresentada uma tabela de preços [20] onde é possível constatar alguns dos *flavors* que a EC2 disponibiliza, sendo que estes valores correspondem a instâncias localizadas na União Europeia (Frankfurt) com o sistema operativo baseado em Linux.

Flavor	vCPU	ECU	Memória (GiB)	Armazenamento (GB)	Preço (por hora)
t2.micro	1	Variável	1	0 (só EBS)	€0,012
t2.small	1	Variável	2	0 (só EBS)	€0,035
t2.medium	2	Variável	4	0 (só EBS)	€0,048
m3.medium	1	3	3,75	1 x 4 SSD	€0,066
m3.large	2	6.5	7,5	1 x 32 SSD	€0,134
m3.xlarge	4	13	15	2 x 40 SSD	€0,267
m3.2xlarge	8	26	30	2 x 80 SSD	€0,535

Tabela 1 - Preços de instâncias na EC2

Microsoft Azure

A Microsoft Azure [21] é uma solução desenvolvida pela Microsoft com a versão inicial lançada em 1 de fevereiro de 2010. Esta solução fornece os serviços de IaaS e PaaS e suporta diferentes linguagens de programação, *frameworks* e sistemas operativos. Esta combinação permite construir, implementar e gerir os serviços de forma personalizada, rápida e fácil, permitindo uma boa produtividade.

A Microsoft Azure é do tipo *cloud* híbrida o que significa que uma organização que escolha esta solução poderá dividir os dados e aplicações pela *cloud* pública e privada, tornando a sua migração mais fácil e segura, uma vez que poderá aproveitar os seus *data centers* para *cloud* privada contendo informações e aplicações confidenciais, para que na *cloud* privada sejam armazenadas a

informações e aplicações destinadas ao público. Esta solução tem no seu SLA um compromisso de disponibilidade do serviço em 99,95% do tempo. [22]

Esta solução, como em todas as soluções de IaaS, dispõe de vários tipos de *flavors* que permitem que o utilizador possa escolher os recursos, conforme as suas necessidades do que pretende ter na sua instância. A seguinte tabela de preços [23] permite elucidar sobre alguns dos *flavors* que a Microsoft Azure disponibiliza na sua *cloud* pública, sendo que os preços abaixo correspondem a instâncias na União Europeia com o sistema operativo baseado em Linux.

Flavor	vCPU	Memória (GB)	Armazenamento (GB)	Preço (por hora)
A0	1	0.75	20	€0,0149
A1	1	1,75	70	€0,0447
A2	2	3,5	135	€0,0894
A3	4	7	285	€0,1788
A4	8	14	605	€0,3575
A5	2	14	135	€0,2011
A6	4	28	285	€0,4022
A7	8	56	605	€0,8043

Tabela 2 - Preços de instâncias no Azure

Rackspace Cloud

O Rackspace Cloud [24], anteriormente denominado de Mosso LLC, foi anunciado em 4 de março de 2006 pela Rackspace. Esta solução para além de fornecer IaaS também disponibiliza PaaS fazendo com que seja uma solução bastante completa possibilitando ambientes de desenvolvimento, *framework* e outras ferramentas suportando assim os serviços implementados. A Rackspace ao longo do tempo tem contribuído para projetos *open-source* [25], um dos quais seu concorrente direto – o OpenStack, modelo para as soluções IaaS open-source.

O Rackspace Cloud é uma *cloud* híbrida, permitindo às organizações terem *cloud* pública e/ou *cloud* privada, sendo uma ótima opção para as organizações migrarem para este novo paradigma ou mesmo como uma forma de obter maior segurança nos dados e serviços confidenciais, colocando-os na *cloud* privada e os serviços e dados públicos na *cloud* pública, sendo esta última de mais fácil acesso às grandes massas. Esta solução tem no seu SLA uma disponibilidade de 100%, garantido assim um serviço 24x7x365 aos utilizadores deste serviço[26].

Esta solução na Europa tem os seus servidores localizados em Londres. E como todas as soluções anteriores dispõe de *flavors* que irão permitir ao utilizador escolher os recursos que pretende que as instâncias tenham. De seguida é apresentada uma tabela com alguns *flavors* escolhidos como forma demonstrativa dos preços [27] praticados pela Rackspace na sua *cloud* pública. A Rackspace disponibiliza um serviço de três níveis de gestão para cada instância criada, sendo os valores a seguir exibidos referentes ao nível básico da gestão e com o sistema operativo baseado em Linux.

Flavor	vCPU	Memória (GB)	Armazenamento (GB)	Largura de Banda (Mb/s)	Preço (por hora)
General1-1	1	1	20	200	€0,037
General1-2	2	2	40	400	€0,074
General1-4	4	4	80	800	€0,148
General1-8	8	8	160	1.600	€0,296
Memory1-15	2	15	-	625	€0,218
Memory1-30	4	30	-	1.250	€0,425
Compute1-15	8	15	-	1.250	€0,395
Compute1-30	16	30	-	2.500	€0,790

Tabela 3 - Preços de instâncias no Rackspace

LunaCloud

LunaCloud [28] lançou a sua solução de IaaS em 1 de junho de 2012, tratando-se de uma empresa portuguesa com forte presença em território europeu, com recente expansão para a Rússia e com planos de desenvolvimento para o continente americano. Disponibiliza aos utilizadores também PaaS utilizando o serviço Jelastic permitindo um maior suporte no desenvolvimento e implementação de outros serviços. Com apenas 2 anos de existência esta solução premeia a *performance* e facilidade de processos, que lhe garantiram já um significativo número de prémios e distinções. Um bom exemplo são os testes conduzidos pela Cloud Spectator, durante 30 dias às soluções de IaaS da Amazon, Rackspace e LunaCloud obtendo esta melhores resultados que os outros dois concorrentes, mostrando o potencial e maturidade deste serviço. O resultado dos testes efetuados foram publicados em novembro de 2012. [29]

A LunaCloud disponibiliza um serviço de infraestrutura público, o que permite às organizações terem ao seu dispor apenas os recursos que o seu sistema necessita, pagando apenas por aquilo que precisam, ajustando os recursos às suas reais necessidades. Esta solução tem, no seu SLA, um

compromisso do serviço estar disponível em 99,99% do tempo, o que equivalente a cerca de 53 minutos de inatividade num ano[30].

Esta solução disponibiliza vários tipos de *flavors* com um conjunto de configurações que permitem ao utilizador escolher os recursos que pretende para a sua instância, indo ao encontro das necessidades do seu sistema. A Tabela 4 procura fazer a análise comparativa de alguns *flavors* que a LunaCloud disponibiliza. Os preços [31] abaixo indicados são para instâncias com sistema operativo baseado em Linux. Todo o tráfego público de entrada é gratuito, já no que respeita ao trafego público de saída, está limitado a 1000 GB por mês.

Flavor	vCPU	Memória (GB)	Armazenamento (GB)	Preço (por hora)
PointFive	1	0,5	10	€0,0155
One	1	1	50	€0,0270
Two	2	2	100	€0,0540
Four	2	4	250	€0,0990
Eight	2	8	500	€0,1840
OneSix	4	16	1000	€0,3680
ThreeTwo	4	32	1000	€0,6080

Tabela 4 - Preços de instâncias na LunaCloud

OpenStack

OpenStack [32] começou em julho de 2010 com uma parceria entre a National Aeronautics and Space Administration (NASA) e a Rackspace. Mais tarde a OpenStack Foundation (uma fundação sem fins lucrativos) ficou responsável pela gestão desta promissora solução IaaS. Atualmente este projeto é patrocinado por uma diversidade de organizações [33] entre as quais se encontra Canonical, IBM, HP, Intel, Red Hat, Cisco, Dell, estando ainda presentes alguns parceiros do projeto MCN como a SAP, a Orange, a Intel e a NEC.Essencialmente desenvolvido em Python sob a licença Apache License, o Openstack consiste num conjunto de projetos interrelacionados responsáveis por gerir os diferentes recursos.

Esta solução é uma *cloud* privada aproveitando-a de forma eficiente, oferecendo maior controlo sobre os recursos e serviços, permitindo efetuar um desenvolvimento e testes diretamente com o sistema.

O OpenStack tem à disposição do utilizador seis diferentes tipos de *flavors* com recursos diferentes, permitindo-lhe escolher os recursos que pretende que as suas instâncias tenham. A tabela abaixo apresenta os *flavors* padronizados [34].

Flavor	vCPU	Memória (GB)	Armazenamento (GB)
m1.tiny	1	0,5	1
m1.small	1	2	20
m1.medium	2	4	40
m1.large	4	8	80
m1.xlarge	8	16	160

Tabela 5 - *Flavors* das instâncias do Openstack

Comparação

Feita uma breve introdução a soluções presentes de IaaS, permite perceber que existem soluções suficientes para todos os problemas. Assim uma comparação possibilita perceber a melhor solução para cada propósito. Assim a Tabela 6 permite fazer uma comparação e a Tabela 7 as vantagens e desvantagens das diferentes soluções IaaS.

IaaS	Implementação	SLA	Preço (min/max)
Amazon Elastic Compute Cloud	*Cloud* Pública	99,95%	€0,012/€0,535
Microsoft Azure	*Cloud* Híbrida	99,95%	€0,0149/€0,8043 (*Cloud* Pública)
Rackspace Cloud	*Cloud* Híbrida	100%	€0,037/€0,790 (*Cloud* Pública)
LunaCloud	*Cloud* Pública	99,99%	€0,0155/€0,6080
OpenStack	*Cloud* Privada	-	€0

Tabela 6 - Comparação entre soluções IaaS

A Amazon Elastic Compute Cloud tem o serviço mais bem cotado no mercado, mas tem limitações que podem ser impeditivas a um projeto de telecomunicações. A Microsoft Azure têm a mais diversidade, flexibilidade e mais características que auxiliam num projeto de grande envergadura, mas tudo isto com o aumento do custo. A Rackspace Cloud é referência para muitas *clouds* e possui

um suporte de excelência, mas possui poucas características e funcionalidades, além de que os *tenants* estão restringidos à região, criando deste modo uma grande limitação. A LunaCloud é um serviço recente mas que apresenta os melhores resultados e está em franca expansão. Por último o OpenStack é *open*-source, tem uma boa comunidade, a sua arquitetura é constituída por módulos (projetos) em que cada um tem a sua finalidade, mas é um projeto em desenvolvimento, tem uma curva de aprendizagem elevada e possui um *deployment* complexo.

IaaS	Vantagens	Desvantagens
Amazon Elastic Compute Cloud	• Serviço mais utilizado • Serviço mais económico	• Serviço limitado a outros serviços ou plataformas
Microsoft Azure	• Serviço com melhor PaaS • *Cloud* Hibrida permite trocas de instâncias entre *Cloud* Publica e Privada	• Serviço mais caro
Rackspace Cloud	• Serviço referência no paradigma *cloud* • Excelente suporte	• Poucas características e funcionalidades • *Tenants* limitados à região
LunaCloud	• Serviço com melhores resultados • Serviço Europeu em expansão	• Serviço recente
OpenStack	• *Open-Source* • Arquitetura modular • Enorme comunidade	• Projeto em desenvolvimento • Elevada curva de aprendizagem (devido aos inúmeros projetos) • Complexo *deployment*

Tabela 7 - Características das soluções IaaS

2.4. OpenStack

O OpenStack foi a IaaS escolhida para o projeto MCN. Esta consiste numa série de projetos interrelacionados que oferecem todos os serviços que um IaaS deve fornecer aos utilizadores. Encontrando-se atualmente na versão Kilo, lançada em 30 de abril de 2015, alguns dos seus projetos são:[35]

- Horizon (Dashboard)
- Keystone (Serviço de identificação)
- Nova (Serviço de computação)
- Neutron (Serviço de rede)

- Swift (Serviço de armazenamento em objetos)
- Cinder (Serviço de armazenamento em bloco)
- Glance (Serviço de imagens)
- Ceilometer (Serviço de métricas)
- Heat (Serviço de orquestração)
- Designate (Serviço de DNS)

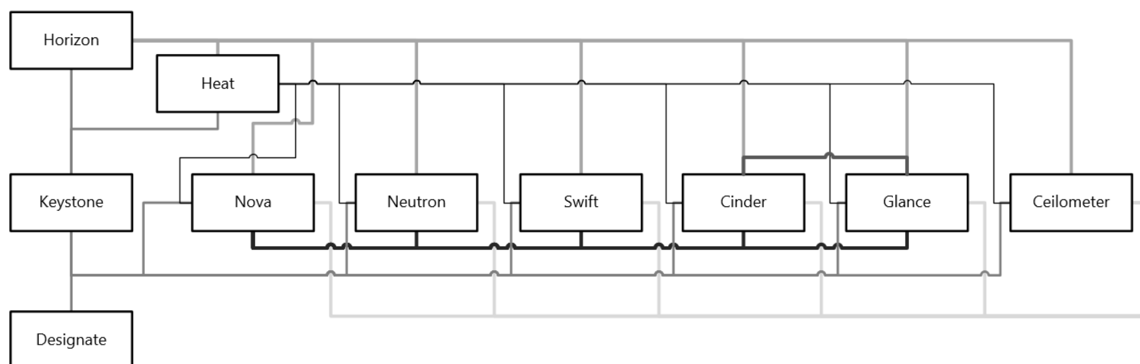


Figura 12 - Arquitetura conceptual do Openstack

Horizon

Horizon [36] é o dashboard do OpenStack, que disponibiliza ao utilizador uma aplicação web, desenvolvida em Django, que permite através da sua interface gráfica interagir com alguns serviços disponíveis no OpenStack.

Desta forma tarefa dos utilizadores de gerir as suas instâncias fica mais facilitada e intuitiva, ao inverso da linha de comandos. Assim, um utilizador pode selecionar as alterações de recursos que pretende no seu sistema, respeitando os limites estabelecidos pelos administradores para os projetos.

Keystone

O keystone [37] é o serviço responsável pela gestão da autenticação e autorização para os outros serviços do OpenStack, gestão da lista de *endpoints* para todos os serviços do OpenStack, gestão das políticas, sendo implementado no OpenStack Identity API.

Na figura seguinte é possível visualizar os quatro *backends* que o keystone contem. O *backend identity* contém validações das credenciais de autenticação e dados dos projetos, grupos, domínios, utilizadores e funções. O *backend token* valida e gere os *tokens* usados para autenticação dos pedidos com as credenciais dos utilizadores já verificados O *backend catalog* contém uma lista dos *endpoints* dos serviços que estão sob a alçada do OpenStack. O *backend policy* fornece mecanismos que possibilitam a definição de regras sobre permissões.

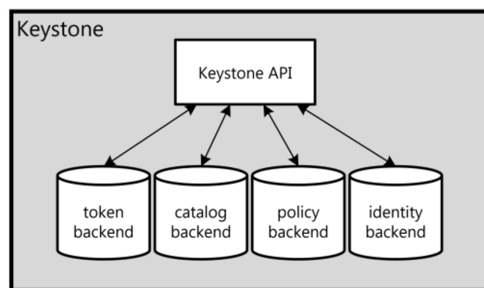


Figura 13 - Arquitetura lógica do Keystone

Assim, neste modelo podem existir vários projetos, sendo que um determinado projeto pode ter vários utilizadores que por sua vez podem estar ligados a um ou vários projetos, com funções distintas em cada um, tornando-se possível desta forma definir as permissões (funções) que o utilizador tem no projeto assim como a quota de recursos disponível para cada projeto.

Nova

Nova [38] é um dos serviços mais complexos e fundamentais do OpenStack, sendo responsável pela gestão de infraestrutura do sistema. Todas as atividades necessárias para manter uma instância são controladas por este serviço. O Nova tem cinco principais funções: Gestão do ciclo de vida das instâncias, Gestão da rede nas instâncias, Controlar o armazenamento nas instâncias, Gestão de grupos de segurança e Criação de *snapshots* de instâncias

A figura seguinte ilustra a arquitetura do Nova, assim como a comunicação entre os seus componentes.

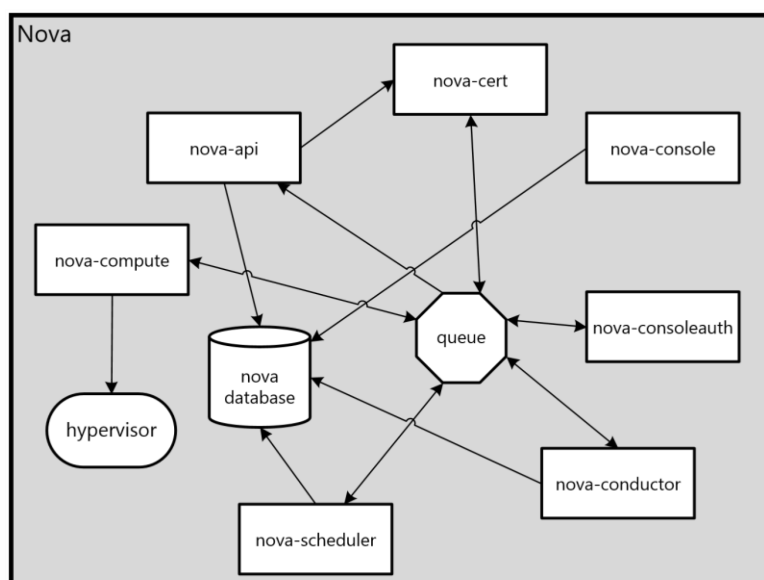


Figura 14 - Arquitetura lógica do Nova

O nova-api responsável por responder ao utilizador pedidos às APIs de computação caracteriza-se por responder ao utilizador com pedidos às APIs de computação. É também responsável por gerir e dar acesso à infraestrutura (e.g. reiniciar uma instância e gerir quota de recursos).

O nova-compute é o componente que cria e termina instâncias, tudo isto através da API do hypervisor e garantido ainda que o ciclo de vida de cada instância seja registado na base de dados. O nova-scheduler determina (sendo estas decisões configuráveis) em que nó de computação uma instância deve ser criada. O nova-console é o componente que possibilita o acesso à linha de comandos das instâncias. Através do componente nova-novncproxy um proxy é exposto utilizando VNC para aceder à linha de comandos. As autenticações às instâncias são geridas pelo componente nova-consoleauth. O nova-cert é o gestor de certificados durante os processos no Nova. O componente de *queue* é utilizado como um componente central, orientando as mensagens para que os componentes possam comunicar entre si, podendo ser utilizado um AMQP à escolha dos administradores. A nova database armazena – utilizando SQL – o estado de elemento da cloud (projetos, redes disponíveis e instâncias em execução). Por fim, o nova-conductor é o componente responsável por gerir os acessos à nova database.

Neutron

O Neutron [39] é o serviço do OpenStack responsável pela rede. Fornecendo NaaS entre as interfaces dos dispositivos geridos por outros serviços. Possibilita ao utilizador controlar o tráfego e gerir a criação de redes e interfaces, e para além de permitir flexibilidade na rede, possibilita também aos administradores a alteração do modelo de rede e adaptá-lo conforme o tráfego. O serviço Neutron permite também que o administrador escolha entre IPs fixos ou flutuantes.

Swift

O Swift [40] é um serviço de armazenamento de objetos, que ao serem armazenados em diretorias virtuais permitem armazenar e pesquisar ficheiros A arquitetura distribuída deste serviço permite escalabilidade horizontal, redundância e alta disponibilidade.

Cinder

O Cinder [41] é o serviço responsável pelo armazenamento em bloco. O serviço disponibiliza os dispositivos de armazenamento (volumes) utilizados pelas instâncias para armazenamento persistente e armazenamento de *snapshots*. Um volume apenas pode ser utilizado por uma instância, não podendo ser partilhado.

Glance

O Glance [42] é o serviço responsável pela gestão das imagens dos discos virtuais e dos *snapshots*. As imagens contêm ficheiros de sistemas (sistema operativo com software pré-definido) para que o utilizador possa iniciar uma instância com base na sua imagem e trabalhar nela a partir dessa imagem. Os *snapshots* são uma cópia do estado (software instalado e configurações realizadas) de uma instância. As imagens e *snapshots* são armazenadas no serviço Swift.

Ceilometer

O Ceilometer [43] é o serviço de monitorização e métrica de recursos dos serviços do OpenStack. O OpenStack recomenda para o armazenamento a utilização da MongoDB, uma base de dados NoSQL escalável e com uma boa *performance*. Este serviço saiu apenas na versão havana do OpenStack.

Heat

O Heat [44] é um serviço que fornece orquestração ao OpenStack. Surgindo apenas na versão havana, permite descrever a infraestrutura ou construir aplicações, denominadas "stacks" em ficheiros chamados de "template". Este *template* pode ser escrito em YAML e JSON. Utilizando o *template* o Heat invoca as APIs do serviços necessário, de forma a que os recursos sejam aprovisionados e a stack criada. Na seguinte figura está explanada a arquitetura do Heat.

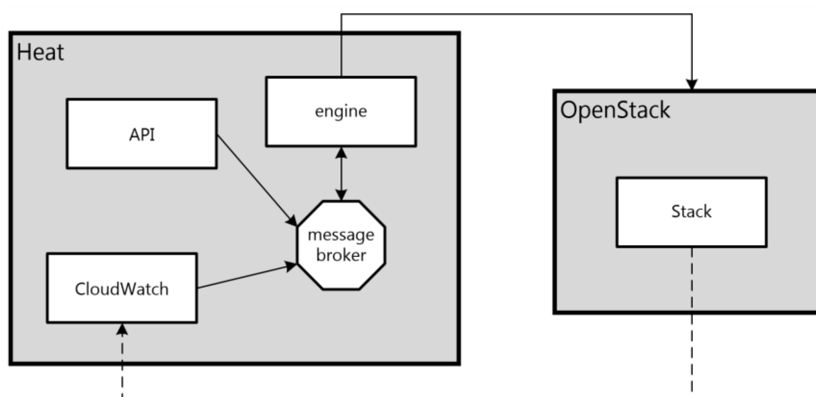


Figura 15 - Arquitetura lógica do Heat

O engine é o componente essencial, tendo a função de analisar e implementar os *templates* , este é o único componente que comunica diretamente com os serviços do OpenStack. A API dá acesso ao OpenStack através de sistemas RESTful. CloudWatch obtém as métricas das instâncias, de forma a monitorizar o seu estado. Todos estes componentes comunicam entre si através de um message broker.

Designate

O Designate [45] é um serviço – incubado recentemente na versão Juno – que é responsável por facultar DNS aos restantes serviços no OpenStack. Este serviço suporta PowerDNS e BIND9. A arquitetura assenta o seguinte esquema.

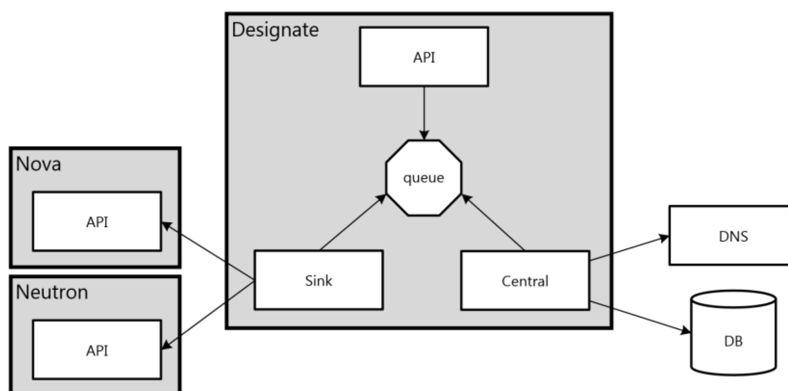


Figura 16 - Arquitetura lógica do Designate

O designate-api implementa a API, recebe e verifica os pedidos no serviço Keystone, transferindo-os para queue. O designate-central é responsável por coordenar os acessos ao *backend* do DNS (servidor e base de dados). O designate-sink é um serviço atento às eventuais alterações nos serviços Nova e Neutron, que podem adicionair ou remover registos no DNS. Este é um serviço opcional, mas que permite tornar o serviço de DNS num processo mais autónomo. O queue é um componente intermédio que serve de comunicação entre os serviços do Designate.

2.5. Soluções DNS

Domain Name Server é um dos protocolos da camada da aplicação do modelo TCP/IP mais úteis no acesso à internet, uma vez que traduz domínios – que podem ser facilmente memorizados pelos utilizadores – para endereços IP.

Existem três tipos de servidores DNS, os *authoritative*, os *recursor* e os *forwarder*. Os servidores *authoritative* tratam da tarefa essencial do DNS, i.e. converter domínios em endereços. Os servidores *recursor* e *forwarder* são responsáveis por encaminhar as *queries* de DNS ao seu destino. Os *recursor* têm uma lista mais vasta de servidores authoritative e podem facilmente resolver o pedido de um utilizador, enquanto que os *forwarder* apenas envia *queries* de DNS para o seguinte servidor.

Existem dois tipos de servidores *authoritative*, os *master* ou primários e os *slave* ou secundários. Os servidores *master* são responsáveis por armazenar toda a informação necessária ao processo DNS. Desta forma, o servidor tem de possuir um local de armazenamento (base de dados ou

ficheiro), contendo todos os dados relativos a essa zona. Os servidores *slave* são usados em conjunto com os *master*, sendo importantes para casos de redundância, manutenção e *performance*. Para cada domínio existe sempre um servidor *master* e um ou mais servidores *slave*. Quando o servidor *master* está inoperacional, os pedidos são redirecionados para o servidor *slave*. Para evitar que os servidores necessitem constantemente de pedir informações, estes possuem um mecanismo de cache, que lhes permite memorizar localmente domínios e endereços que foram pedidos recentemente. De maneira a evitar o armazenamento de domínios desatualizados é definida uma data de expiração para cada domínio.

Este serviço utiliza os protocolos UDP e TCP para transporte de informação DNS. O protocolo UDP é utlizado para *queries* de resolução de um domínio, enquanto que o protocolo TCP é utilizado em duas situações: para respostas que excedam os 512 bytes ou para transferência de informações sobre uma determinada zona entre servidores *master* e *slave*.

GeoDNS é uma característica recente dos servidores de DNS que permite dividir os pedidos geograficamente através do seu endereço. Desta forma, torna-se possível que o utilizador obtenha na resposta de DNS o endereço do servidor mais próximo de si.

PowerDNS

O PowerDNS [46] é uma solução gratuita de DNS, lançada em 1999 pela empresa Holandesa PowerDNS sendo desenvolvido em C++ sob licença da GNU General Public License. Esta solução multiplataforma, tem o seu servidor *authoritative* na versão 3.4.1 e o *recursor* na versão 3.6.2. O PowerDNS dispõe de vários tipos de *backends*, permitindo ter ficheiros de zonas semelhantes ao BIND, base de dados relacionais e algoritmos de *load balacing* e *failover*. Esta solução de DNS tem suporte para IPv6, permitindo responder a pedidos relativos à nova versão do IP. Além disso possui características como: API [47] que permite controlar o seu estado (como adicionar domínios e zonas); *Scripts* [48] em Lua que permitem configurar o funcionamento do *recursor* (e.g. filtrar *queries* de DNS), tornando a sua gestão mais acessível.

BIND

O Berkeley Internet Name Domain (BIND) [49] é uma solução gratuita de DNS, lançada em 1984 por quatro estudantes da Universidade de Califórnia – Berkeley. Em 2012 esta solução passaria a ser da responsabilidade da Internet Systems Consortium (ISC). Esta solução encontra-se na versão 9.10.0 e é multiplataforma. Devido a problemas graves de segurança o BIND 9 foi completamente reescrito e só a partir desta versão passou a suportar base de dados relacionais e IPv6.

2.6. Escalabilidade

Escalabilidade é uma caraterística desejável numa arquitetura ou infraestrutura, mas que nem sempre é conseguida na sua totalidade. Permite que um determinado sistema suporte todos os acessos, mesmo na ocorrência de um aumento de carga excessivo, ou seja, o sistema mantem a qualidade de serviço independentemente dos acessos num determinado momento.

Existem duas abordagens de escalabilidade horizontal *(scale in/out)* e vertical *(scale up/down)*. A escalabilidade horizontal aumenta o número de nós/servidores no sistema, enquanto a escalabilidade vertical adiciona recursos a um nó/servidor no sistema.

Escalabilidade de serviços na cloud

Na *cloud* a abordagem mais utilizada é a horizontal, uma vez que permite acrescentar mais uma instância para o processamento de operações, ficando pronta a ser utilizada e eliminado tempos de inatividade, como acontece na escalabilidade vertical, em que os recursos da instância têm de ser modificados.

Na escalabilidade de serviços na cloud é possível distinguir duas técnicas: previsão e reação. A técnica de previsão é um algoritmo proactivo que, com base no historial dos recursos utilizados tenta determinar os recursos que serão necessários num determinado momento, de forma a não desperdiçar recursos desnecessariamente. Na técnica de reação o sistema reage às alterações, verificando se os recursos disponíveis são suficientes para a procura no momento – esta técnica já se encontra implementada em muitos IaaS. Estas duas técnicas podem ser utilizadas em conjunto. O projeto Heat [44] para o OpenStack [32], tem a vertente de escalabilidade reativa, isto é, a *stack* vai monitorizando as instâncias e conforme as métricas definidas como sendo a utilização de CPU, memória e rede este acrescenta ou retira recursos ao serviço. O algoritmo PRESS [50] é uma proposta que utiliza a técnica de previsão, para escalar o serviço e evitar que os recursos sejam desaproveitados. Este algoritmo tem por base a utilização do CPU, memória e rede para gerir os recursos disponíveis da melhor maneira. O algoritmo AutoFlex [51] combina as técnica preditivas e reativas, uma vez que apenas com técnica reativa os recursos podem ser mal aproveitados(e.g. podendo acontecer "picos" ocasionais que não justificam o acrescento de uma nova instância).

Apesar de tudo, a validação dos algoritmos não proveu de qualquer estratégia para otimizar os recursos de forma eficiente, não servindo portanto o propósito definido pelo algoritmo.

Técnica	Vantagem	Desvantagem
Reativa	Fácil de implementar, uma vez que através de MaaS consegue-se criar *thresholds* que permitem definir a quantidade de instâncias necessárias	Pode conduzir a recursos mal aproveitados, muito devido aos falsos "picos" ou a uma escalabilidade ineficiente devido a uma má leitura do consumo dos recursos
Preditiva	Bastante útil para serviços com padrões de acessos semelhantes durante o mesmo período de tempo	Histórico enganador, pode levar a que os recursos sejam mal geridos, alocando recursos desnecessariamente
Composto (Reativa e Preditiva)	Solução completa, que permite conjugar as suas vantagens e combater as suas desvantagens.	Difícil de implementar, devido à conjugação de duas técnicas distintas.

Tabela 8 - Diferenciação de técnicas de escalabilidade

Para este processo de escalabilidade na *cloud* é fulcral o serviço de monitorização, uma vez que reúne as métricas que as instâncias enviam, permitindo supervisionar os recursos que cada instância está a consumir num determinado momento.

Escalabilidade de DNS

O desempenho do DNS está muito dependente da rapidez com que o servidor de DNS consegue responder à carga que lhe são destinados Embora existam estudos demonstrativos [52]-[53] da *performance*, não chegam a abordar a forma como a arquitetura de DNS pode obter características de serviços *cloud*, nomeadamente a flexibilidade, elasticidade e *failover* e muito embora comecem a surgir propostas para obtenção destas características de serviços *cloud*, cada solução apresentada demostra uma especificação orientada aos seus próprios desafios e exigências. Estes inconiventes motivaram a que um serviço fosse criado de forma a poder suplantar estas indefinições. Um algoritmo de escalabilidade – com o auxílio de um sistema de monotorização – dota este processo com maior autonomismo e simplicidade permitindo, desta forma, uma melhor gestão de custos e recursos por parte do serviço.

2.7. Algoritmo de decisão

Um algoritmo de decisão permite auxiliar na previsão de resultados, indicando a melhor resolução. Existem dois tipos de algoritmos que destacam, nesta vertente, utilizando abordagens distintas. O *Decision Tree* é constituído por um conjunto de ramificações com decisões, que permitem verificar o melhor resultado. Como a árvore é criada com base em historial este algoritmo precisa de um tempo para tomar decisões corretas. Pode tornar o processo lento – dependente do sistema a que

destina – com o passar do tempo muito devido à construção da árvore cada vez que o serviço precisa de uma decisão.

O *Multiple Attribute Decision Making* é um tipo de algoritmo que a partir de múltiplos atributos toma uma decisão, este tipo de algoritmo é extremamente flexível uma vez que se pode adaptar a qualquer situação e é um algoritmo rápido pois apenas necessidade de construir matrizes com os vários atributos. Pelo lado negativo, se os atributos não forem bem selecionados pode não ter os resultados esperados.

Algoritmo	Vantagens	Desvantagens
Decision Tree	• Com base em historial	• Difícil implementação. • Precisa de treino • Lento
Multiple Attribute Decision Making	• Rápido • Flexível	• Suscetível a falhas

Tabela 9 - Algoritmos de decisão

Capítulo 3 : *DNS as a Service*

O serviço de DNS tem a funcionalidade de traduzir nomes de domínios em endereços IP. A mudança deste serviço para a *cloud* permite ter um sistema mais eficiente, com a gestão recursos necessários para sustentar o serviço, permitindo que o serviço seja escalado conforme a sua utilização e seja tolerante a falhas, obtendo-se desta forma um serviço consistente e com qualidade. Na figura 17, apresenta-se a forma como os componentes comunicam entre si para gerir o serviço de DNSaaS.

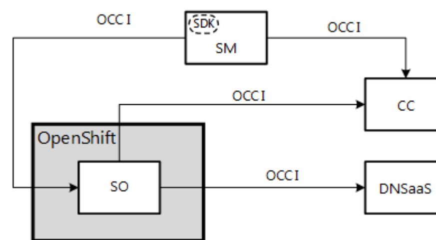


Figura 17 – Dependências do DNSaaS

O SM é o componente que trata de toda a logística na gestão do serviço de DNS, isto é, administra o(s) SO através do PaaS, OpenShift e comunica com o CC aquando da criação e eliminação do SO, para que o CC possa libertar os recursos. O SO orquestra o serviço de DNS e contacta o CC caso o serviço necessite de fazer alterações (escalabilidade) à sua infraestrutura. A comunicação entre os componentes é feita através da interface OCCI permitindo interoperabilidade e portabilidade, retirando deste modo possíveis restrições a soluções IaaS.

O SM contém um SDK que trata de gerir as operações do serviço (adicionar, pesquisar e remover domínios e registos) e operações sob o SO (adicionar, remover e obter *endpoints* do SO). Nas operações realizadas através do SDK, colaborei com um método que trata de iniciar o SO com a arquitetura pré-definida e quando todo o serviço estiver complemente iniciado devolve os *endpoints* relativos a esse SO, possibilitando as operações do serviço mencionadas anteriormente.

O algoritmo será incorporado no SO, ajudando-o na orquestração do DNSaaS. Nesta tarefa é auxiliado pelo MaaS, para verificação da utilização e qualidade que o serviço está a prestar no

momento. Sendo assim o trabalho incide essencialmente no SO tendo em conta a integração do algoritmo de escalabilidade, o SDK para inicialização do serviço e o DNSaaS.

3.1. Arquitetura

A arquitetura do serviço de DNS foi projetada para ser tolerante a falhas e eficiente, sendo escalável conforme a utilização do serviço. No planeamento da arquitetura surgiram duas grandes questões, relativas à *performance* e ao custo do sistema. A base de dados de um serviço de DNS é o componente mais instável e lento, podendo colocar em causa a qualidade do serviço prestado devido às suas limitações (conforme verificado nos testes de validação da arquitetura). Neste sentido, foram planeadas duas arquiteturas para compreender se a base de dados deveria ser centralizada – havendo consulta aos servidores DNS sempre que não há resposta na cache, podendo surgir com alguma frequência *bottlenecks* – ou distribuída – havendo uma base de dados *master* que envia o seu conteúdo, para as base de dados *slaves* que se encontram na instância do servidor de DNS, reduzindo assim a latência nas consultas à base de dados, mas podendo haver problemas na replicação da *master* e *slave* e a fiabilidade do serviço pode ser colocada em causa. A segunda pergunta envolve o custo de manutenção do sistema, pretendendo-se um sistema de qualidade/custo, com quantas instâncias ativas o serviço tem de ter para pode responder ao volume de tráfego no momento.

Na arquitetura do DNS existem dois *forwarders* a receber pedidos de DNS provindos dos clientes, permitindo que a carga seja distribuída e em caso de falha de um *forwarder* o outro continua a trabalhar e a responder aos pedidos enviados para o serviço. Na figura 18 e figura 19 – centralizada e distribuída, respetivamente – é possível visualizar as arquiteturas propostas, distinguíveis pelo modo como a base de dados opera, ambas contendo quatro componentes: DBaaS, DNS Server, DNS Forwarder e API.

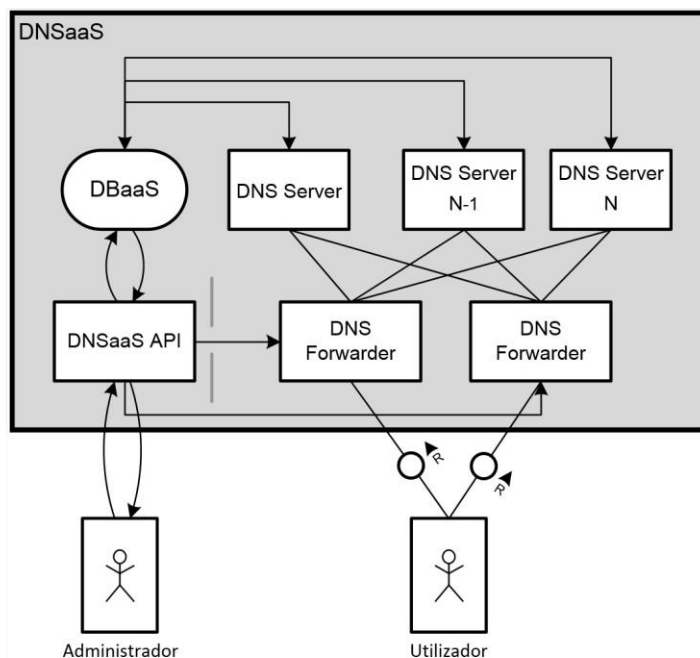


Figura 18 - Arquitetura com base de dados centralizada

O DBaaS funciona como uma base de dados *master*, onde toda a informação relativa ao DNS é armazenada, mantendo os domínios e registos para os diferentes *tenants*. Na arquitetura com base de dados distribuída este componente é ainda responsável por replicar os dados para as outras base de dados nos servidores de DNS. As base de dados nos servidores de DNS são utilizadas na sua maioria em operações de leitura, contendo otimizações, como índices nos atributos mais requisitados, como partir dos registos A devolver o IP.

O DNS Server é o componente essencial do serviço de DNS, uma vez que este trata da conversão nome-endereço, suportando geodns. O serviço de DNS começa com um servidor de DNS, mas qualquer uma das duas arquiteturas suporta um número dinâmico de servidores de DNS, permitindo que o serviço seja escalável.

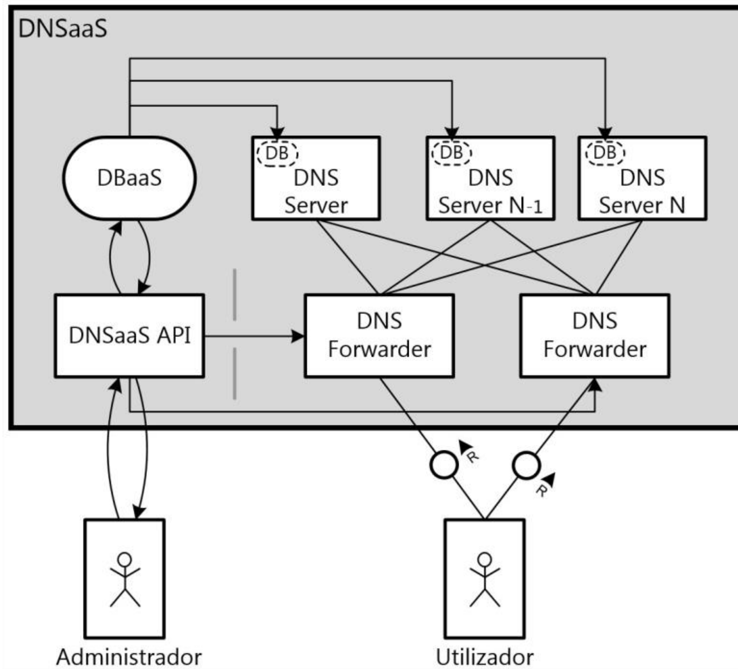


Figura 19 - Arquitetura com base de dados distribuída

O DNS Forwarder é o *frontend* do serviço para o utilizador, sendo que a principal função deste componente é encaminhar os pedidos de DNS para servidor(es) de DNS, utilizando um algoritmo round-robin para distribuição de carga. É de salientar que o *forwarder* tem em conta o estado do(s) servidor(es) DNS para encaminhar os pedidos para o(s) servidor(es) com maior *performance*. São utilizados dois *forwarders* para evitar quer falhas quer o congestionamento num só forwarder.

A API é o *frontend* do serviço para o administrador, permitindo-lhe a configuração do serviço. Este componente mantém o serviço a gerir-se sozinho e a funcionar corretamente. É um intermediário durante operações CRUD de domínios e registos, mantendo a BDaaS atualizada com os serviços disponíveis. Este componente é ainda responsável por manter a lista de servidores DNS atualizados e informar os *forwarders*, para que ele possa distribuir a carga e funcionar corretamente.

Capítulo 4 : Validação da arquitetura

Como descrito anteriormente, foram consideradas duas abordagens: base de dados centralizada e distribuída. Para validar estas arquiteturas foram realizados testes de *stress* de forma analisar o comportamento das duas arquiteturas perante um grande volume de tráfego. Para simulação de tráfego foi utilizado a ferramenta dnsperf [54]. Nos testes cada cliente envia 50.000 pedidos de DNS e o número de clientes pode variar de 1, 10, 25, 35 ou 50, podendo no máximo enviar 2.500.000 *queries*. Para confirmação e comparação de valores, cada teste foi executado quatro vezes, tendo em conta que os valores obtidos estavam na mesma gama de valores não se justificando mais repetições. A Tabela 10 permite mostrar como os diferentes testes foram configurados. Nos testes efetuados foi sempre testado como o primeiro pedido, ou seja, foi desativada a *cache* para permitir visualizar o comportamento no "pior cenário".

Para obtenção das métricas de cada um dos componentes foi utilizado um MaaS com o agente Zabbix [55] a obter métricas com intervalo de um segundo. As figura 20 e figura 21 permitem visualizar como foram efetuados os testes às duas arquiteturas.

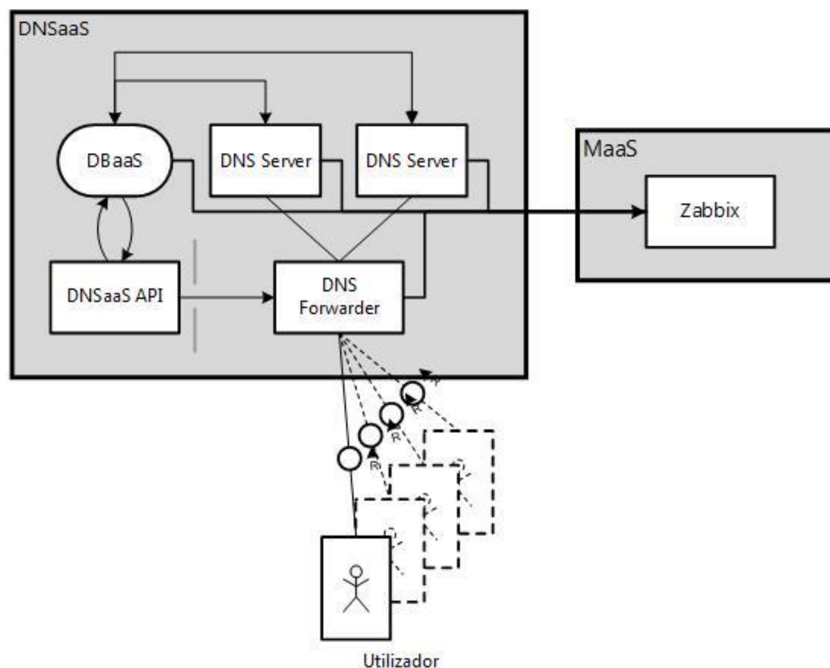


Figura 20 - Testes com base de dados centralizada

Na arquitetura centralizada, existe apenas uma base de dados que lida com todos os pedidos dos servidores de DNS, enquanto que na arquitetura distribuída, existe uma base de dados em cada servidor de DNS. Ambas as base de dados foram configuradas para utilizarem a base de dados em memória utilizando MySQL.

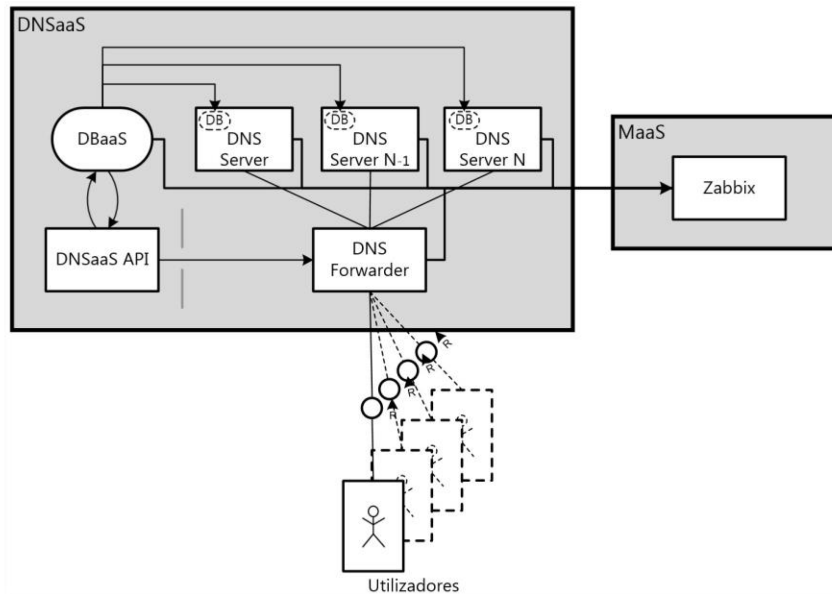


Figura 21 - Testes com base de dados distribuída

Nos testes foi utilizado apenas um *forwarder*, apesar de na arquitetura estarem dois representados, uma vez que o objetivo dos testes era a avaliar a arquitetura para escalar, visualizando o comportamento dos servidores de DNS na resolução de um elevado número de pedidos DNS. O forwarder utilizou o PowerDNS com as configurações padrão do recursor.

Durante os testes, o número de servidores DNS varia entre 1 e 3, estando conectados com o forwarder. O servidor DNS utilizou o PowerDNS com as configurações *standard* do *authoritatitive*.

Os DNS Servers estavam configurados com 2 vCPUs e 4 GB de memória (flavor m1.medium), enquanto os DNS Forwarder possuíam 4 vCPUs e 8 GB de memória (flavor m1.large). Foram utilizadas ligações de Gb, entre as máquinas, para reduzir latência e maximizar throughput.

Id do Teste	Nº de DNS *Servers*	Nº de Clientes	Base de dados	*Queries* enviadas	Estado
1_C_1	1	1	Centralizada	50.000	Concluído
1_C_10	1	10	Centralizada	500.000	Concluído
1_C_25	1	25	Centralizada	1.250.000	Concluído
1_C_35	1	35	Centralizada	1.750.000	Falhou
1_C_50	1	50	Centralizada	2.500.000	Falhou

1_D_1	1	1	Distribuída	50.000	Concluído
1_D_10	1	10	Distribuída	500.000	Concluído
1_D_25	1	25	Distribuída	1.250.000	Concluído
1_D_35	1	35	Distribuída	1.750.000	Falhou
1_D_50	1	50	Distribuída	2.500.000	Falhou
2_C_1	2	1	Centralizada	50.000	Concluído
2_C_10	2	10	Centralizada	500.000	Concluído
2_C_25	2	25	Centralizada	1.250.000	Concluído
2_C_35	2	35	Centralizada	1.750.000	Concluído
2_C_50	2	50	Centralizada	2.500.000	Falhou
2_D_1	2	1	Distribuída	50.000	Concluído
2_D_10	2	10	Distribuída	500.000	Concluído
2_D_25	2	25	Distribuída	1.250.000	Concluído
2_D_35	2	35	Distribuída	1.750.000	Concluído
2_D_50	2	50	Distribuída	2.500.000	Concluído
3_D_1	3	1	Distribuída	50.000	Concluído
3_D_10	3	10	Distribuída	500.000	Concluído
3_D_25	3	25	Distribuída	1.250.000	Concluído
3_D_35	3	35	Distribuída	1.750.000	Concluído
3_D_50	3	50	Distribuída	2.500.000	Concluído

Tabela 10 - Configuração dos testes

As métricas recolhidas dos três componentes utilizador, *forwarder* e servidor DNS estão reunidas na tabela 11. As métricas da *performance* do sistema (utilização do CPU e memória) também foram recolhidas nos servidores DNS e no *forwarder*.

Componente	Métrica	Descrição
Utilizador	Queries Throughput (qps)	Contador de *queries* respondidas por segundo
	Queries Latency (ms)	Tempo despendido pelo DNSaaS, na resolução da *query* DNS
DNS Forwarder	Out Queries	Contador de *queries* respondidas com sucesso
	Answer Slow	Contador de *queries* respondidas em mais de 1 segundo
	Answer 10-100	Contador de *queries* respondidas entre 10 e 100 milissegundos
	Answer 100-1000	Contador de *queries* respondidas entre 100 e 1000 milissegundos

	Concurrent Queries	Contador de *queries* recebidas em paralelo
	Outgoing Timeouts	Contador de *queries* que atingiram o tempo máximo (*timeout*)
	Lantecy (ms)	Tempo despendido pelo *forwarder*, na resolução da query DNS
	Questions	Contador de *queries* processadas
	ServFail Answers	Contador de respostas ServFail. ServFail pode acontecer quando não consegue responder à *query* de DNS, devido a algum problema ou má configurado. Entre estes problemas podem estar erros (*timeout* ou outra exceção) na base de dados
	Throttled Out Queries	Contador de resposta *throtlled*. Isto acontece se um servidor DNS não responde ou se as suas respostas são inúteis, podendo ocorrer um bloqueio à *query* causando um atraso.
DNS Server	Latency (ms)	Tempo despendido pelo servidor de DNS, na resolução da query DNS
	Queries	Contador de *queries* processadas
	Timeout packets	Número de *queries* que atingiram o tempo de *timeout*
	ServFail packets	Contador de respostas ServFail

Tabela 11 - Métricas dos componentes

4.1. Avaliação

Nas avaliações da arquitetura foram considerados somente os testes com todas as *queries* enviadas e que tenham sido executados até ao fim. Com os resultados dos testes foram feitas duas avaliações: avaliação à *performance* da base de dados distribuída e centralizada e estudo da *performance* da variação de servidores DNS. A avaliação da arquitetura do serviço DNSaaS resultou na elaboração de um *paper*, onde me foi possível contribuir com a descrição e análise à arquitetura do serviço de DNS [12].

BD Distribuída vs BD Centralizada

Na avaliação da base de dados distribuída e centralizada, pretende-se averiguar qual a melhor abordagem para o comportamento da base de dados, tendo em conta a *performance* do serviço.

Na figura 22 é possível visualizar a latência no cliente. Tanto a distribuída como a centralizada, com apenas um servidor DNS só chegaram até aos 25 clientes (i.e. 1.250.000 *queries*), não revelando grande diferença entre distribuída e centralizada. Nos testes com dois servidores DNS, o teste da base de dados centralizada com 50 clientes não foi concretizado, devido aos tempos de resposta serem demasiado longos, levando a não conclusão do teste. Nesta comparação os resultados com

a base de dados distribuída são superiores, obtendo um ganho de cerca de 150%, nos testes de 35 clientes. Sendo assim, nesta primeira comparação a BD distribuída evidência resultados superiores, e portanto, mais favoráveis.

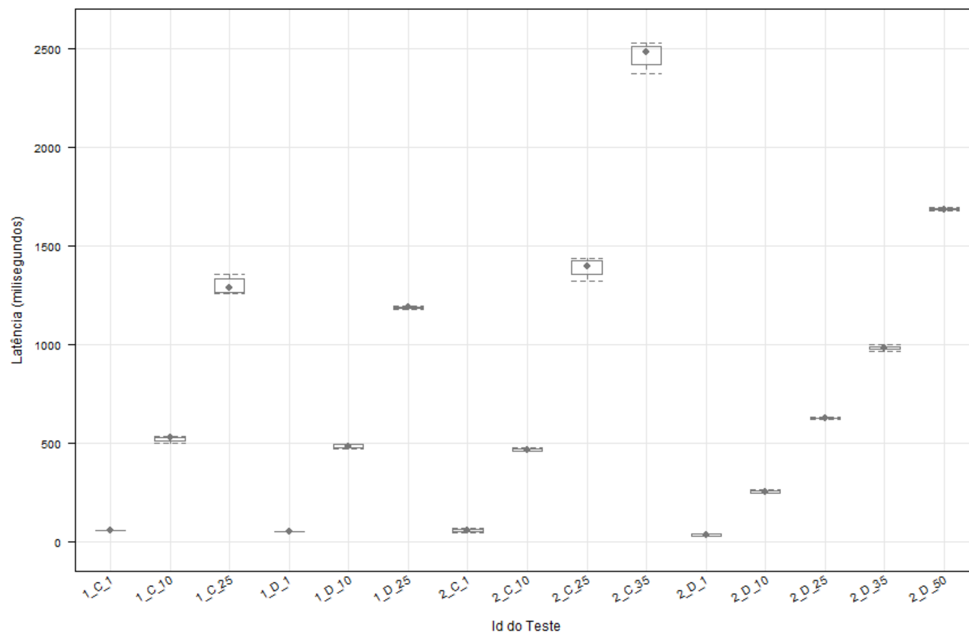


Figura 22 - Latência no teste distribuída vs centralizada no cliente

Na figura 23 é possível analisar a quantidade de *queries* que são respondidas por segundo no cliente. Nos testes com um servidor DNS nota-se uma ténue melhoria, apesar de não ser muito significativa. Já nos testes com dois servidores verificam-se melhores resultados para a base de dados distribuída registando-se casos com melhorias em cerca de 100%, com 1 cliente e 50.000 *queries*. De notar que nesta métrica, a base de dados centralizada, quer com um ou dois servidores DNS, tem valores muito semelhantes denotando-se a concorrência de acesso à base de dados, havendo então um *bottleneck*.

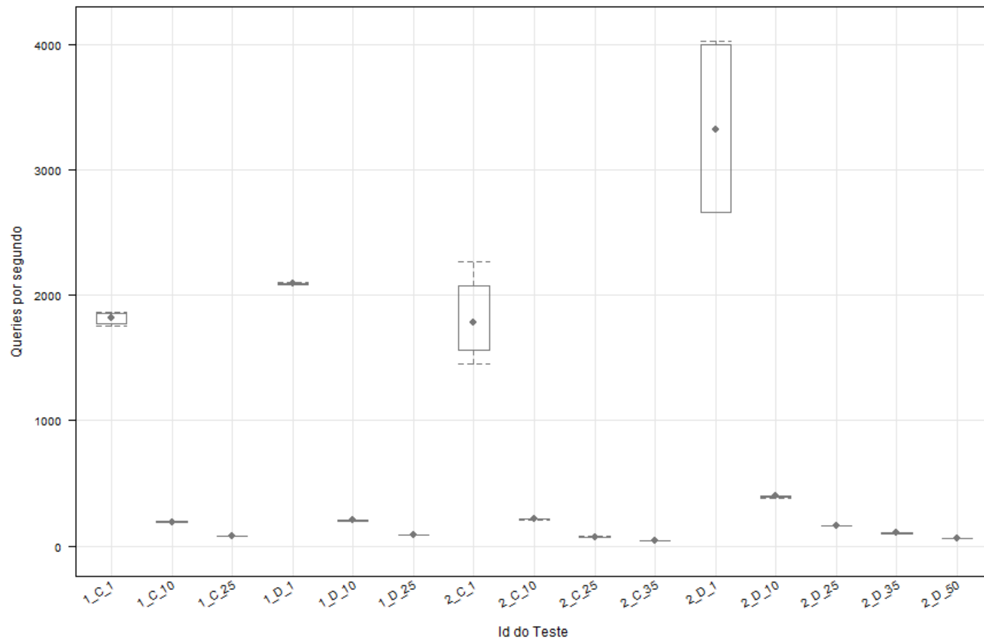


Figura 23 - QPS no teste distribuída vs centralizada no cliente

Na figura 24 destaca-se a subtração entre as *queries* enviadas e perdidas no cliente. Nesta relação é visível que os testes sobre a base de dados distribuída superaram a base de dados centralizada, que em casos extremos (teste com 2 servidores DNS e 35 clientes) chega a perder quase 400 *queries*.

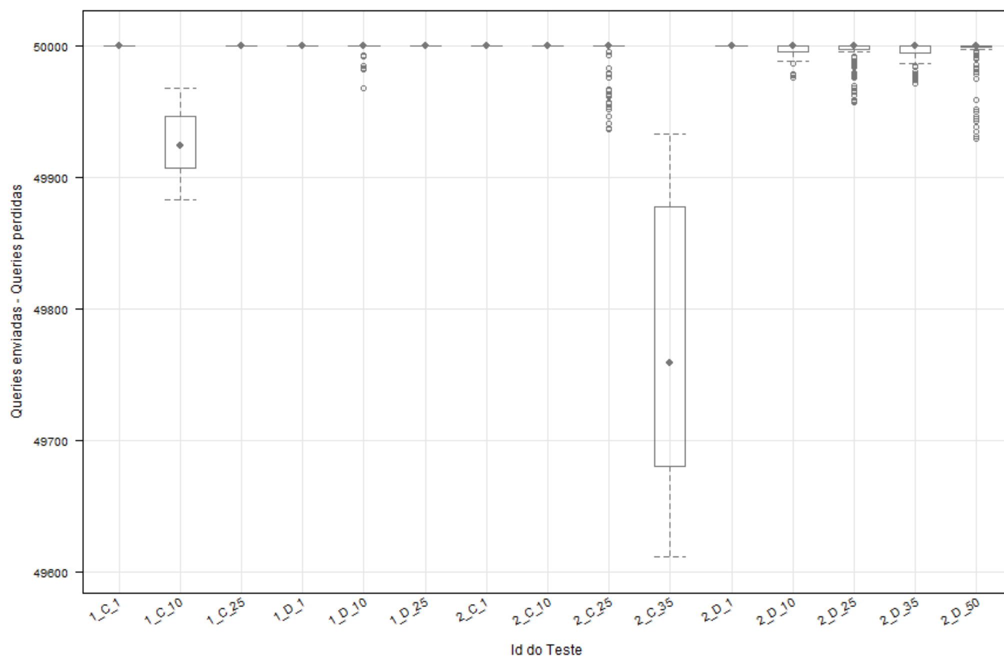


Figura 24 - *Queries* recebidas no teste distribuída vs centralizada no cliente

A título de conclusão, os resultados mostram que a base de dados distribuída obtém melhores resultados, tendo sido portanto esta a arquitetura escolhida para o serviço de DNS.

Número de Servidores

Concluída a análise sobre a forma como a base de dados se coaduna com o serviço, efetuou-se uma avaliação – sobre a arquitetura distribuída – ao número de servidores necessários para responder à procura atual. Para isso, aos dois servidores iniciais, acrescentou-se um adicional, para que – com três no total – seja possível analisar a evolução do comportamento e *performance* perante um excesso de carga.

Na figura 25 é apresentada a latência no cliente, onde é possível verificar a evolução dos pedidos com o aumento do número de servidores DNS. Com apenas um servidor obtém-se valores muito aceitáveis, obtendo uma média de 500 ms, com 10 clientes (i.e. 500.000 *queries*), conseguindo-se perceber que um servidor é suficiente para uma utilização simples do serviço, podendo este iniciar com apenas um servidor DNS e garantir respostas inferiores 1.250.000 *queries*, uma vez que os testes mostram que com esse volume e *queries*, as respostas têm uma latência superior a 1 segundo. Com dois servidores temo um ganho, em relação a um servidor, em cerca de 50%, em todos os testes possíveis de comparar. Quando for necessário o serviço ter dois servidores DNS pode esperar-se que este responda com latência inferior a 1 segundo até 1.750.000 *queries*, obtendo desta forma uma boa *performance*. Por fim, com três servidores DNS o serviço consegue dar resposta 2.500.000 *queries* com uma latência de cerca de 1 segundo.

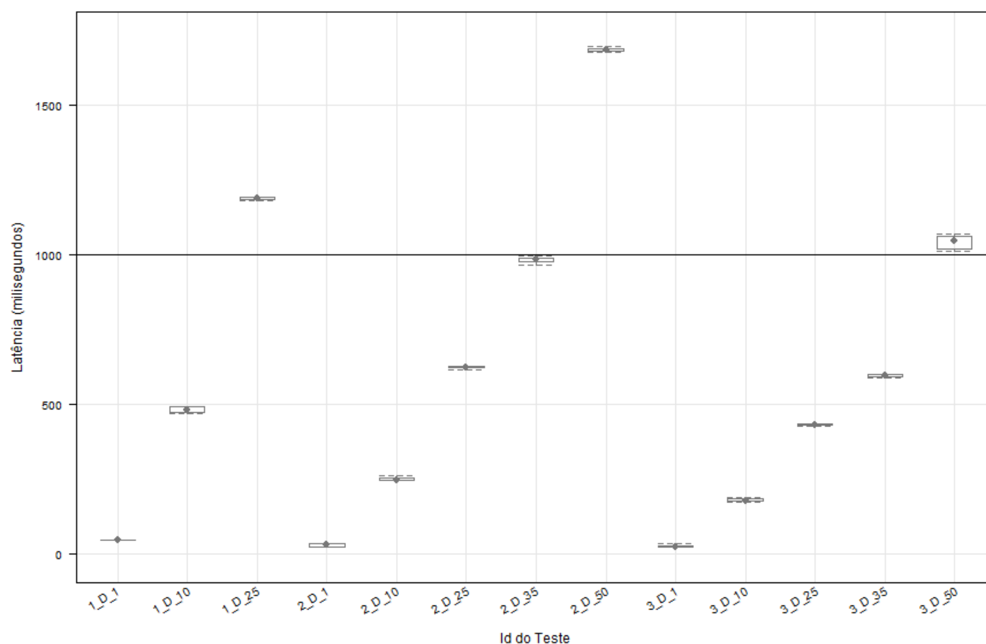


Figura 25 - Latência no teste de número de servidores no cliente

Capítulo 5 : Algoritmo de Escalabilidade

O algoritmo de escalabilidade foi projetado para garantir que o serviço de DNS tenha a qualidade e *performance* esperada, sem que para isso consuma recursos desnecessários. Para atingir este mesmo objetivo foi especificado um algoritmo semirreativo, ilustrado na Figura 26, com 2 níveis de deliberação: no primeiro nível existem 5 KPIs e num segundo nível – dependente do primeiro – onde está presente um algoritmo de decisão MADM. No primeiro nível os 5 KPIs correspondem às métricas de latência, percentagem de *answer slow* e número de questões medidas no *forwarder* e utilização do CPU e memória nos servidores *backend* do DNS. Estas métricas servem de indicadores de forma a verificar se os limites de *performance* estão a ser respeitados. A escolha destas métricas deve-se à análise da arquitetura (Capítulo 4) que permitiu verificar os melhores indicadores de desempenho. Estes limites (*thresholds*) são definidos de acordo com a avaliação feita anteriormente para validar a arquitetura. No caso de ser excedido alguns destes *thresholds* e caso seja detetado que o serviço precisa de alterações à sua estrutura (*scale in/out*) é utilizado o segundo nível de deliberação com um algoritmo MADM que, com base em diversas métricas a decisão de escalar ou não o serviço de DNS.

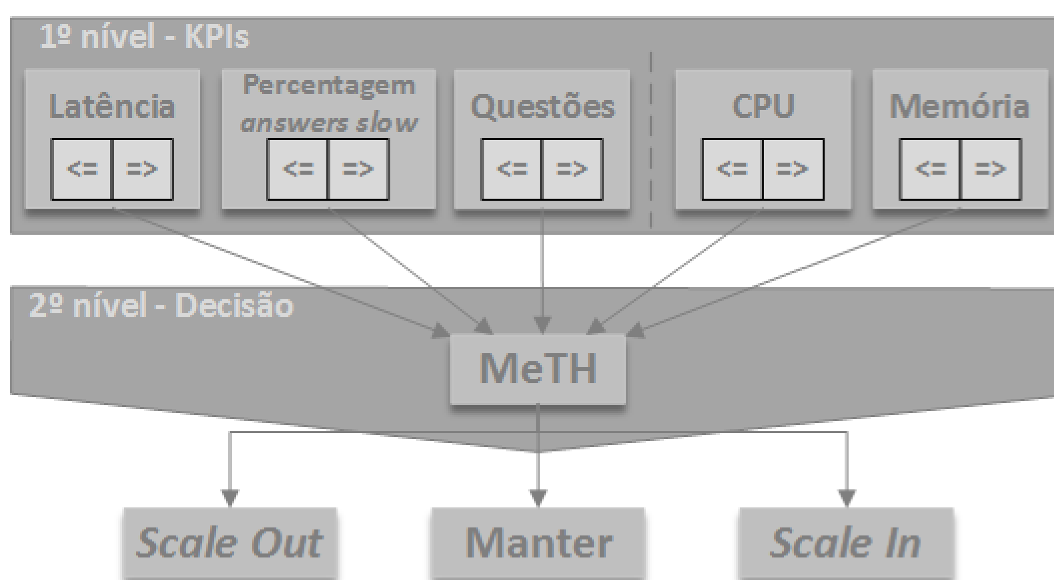


Figura 26 - Estrutura do algoritmo

O algoritmo MADM escolhido foi o MeTHODICAL (MeTH) [56], sobretudo por cumprir todos os requisitos pretendidos, ser rápido a responder e de fácil implementação foram aspetos que se revelaram decisivos para a seleção deste algoritmo. O MeTH tem um conjunto de características úteis que permitiram especificar o seu comportamento; entre estas características distinguem-se o envio de conjunto de métricas; a diferenciação das métricas através de benefícios/custos; e ainda a especificação dos pesos de cada métrica. Com a informação enviada o algoritmo retorna um *score* relativamente à quantidade de servidores de DNS que o serviço deve ter, tendo em conta a carga atual e mantendo o consumo reduzido de recursos. As métricas enviadas para o algoritmo MeTHODICAL são as *questions*; *concurrent queries*; *answer slow*; *timeout*; latência retiradas do *forwarder*; latência de *scaling* (*provisioning* e *booting*); utilização do CPU e memória dos servidores DNS e por último; o preço (por hora) de manter a estrutura do serviço.

O algoritmo MeTHODICAL, para tomar a decisão de escalar, recebe informação com a previsão dos valores que as métricas tendo sempre em conta o número de servidores de DNS no serviço e o peso que cada métrica, têm na tomada de decisão de escalar, como demonstrado na Tabela 12. Quando a decisão do algoritmo é *scale in* são enviadas as linhas $x-1$ e x, e quando a decisão é *scale out* são enviadas as linhas x e $x+1$ permitindo a comparação com o estado atual (número de servidores atuais) com o estado proposto pelo algoritmo.

Nº Servidores DNS	Benefício		Custo						
	Questions	*Concurrent Queries*	Latência	*Answer Slow*	*Timeout*	Latência de *Scaling*	Preço	CPU	Memória
$x-1$	Questions	Concurrent Queries	Previsão Latência	Previsão Answer Slow	Previsão Timeout	Latência de Scaling	$(\text{preço/hora}) \times (x-1)$	CPU $\times 1{,}5$	Memória $\times 1{,}5$
x	Questions	Concurrent Queries	Latência	Answer Slow	Timeout	0	$(\text{preço/hora}) \times x$	CPU	Memória
$x+1$	Questions	Concurrent Queries	Previsão Latência	Previsão Answer Slow	Previsão Timeout	Latência de Scaling	$(\text{preço/hora}) \times (x+1)$	CPU $\times 0{,}5$	Memória $\times 0{,}5$

Tabela 12 - Métricas enviadas para o algoritmo MeTH

Esta informação é resultado da validação da arquitetura do DNS, onde se conseguiram relações entre as métricas de forma que permitiu a sua escolha, verificar a sua evolução e os pesos que têm consoante o número de servidores. Relativamente aos benefícios, nomeadamente as *questions* e *concurrent queries* os valores mantém-se uma vez que não existem previsões pois são fatores que um algoritmo reativo não consegue calcular, por carecer de historial que não dispõe. Além disto,

pretende-se que o algoritmo reaja à carga que os servidores enfrentam num determinado momento.

Da anterior análise da arquitetura do serviço (Capítulo 4) pode verificar-se que há uma relação linear entre a latência e o número de clientes e servidores. A análise de variação permite estabelecer a seguinte equação para a latência.

$$\text{Previsão Latência (ms)} = (\alpha - \beta_1 \times \text{Servidores} \times \beta_2 * \text{Clientes}) \times 0.001$$

em que:

$$\text{Previsão Latência} \geq 0, \alpha = 15582.2,\ \beta_1 = 2180.92, \beta_2 = 11861.9$$

Este modelo traduz o aumento do número de clientes consoante uma maior latência, diminuindo com a introdução de mais servidores. Relativamente à métrica *answers slow* o modelo que se aferiu corresponde à seguinte equação.

$$\text{Previsão Answers Slow} = \alpha - \ \text{Servidores} + \beta_1 \times \text{Clientes}^2 + 0.4 * \text{Clientes} * \text{Servidores}$$

em que:

$$\text{Previsão Answers Slow} \geq 0, \alpha = -25,\ \beta_1 = 30, \beta_2 = 0.4$$

A introdução de servidores leva a um decréscimo em *answers slow*. Contudo o aumento de clientes em simultâneo leva a um aumento. Nos *timeouts* a análise traduziu-se na seguinte equação.

$$\text{Previsão Timeout} = \alpha + \ \beta_1 \times \text{Clientes} + \beta_2 \times \text{Clientes} \times \text{Servidores}$$

em que:

$$\text{Previsão Timeout} \geq 0, \alpha = -32812.6,\ \beta_1 = 3117.6, \beta_2 = 402.5$$

Este modelo mostra que quando o número de clientes aumenta os *timeouts* acompanham esse crescimento, diminuindo com a introdução de mais servidores.

A Latência de *Scaling* no estado atual (x) não tem qualquer custo uma vez que o serviço não tem que fazer nenhuma atualização à sua arquitetura, enquanto que quando é necessário escalar $(x - 1$ ou $x + 1)$ esta métrica terá o valor da última Latência de *Scaling*.

O cálculo do preço por hora está diretamente ligado com o número de servidores DNS. Por fim o CPU e a memória na previsão do *scale in* têm o dobro da leitura atual enquanto que durante o *scale out* têm metade do valor atual.

Como referido anteriormente o algoritmo de decisão permite organizar as métricas entre benefício e custo, divisão representada na Tabela 13. Os benefícios devem tender para +∞ enquanto que os custos devem tender para 0. Nos benefícios estão as métricas *questions* e *concurrente queries*, em que *questions* têm um valor superior às *concurrent queries*.

Métrica	Benefício		Custo						
	Questions	*Concurrent Queries*	Latência	*Answer Slow*	*Timeout*	Latência de *Scaling*	Preço	CPU	Memória
Peso	60	40	25	20	18	20	0	12	5
Total	100		100						

Tabela 13 - Peso das métricas

Já relativamente aos custos, as métricas são a latência; *answer slow*; *timeout*; latência de scaling; preço; CPU e memória. As métricas que caracterizam o serviço de DNS têm um maior destaque. A latência é a métrica mais importante do custo, informando do tempo de resposta às *queries* por parte do serviço, ou seja uma métrica essencial para aferir a eficiência do serviço. A métrica *answer slow* mostra se o serviço não está a ter uma *performance* ideal apresentando as *queries* em que a resposta foi superior a um segundo, tendo portanto esta métrica um peso revelante nos custos(i.e. o serviço está a ter degradação de performance). Os *timeouts* são as *queries* que excedem os dois segundos e que são descartadas pelo serviço, transmitindo se o serviço está a ser eficiente, tendo por isso um peso razoável nos custos. A Latência de *scaling* traduz-se no tempo de *provisioning* e *booting* das instâncias e tem um grande impacto na altura de escalar, uma vez que permite informar se é viável efetuar alterações ao serviço tendo em conta o tempo, ou seja, verifica se é rentável escalar o serviço tendo em conta o tempo que demora a atualizar o mesmo. O preço não tem qualquer impacto neste caso, uma vez que o *testbed* é privado e não implica quaisquer custos horários, mas em casos em que o *testbed* é público (e.g. LunaCloud, Microsoft Azure) esta variável pode ter peso uma vez que os recursos são pagos. O CPU tem um peso razoável nos custos, pois apesar de transmitir o congestionamento das instâncias, esse valor não transmite a eficiência das mesmas. A memória é o recurso menos utilizado logo, o peso é o inferior às restantes métricas de custo.

5.1. Desenvolvimento

O algoritmo de escalabilidade foi implementado em python, mais concretamente na versão 2.7.10, por ser o utilizado no projeto inserido. Como referido no capítulo 3 sobre a arquitetura do serviço de DNS, o algoritmo é implementado no SO. A Figura 27 tem representado as classes presentes no SO e a comunicação entre elas. O SO é constituído pelas classes ServiceOrchestrator, SOExecution, SOConfigurator, SODecision, PolicyEngine, Metrics e MeTH. Antes de iniciar o desenvolvimento, as últimas duas classes referidas – Metrics e MeTH – não existiam antes do desenvolvimento do algoritmo. De salientar que o seu desenvolvimento implicou mudanças nas restantes classes. A classe central é a ServiceOrchestrator, que cria as ligações às classes SOExecution, SODecision e SOConfigurator. A SOExecution é a classe principal e tem como função controlar o comportamento do serviço DNS. Os métodos que esta classe suporta são:

- **design:** métodos cuja finalidade é iniciar os preparativos para o novo serviço.
- **deploy:** método responsável por realizar o *deploy* das instâncias do serviço.
- **provision:** método que fornece recursos que o serviço necessita.
- **dispose:** método que elimina todas as instâncias e recursos que o serviço alocou.
- **state:** método que retorna o estado do serviço e os respetivos *endpoints.*
- **update:** método que possibilita a atualização das instâncias e recursos do serviço.

Os métodos da classe têm ligações com o que foi anteriormente referido no capítulo ciclo de vida dos serviços (2.2 *Mobile Cloud Networking*).

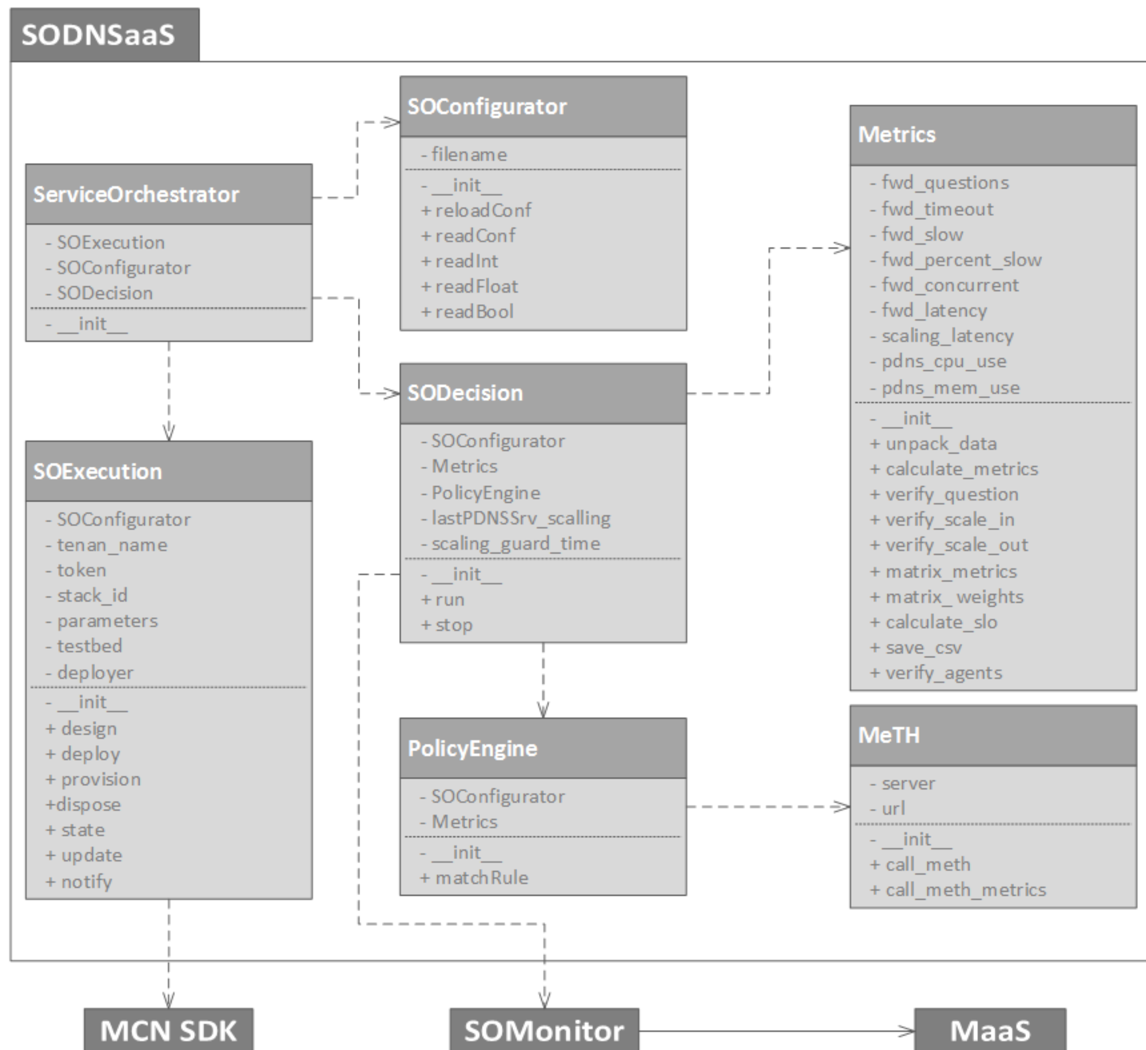


Figura 27 - Diagrama de classes do SO

A classe SOConfigurator contém a informação existente no ficheiro de configuração do serviço, tendo portanto a finalidade de comunicar essas configurações ao SO para ser executado com as opções pretendidas. Os métodos que esta classe contem são:

- **reloadConf:** método que atualiza as configurações.

- **readConf:** método que retorna uma *String* com a configuração da variável pretendida.
- **readInt:** método que retorna um *Int* com a configuração da variável pretendida.
- **readFloat:** método que retorna um *Float* com a configuração da variável pretendida.
- **readBool:** método que retorna um *Boolean* com a configuração da variável pretendida.

A classe SODecision é onde todas as operações de escalabilidade do serviço são realizadas, sendo portanto a peça central na operação de escalabilidade. Esta classe faz-se representar pelos seguintes métodos:

- **run:** método que tem a seu cargo toda a logística de escalabilidade.
- **stop:** método com a responsabilidade de parar a escalabilidade.

SODecision comunica com a SOMonitor para obter o valor das métricas, com Metrics para tratar das métricas obtidas e com a PolicyEngine para saber quantos servidores de DNS o serviço deve ter. Sendo assim a classe PolicyEngine é a classe que toma as decisões na escalabilidade, informando a SODecision sobre a estrutura que o serviço deve ter, tendo em conta as métricas. Assim o método que esta classe tem é:

- **matchRule:** método com a finalidade de encontrar o melhor equilíbrio na escalabilidade do serviço.

A PolicyEngine comunica com a classe MeTH para este algoritmo decidir a melhor arquitetura para o serviço. A classe Metrics é responsável por transformar as métricas obtidas a o partir do MaaS e efetuar as verificações nesses valores transformados, sendo os métodos de classe os seguintes:

- **unpack_data:** método que descompacta as métricas.
- **calculate_metrics:** método que realiza cálculos sobre os valores descompactados.
- **verify_question:** método que verifica se as *questions* estão no intervalo desejado.
- **verify_scale_in:** método que verifica se é necessário remover instâncias.
- **verify_scale_out:** método que verifica se é necessário adicionar instâncias.
- **matrix_metrics:** método que constrói a matriz, com as métricas, enviada para o algoritmo MeTH (exemplo da matriz na Tabela 12).
- **matrix_weights:** método que constrói a matriz, com os pesos das métricas, enviada para o algoritmo MeTH (exemplo da matriz na Tabela 13).
- **calculate_slo:** método que calcula os SLO.
- **save_csv:** método que guarda as métricas num ficheiro.
- **verify_agents:** método que verifica se os agentes do Zabbix estão disponíveis.

Por fim a classe MeTH é responsável por contactar o algoritmo MeTH. Assim os seus métodos são:

- **call_meth:** envia o pedido ao algoritmo MeTH.
- **cal_meth_metrics:** constrói os cabeçalhos necessários a enviar.

Características

O algoritmo foi desenvolvido com certas particularidades considerando as suas necessidades e problemas encontrados. O ciclo tem a periocidade de um segundo para fazer as verificações ao serviço de modo a certificar-se que o serviço se encontra nas condições pretendidas, é um claro exemplo de uma necessidade do algoritmo. Outra particularidade implementada no algoritmo é o *guarding time*, que após ocorrência de uma operação de escalabilidade o algoritmo bloqueia temporariamente – 60 segundos é o pré-definido uma vez que foi o tempo médio que o serviço demora a normalizar – outras tentativas, de escalar de forma a tentar estabilizar o serviço com as alterações efetuadas. As características anteriormente referidas, os KPIs do primeiro nível da estrutura algoritmo e os pesos de cada métrica podem ser devidamente configuradas - sendo esta mais uma característica implementada no algoritmo – permitindo que o algoritmo tenha o comportamento desejado no ambiente implementado, tornando-se portanto num algoritmo adaptável à situação pretendida.

Muitas das métricas que o MaaS (Zabbix) reporta são cumulativas dificultando a verificação o valor dessas métricas no momento atual. Para contornar este mesmo inconveniente as métricas que se encontram nestas condições possuem um *array* de duas posições – muito semelhante a logs circulares – onde, na primeira posição se encontra o valor obsoleto que é substituído pelo antigo atual e na segunda posição o antigo atual é substituído pelo novo atual, permitindo deste modo saber o valor atual das métricas cumulativas. A Figura 28 ilustra esse processo.

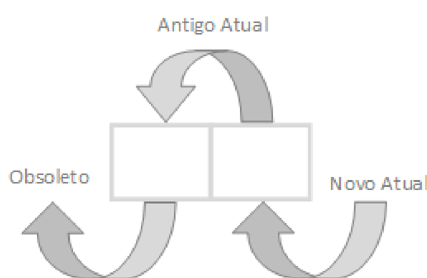


Figura 28 - *Array* cumulativo de duas posições

Devido ao dados do Zabbix não serem completamente consistentes, existindo casos de falhas como demonstra a Figura 29, onde uma métrica cumulativa deixa de existir momentaneamente, chegando inclusive a perder a contagem e noutros casos em que devido a atrasos as métricas não chegam a tempo foi criado um mecanismo para tentar suprir esta falha tendo por base o *array*

cumulativo de duas posições que permite verificar se o valor atual da métrica se encontra dentro do expectável.

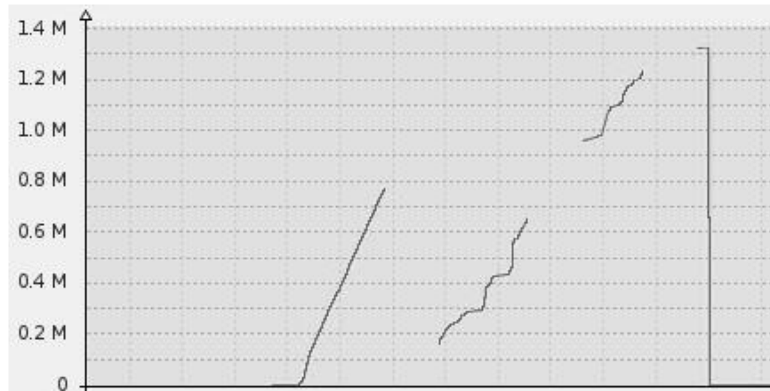


Figura 29 - Falhas na métrica

Atividade do algoritmo

A Figura 30 ilustra o fluxo de dados entre as várias classes no decorrer dos processos de escalabilidade. Assim para o algoritmo iniciar a operação é necessário que seja executado o *provision* no serviço – devido à operação de *provisioning* garantir que o serviço tem todos os recursos que necessidade. O algoritmo de escalabilidade é um ciclo constante com um delay de 1 segundo para garantir que os valores correspondem a segundos diferentes. Na primeira vez que o algoritmo é executado é necessário criar comunicação com o Zabbix de forma a conseguir o acesso às métricas. Depois da verificação deste passo atualizam-se as configurações de forma a garantir que o algoritmo está a correr com as últimas configurações. Segue-se depois a verificação dos agentes do Zabbix, uma vez que se estes estiverem indisponíveis as métricas não são consistentes e portanto o processo de escalabilidade não pode avançar enquanto os agentes não estiverem disponíveis. Garantida a disponibilidade dos agentes são obtidas e organizadas as métricas de forma a tomar decisões com base nas mesmas.

Após isso é necessário verificar se é possível escalar (*Guarding Time,* estado do serviço), caso não seja possível o algoritmo guardas métricas e reinicia o processo, caso seja possível escalar, o algoritmo vai verificar se existe necessidade de escalar, começando este procedimento de dois passos por, primeiramente verificar pelos KPIs de indiciadores que levem a efetuar alterações ao serviço, caso não se verifique necessário qualquer alteração guarda as métricas e recomeça todo o processo, caso os indicadores identifiquem uma possibilidade de escalar o serviço o segundo passo inicia. Este comunica com o algoritmo de decisão MeTH e caso a sua deliberação seja manter a estrutura atual as métricas são guardadas e o processo é reiniciado caso a sua decisão seja escalar o serviço é necessário atualizar o número de instâncias e após este processo de atualização o

algoritmo guarda as métricas e reinicia todo o processo de escalabilidade. O algoritmo de escalabilidade é um processo contínuo e que só termina quando o SO pretende eliminar o serviço.

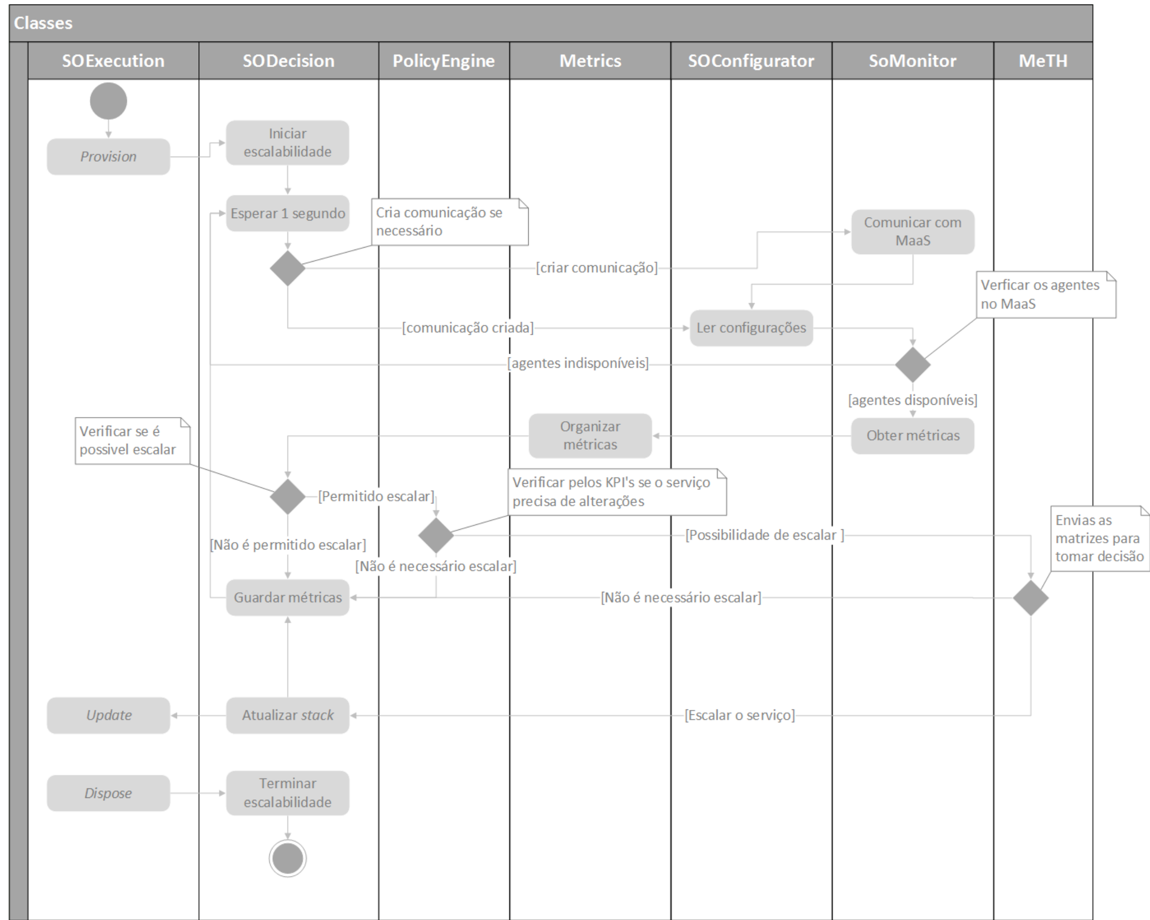


Figura 30 - Diagrama de atividade do algoritmo de escalabilidade

5.2. Testes

Na avaliação do algoritmo desenvolvido foi necessário comparar com outros algoritmos de forma a ter uma perceção da sua eficácia quanto à escalabilidade do serviço. Os algoritmos que serviram de base comparativa com o desenvolvido, foram dois algoritmos com abordagens semelhantes, mas com métricas e perspetivas diferentes, conforme sintetizado na Tabela 14.

Algoritmo	Scale In	Scale Out
CPU ou Memória	≤ 20%	≥ 80%
Latência	≤ 50 ms	≥ 300 ms
2 Níveis	KPIs e MeTH	KPIs e MeTH

Tabela 14 - Comparação entre decisões dos algoritmos

O primeiro algoritmo utiliza *thresholds* das métricas CPU e memória, de forma a identificar a utilização que os recursos estão a ter e desta forma escalar, se necessário. Sendo assim, este algoritmo quando atinge valores inferiores a 20% de utilização de CPU ou memória aciona o *scale in*, no caso oposto é acionado quando existem valores superiores a 80%. Por sua vez, o segundo algoritmo utiliza *thresholds* da métrica de latência, para verificar se o serviço não está a demorar a responder aos pedidos, escalando se tal for necessário. Para isso o algoritmo quando tem valores inferiores a 50 ms de latência aciona o *scale in*, enquanto que o *scale out* é acionado aquando se registam valores de 300 ms.

Devido às limitações e problemas que o *testbed* causou, os *thresholds* acima referidos na Tabela 14 tiveram de ser alterados de forma a tentar combater esses mesmos problemas. Assim foram feitos alguns testes de forma a perceber as restrições do *testbed*, permitindo corrigir os mesmos *thresholds* ao estado atual do *testbed*.

Algoritmo	Scale In	Scale Out
CPU ou Memória	≤ 10%	≥ 70%
Latência	≤ 30 ms	≥ 90 ms
2 Níveis	KPIs e MeTH	KPIs e MeTH

Tabela 15 - *Thresholds* atualizados

Para avaliar o algoritmo foram realizados testes para analisar a sua *performance* perante uma certa quantidade de tráfego. O dnsperf[54] foi a ferramenta utilizada para introduzir tráfego de DNS no serviço DNSaaS. Nos testes o número de clientes varia entre 1, 3 e 5 em que cada cliente envia 50.000 *queries* de DNS. A configuração dos testes evidente na Tabela 16, apresenta a forma como os três algoritmos foram testados, em que cada um foi sujeito a dois tipos de testes. O primeiro teste inicia com 1 cliente seguindo-se após a sua conclusão o mesmo teste com 3 clientes e sucessivamente com 5 clientes procedendo-se depois o percurso inverso, com 3 e 1 cliente – formando uma espécie de parábola. O segundo teste é o inverso do primeiro, começando com 5 clientes, passa depois para os 3 e de seguida 1 cliente e após isso o percurso inverso (3 e 5 clientes). Estes dois tipos de testes permitem verificar a capacidade de adaptação dos algoritmos para mudanças de carga.

ID do Teste	Nº de clientes	*Queries* enviadas	Objetivo
Teste 1	1,3,5,3,1	50.000, 150.000, 250.000, 150.000, 50.000	Verificar a capacidade de adaptação do algoritmo quando começa com pouca carga vai aumentando e depois volta a diminuir voltando ao ponto inicial
Teste 2	5,3,1,3,5	250.000, 150.000, 50.000, 150.000, 250.000	Verificar a capacidade de adaptação do algoritmo quando começa com muita carga vai diminuindo e quando se verifica a diminuição volta ao ponto inicial

Tabela 16 - Configuração dos testes do algoritmo

Nos testes o serviço tinha um *forwarder* com 4 vCPUs e 8 GB de memória (*flavor* m1.large), com o PowerDNS nas configurações padrão do *recursor*. OS DNS *Servers* têm 2 vCPUs e 4 GB de memória (*flavor* m1.medium), utilizando o PowerDNS com as configurações *standard* de *authoritative*, onde o número de DNS *Servers* é alterado pelos algoritmos. A seguinte imagem demonstra a estrutura do serviço durante os testes.

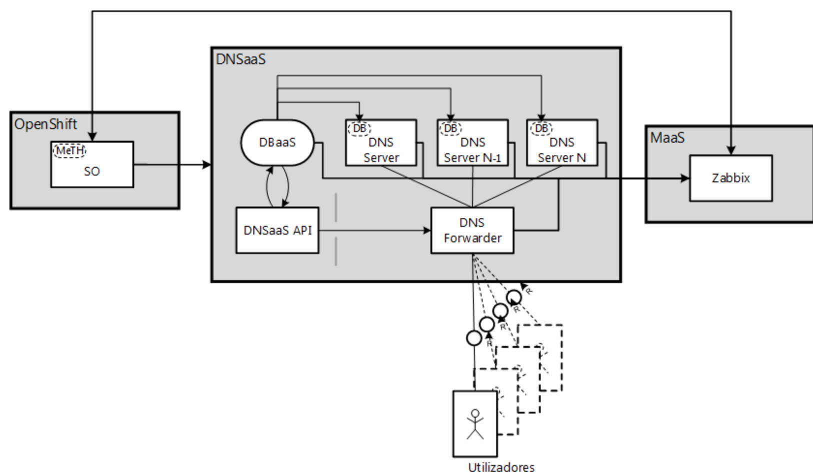


Figura 31 - Estrutura do serviço durante os testes.

Para verificar se os algoritmos têm a *performance* esperada foi elaborado um sistema de contagens de violações de SLO[57]. Este sistema permite avaliar se o serviço tem as características necessárias para responder aos pedidos atuais, não alocando para isso recursos desnecessários. Para calcular as violações de SLO foram utilizadas quatro métricas percentagem de answers slow, latência, percentagem de CPU e memória livre, sendo contadas as violações tendo em conta certas condições por cada métrica. Na métrica percentagem *answer slow* é acumulado 1 ponto a cada 1% que ultrapasse os 20%, na latência é acumulado 1 ponto a cada 1 ms que ultrapasse os 300 ms e

por fim no CPU e memória livre é acumulado 1 ponto a cada 5 % livre, a acumulação destas condições irá ditar as violações de SLO.

A título de exemplo, supondo que num determinado momento é registado 27% de *answer slow*, 384 ms de latência, 77% de utilização de CPU e 20% de utilização de memória. O resultado de violações de SLO seria: 7 (*answer slow*)+ 84 (latência) + 4 (CPU) + 16 (memória).

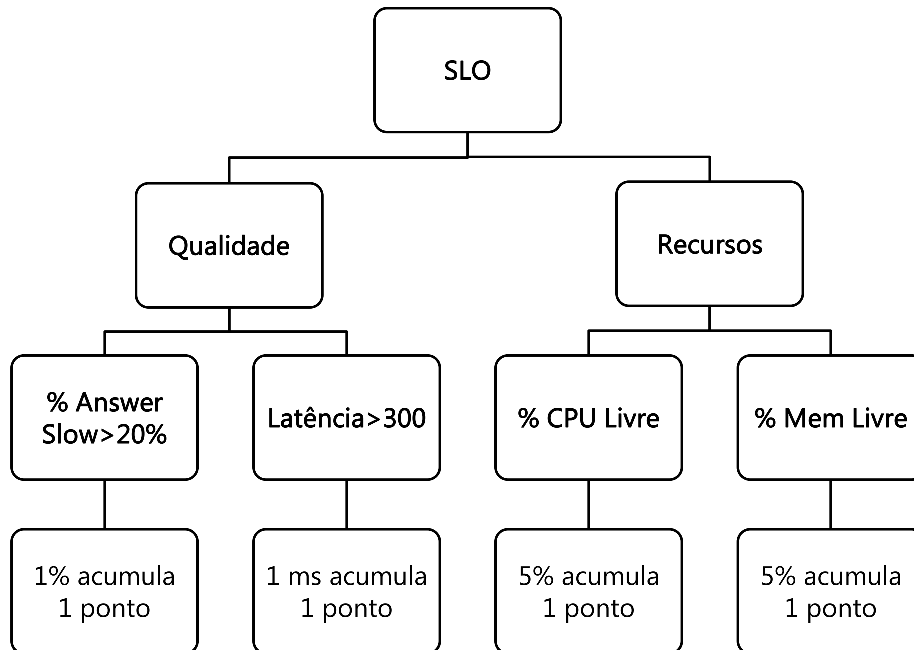


Figura 32 - Ramificação de SLO

5.3. Análise

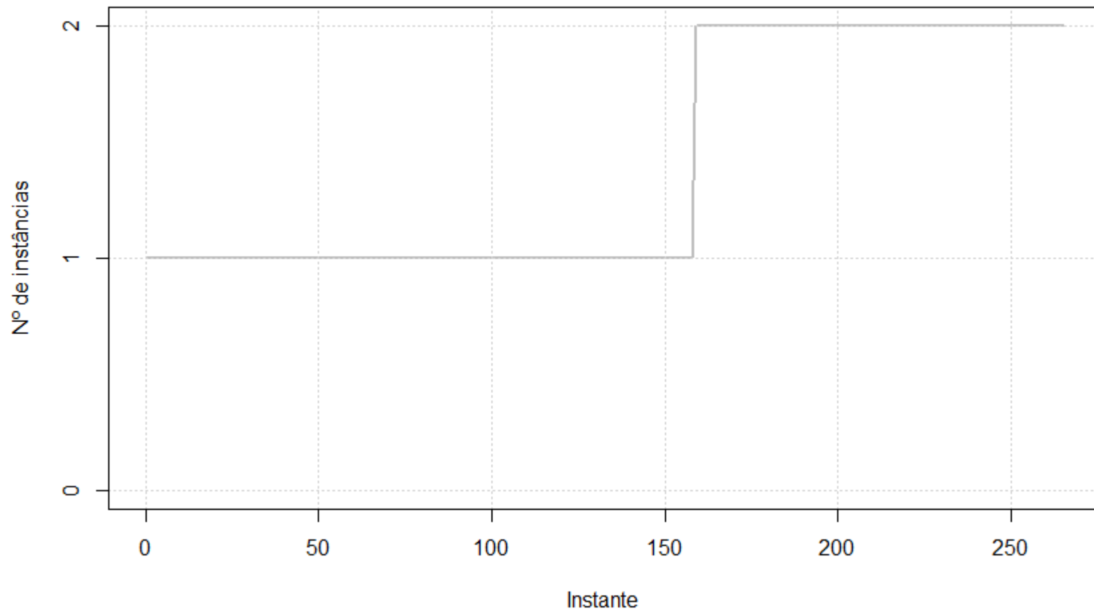
Na análise aos algoritmos, escolheram-se métricas chave sobre o estado do serviço e dos recursos. Assim a escolha recaiu sobre o número de instâncias, latência, percentagem de *answers slow*, utilização de CPU e violações de SLO, sendo esta última métrica conclusiva no rácio desempenho/custo dos algoritmos.

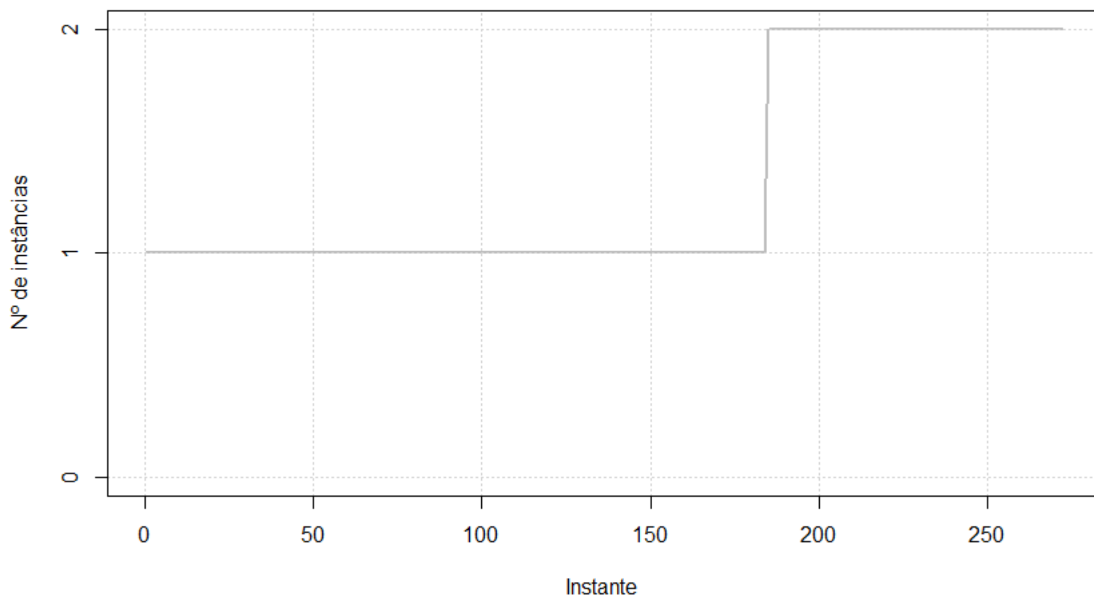
As métricas dos três algoritmos foram analisados ao longo dos instantes que o SO registou, isto é, o instante 50 representado nos gráficos não corresponde a 50 segundos, corresponde ao 50º instante em que o SO registou o valor Desta forma entre instantes o tempo é variável e não existe uma relação direta com a unidade de medida de tempo.

Teste 1

O teste 1 pretende verificar a capacidade de adaptação do algoritmo. Quando é enviada pouca carga, vai aumentando e depois volta a diminuir voltando ao ponto inicial, formando assim uma espécie de parábola relativamente à carga submetida. Assim a primeira métrica analisada – número de instâncias –, destacada na Figura 33, permite perceber que apenas o algoritmo desenvolvido

não chegou a escalar, dando desde logo a entender que existem diferenças na *performance* dos algoritmos. As métricas seguintes irão permitir perceber se a não adição de uma nova instância foi benéfica para o algoritmo.





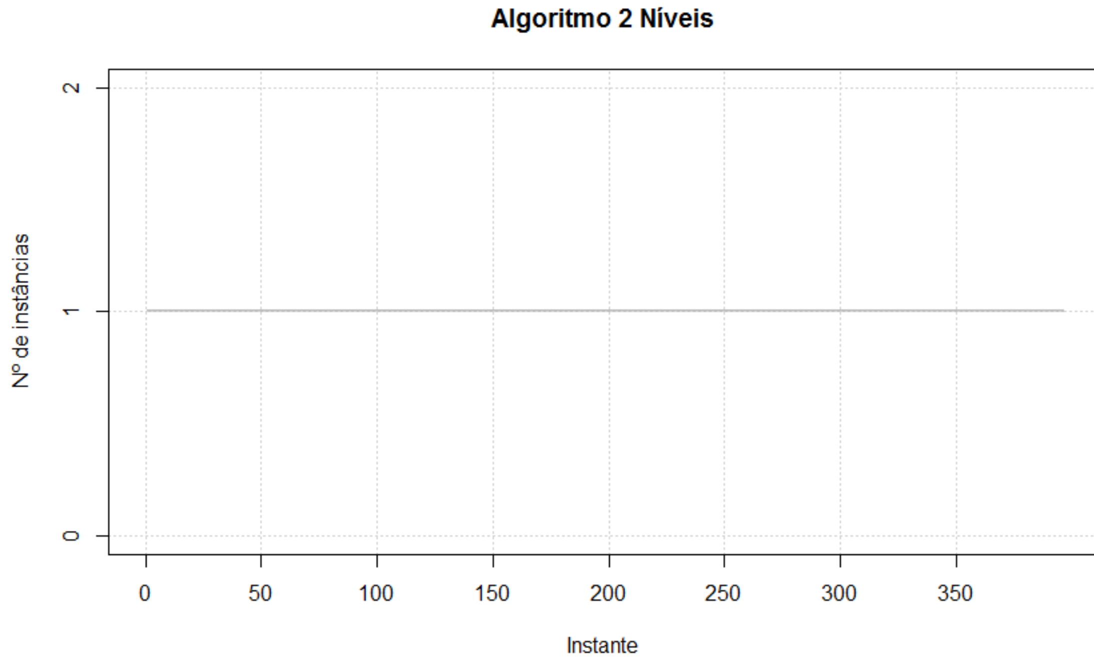
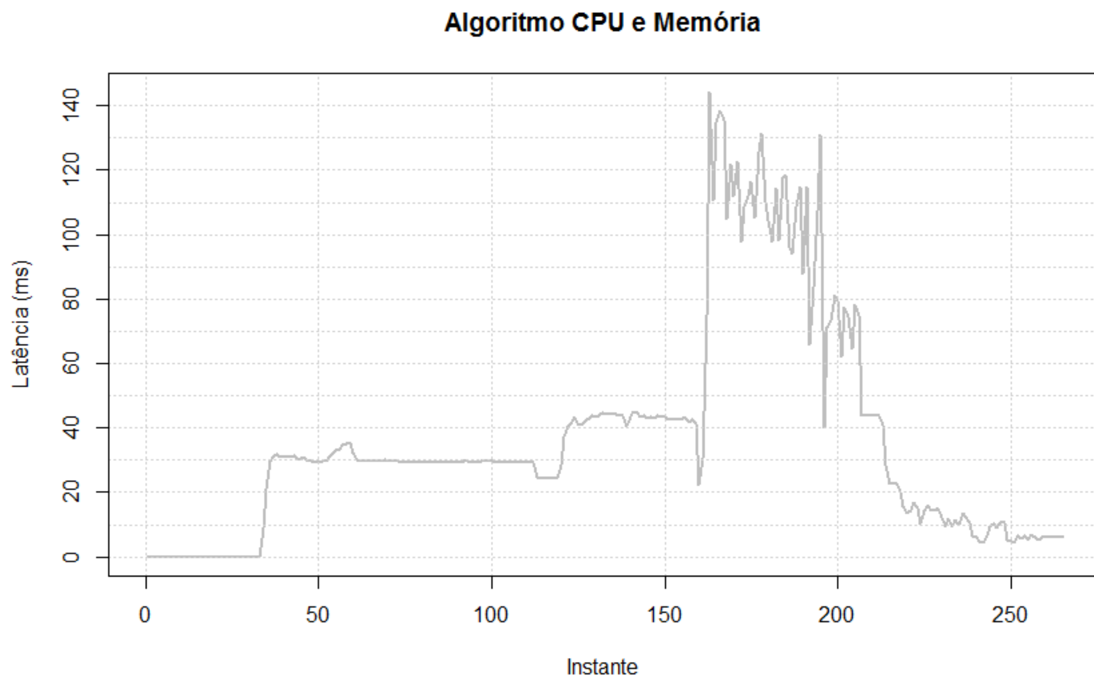


Figura 33 - Comparação dos algoritmos durante o teste 1, relativamente ao número de PDNS

A seguinte métrica analisada é a latência, que permite perceber o tempo necessário para dar respostas às *queries* pedidas. A Figura 34 mostra a latência registada pelos diferentes algoritmos. Da análise a esta métrica pode perceber-se que os três algoritmos chegam à marca dos 140ms de latência e têm resultados muito semelhantes. Assim pode-se verificar que a decisão de não adicionar uma máquina não influenciou a métrica da latência de uma forma positiva.



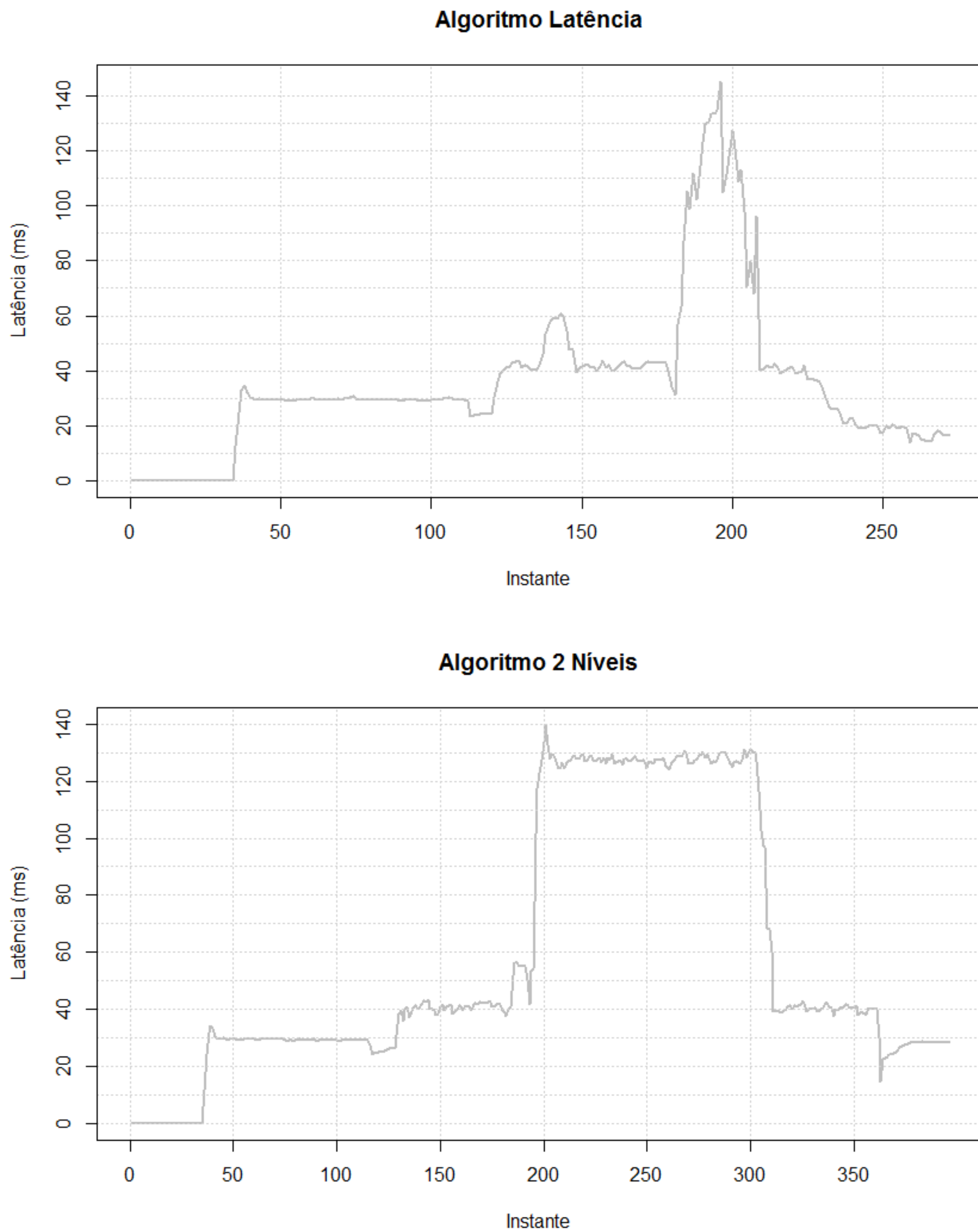
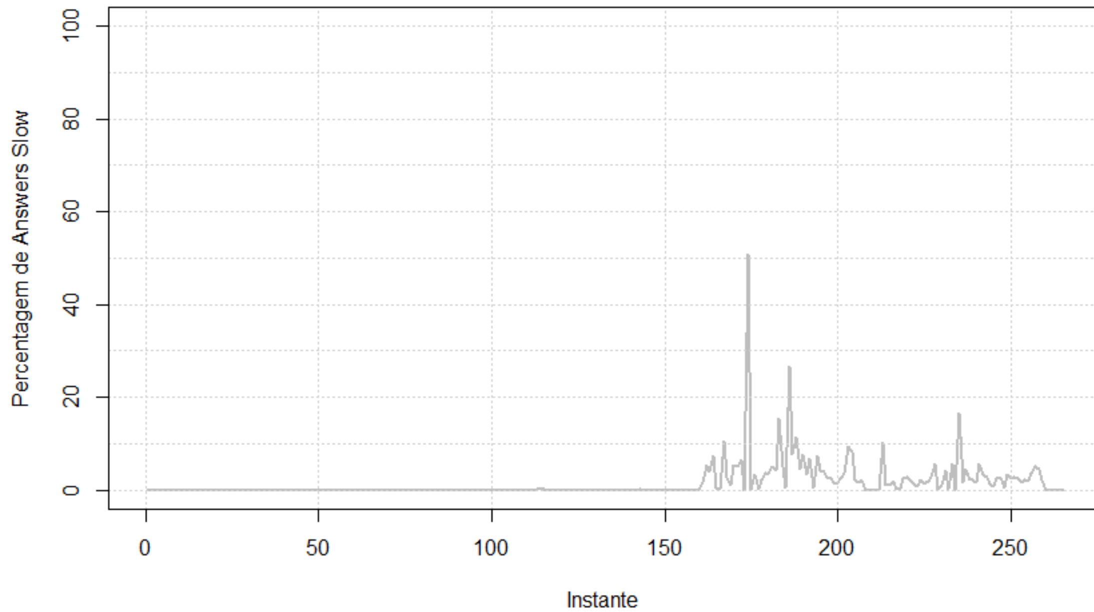
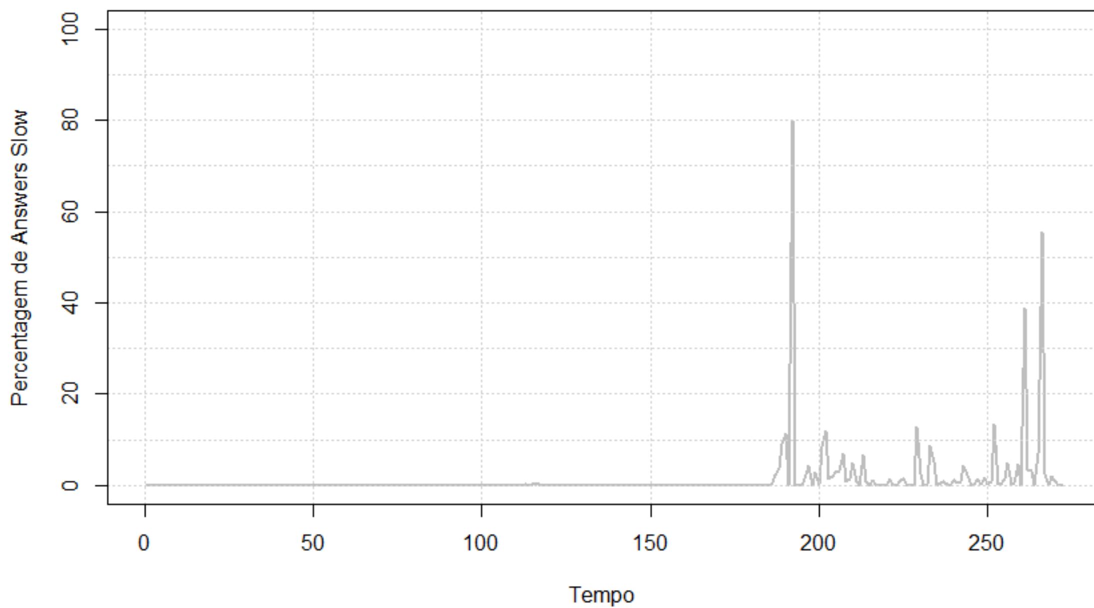


Figura 34 - Comparação dos algoritmos durante o teste 1, relativamente à latência

A métrica ilustrada na Figura 35 é a percentagem de *answers slow*. Esta métrica mostra as *queries* que demoram mais de um segundo, dando a perceber se os tempos de resposta são demasiado elevados. Assim, o valor mais elevado do algoritmo de CPU e Memória é de cerca de 50%, tendo também valores a rondar os 20%. O algoritmo de Latência, que regista os piores resultados nesta métrica, tem como valor máximo 80%, tendo também valores a rondar os 40%. Por fim o algoritmo de 2 níveis não possui valores a registar nesta métrica, evidenciando desde já sinais muito positivos.

Os valores mais elevados nesta métrica ocorrem durante a escalabilidade do serviço, podendo portanto concluir-se que uma escalabilidade no momento errado traz consequências negativas na sua *performance*.





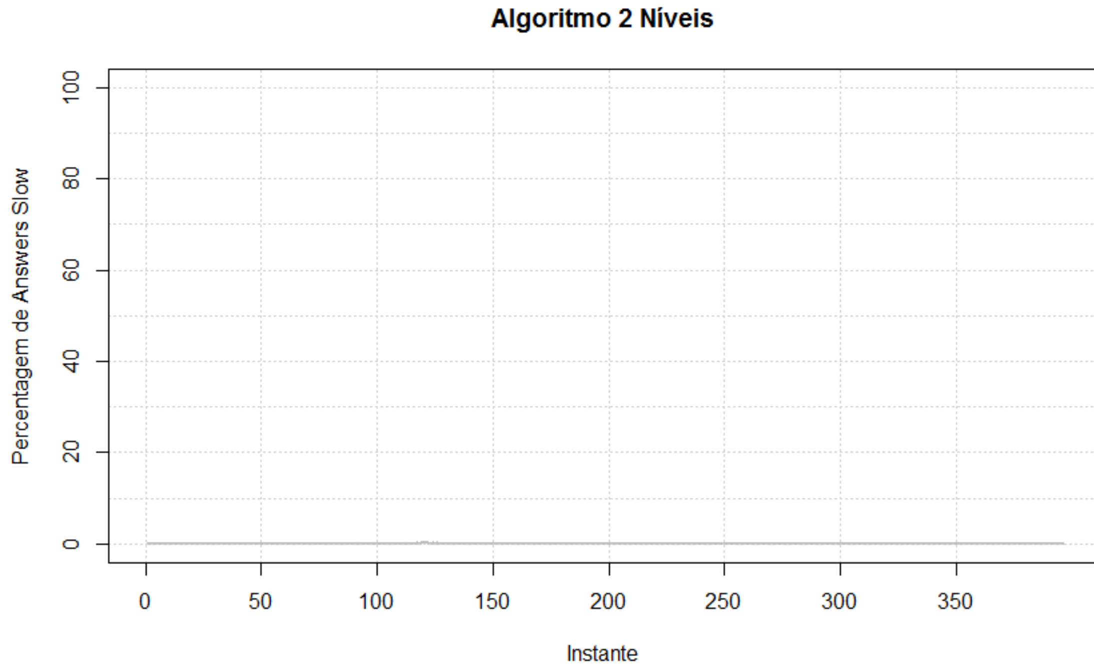
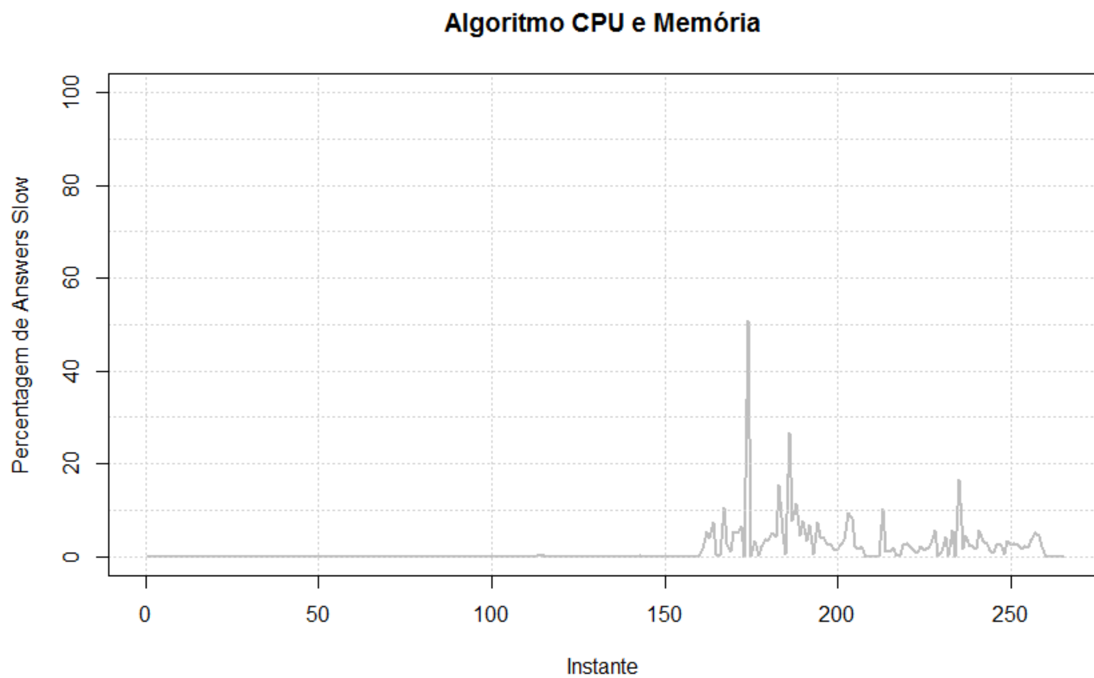


Figura 35 - Comparação dos algoritmos durante o teste 1, à percentagem de *answers slow*

Relativamente à métrica de utilização de CPU, representada na Figura 36, é percetível que dos três algoritmos, o desenvolvido tem a maior utilização de CPU, mas para além disso, depreende-se pelas métricas de latência e percentagem de *answers slow*, que esta maior utilização não afetou a performance do serviço.



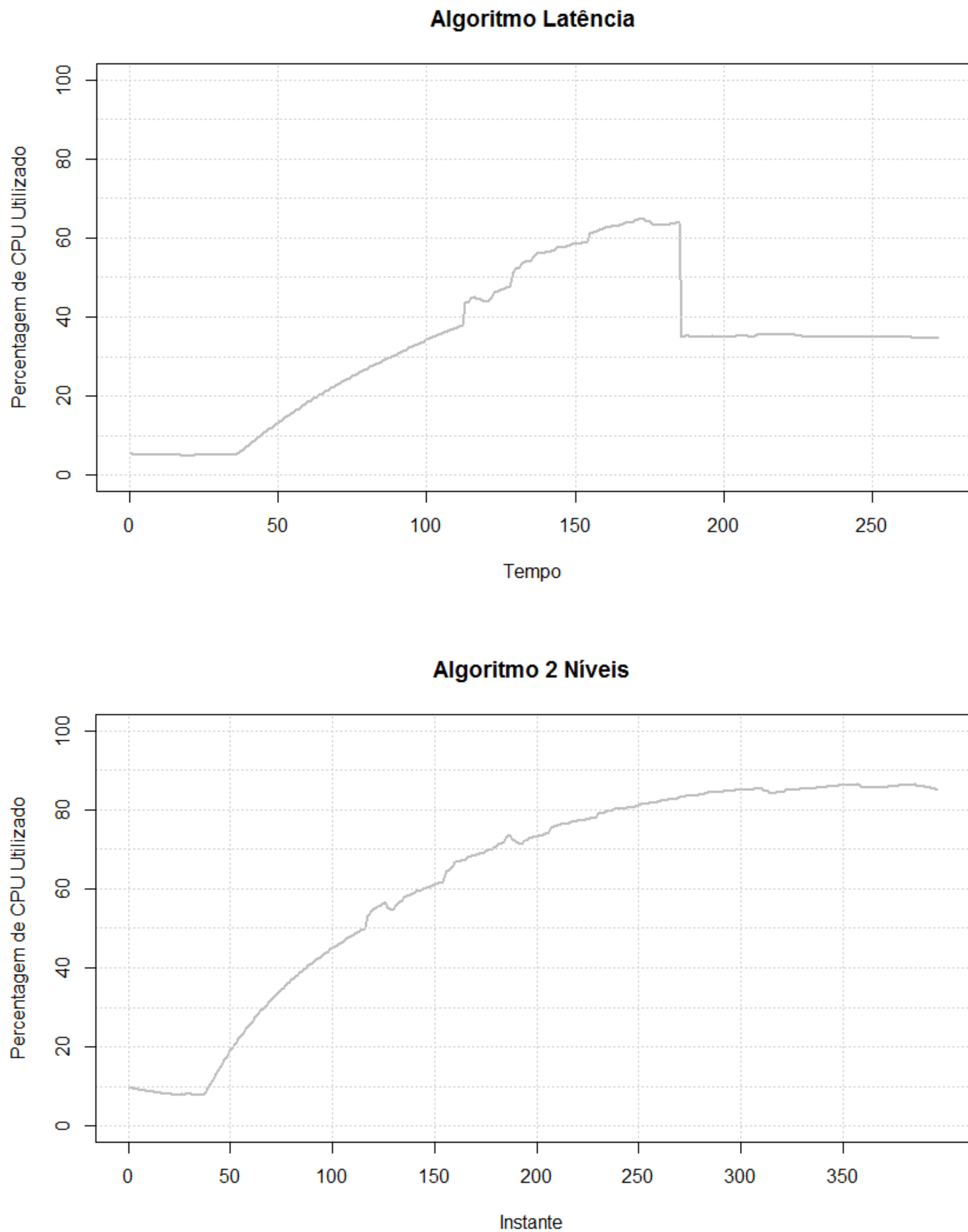
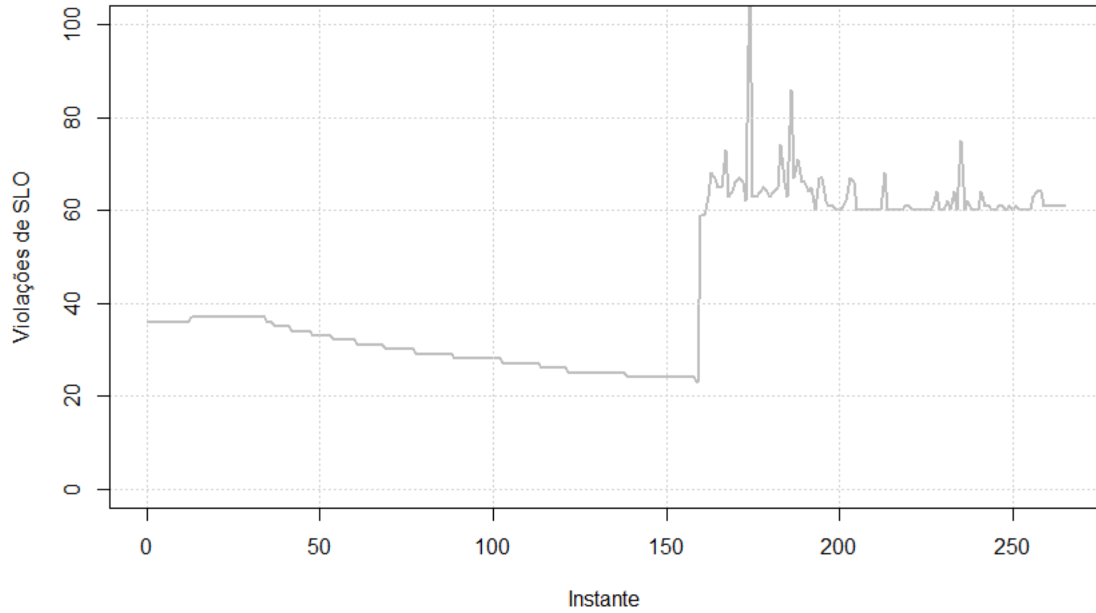
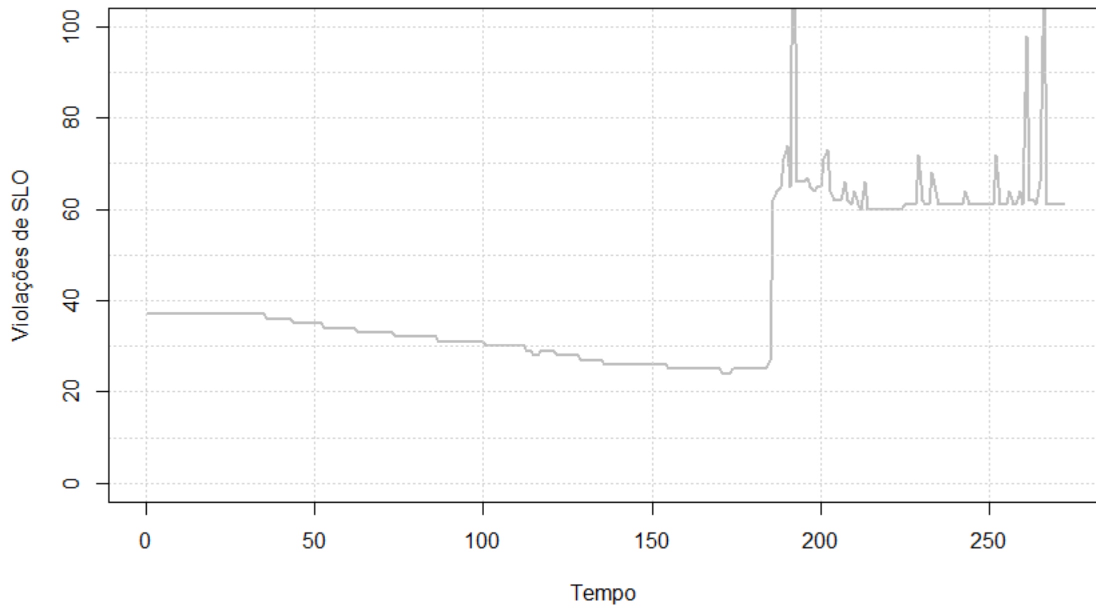


Figura 36 - Comparação dos algoritmos durante o teste 1, relativamente ao CPU utilizado

As contagens de violações de SLO, ilustrada na Figura 37, permite perceber o desempenho dos algoritmos, podendo verificar se o serviço tem qualidade – respondendo corretamente às *queries* no menor tempo possível – não alocando recursos desnecessários. Assim, o algoritmo de 2 níveis, devido a não ter alocado instâncias desnecessárias e não ter influenciado negativamente na *performance* do serviço tem poucas violações de SLO. Por sua vez os algoritmos de Latência, CPU

e Memória têm piores resultados uma vez que escalaram quando o serviço não necessitava expondo o serviço e colocando as respostas de *queries* em segundo plano.





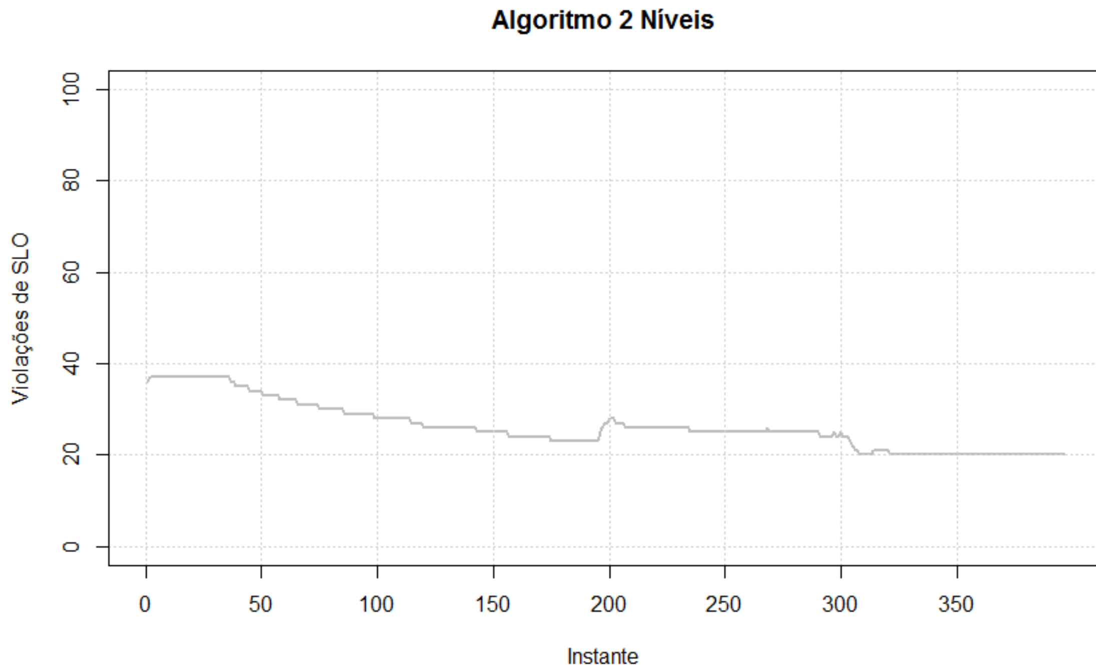
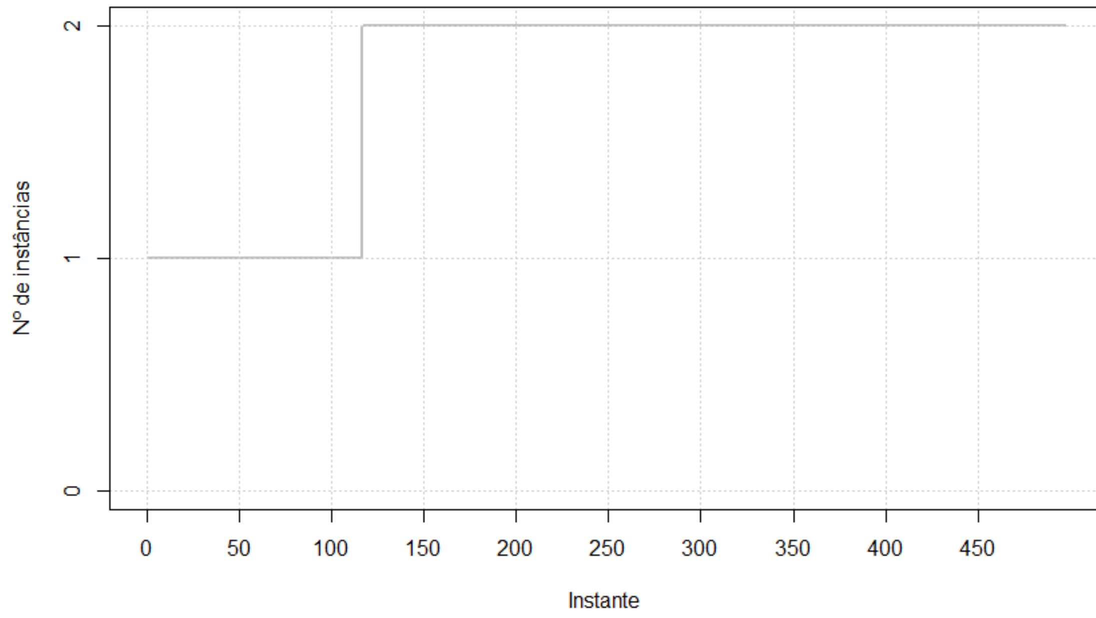


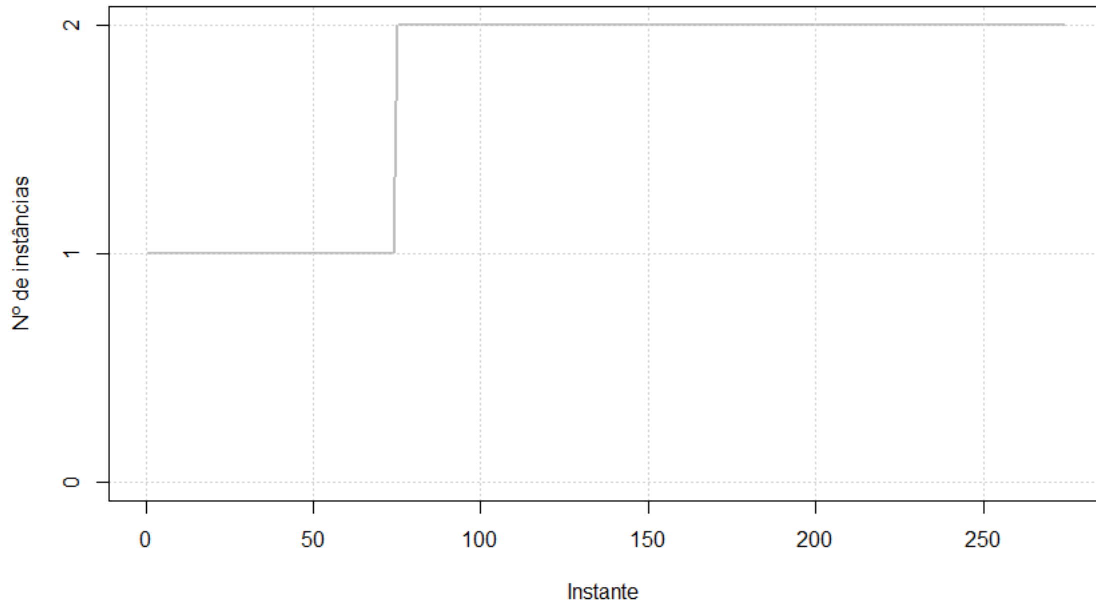
Figura 37 - Comparação dos algoritmos durante o teste 1, relativamente às violações de SLO

Neste primeiro teste pode verificar-se que o algoritmo desenvolvido tem melhores resultados do que os concorrentes e a decisão de não escalar foi a mais acertada, uma vez que não influenciou negativamente o desempenho do serviço de DNS.

Teste 2

O teste 2, funciona de forma inversa ao teste anterior, i.e. pretende verificar a capacidade de adaptação do algoritmo quando é enviada muita carga, vai diminuindo e quando se verifica a diminuição volta ao ponto inicial. Relativamente ao número de instâncias – destacada na Figura 38 – uma vez mais o algoritmo desenvolvido não chegou a escalar e as próximas métricas analisadas ditarão se esta decisão foi a mais correta e não influenciou o algoritmo.


Algoritmo CPU e Memória
Nº de instâncias
0
1
2
0
50
100
150
200
250
300
350
400
450
Instante


Algoritmo Latência
Nº de instâncias
0
1
2
0
50
100
150
200
250
Instante

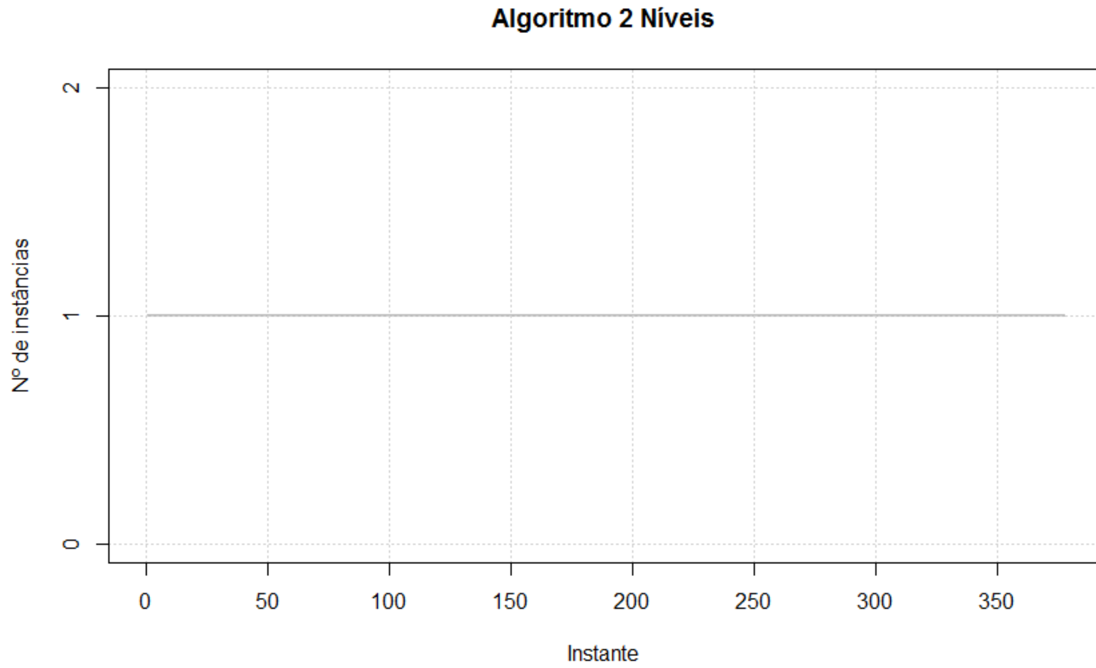
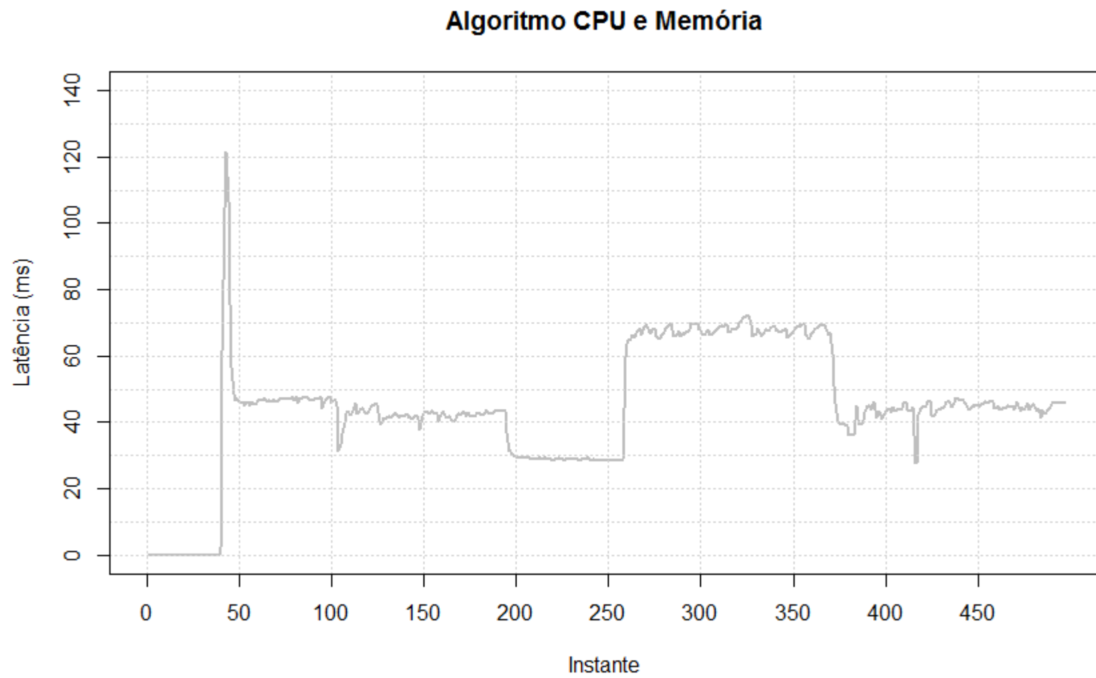


Figura 38 - Comparação dos algoritmos durante o teste 2, relativamente ao número de PDNS

Relativamente à métrica Latência, métrica representada na Figura 39, os resultados são muito semelhantes entre os três algoritmos com ligeira vantagem para o algoritmo de CPU e Memória. Veremos se esta mínima vantagem traz benefícios na conclusão final dos algoritmos.



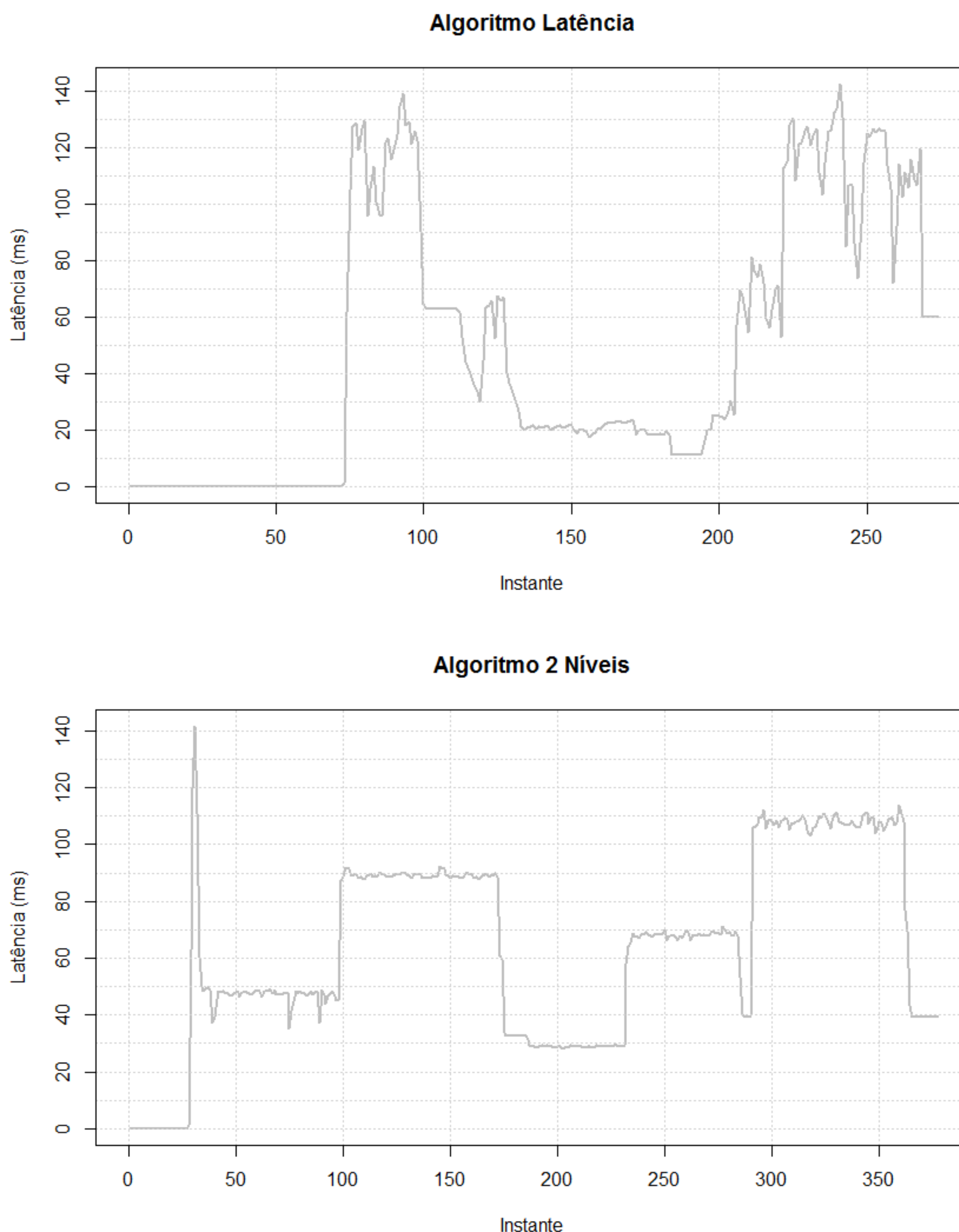
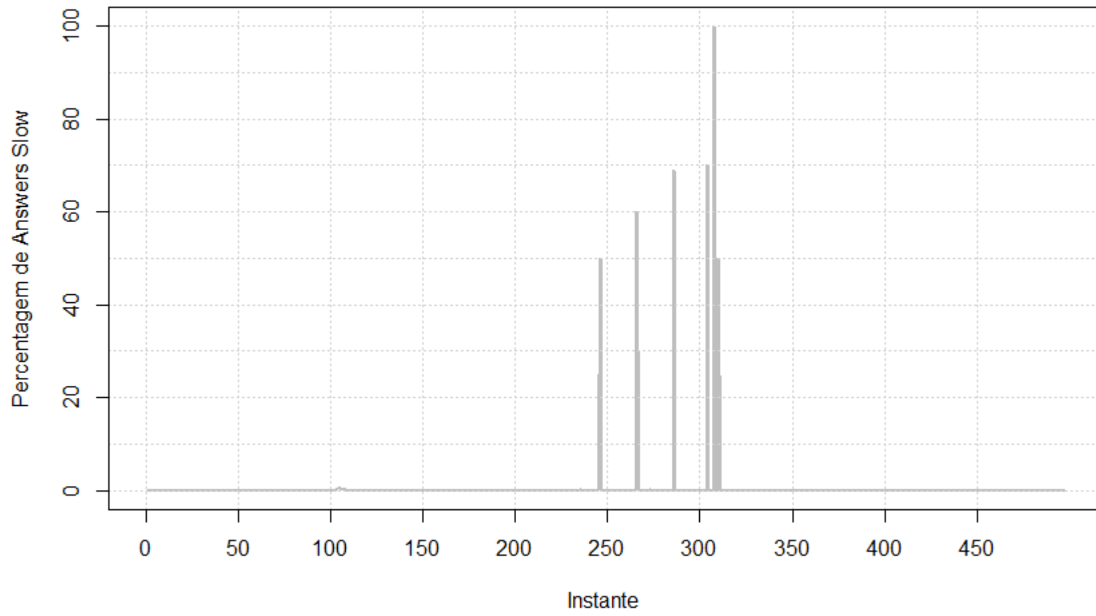
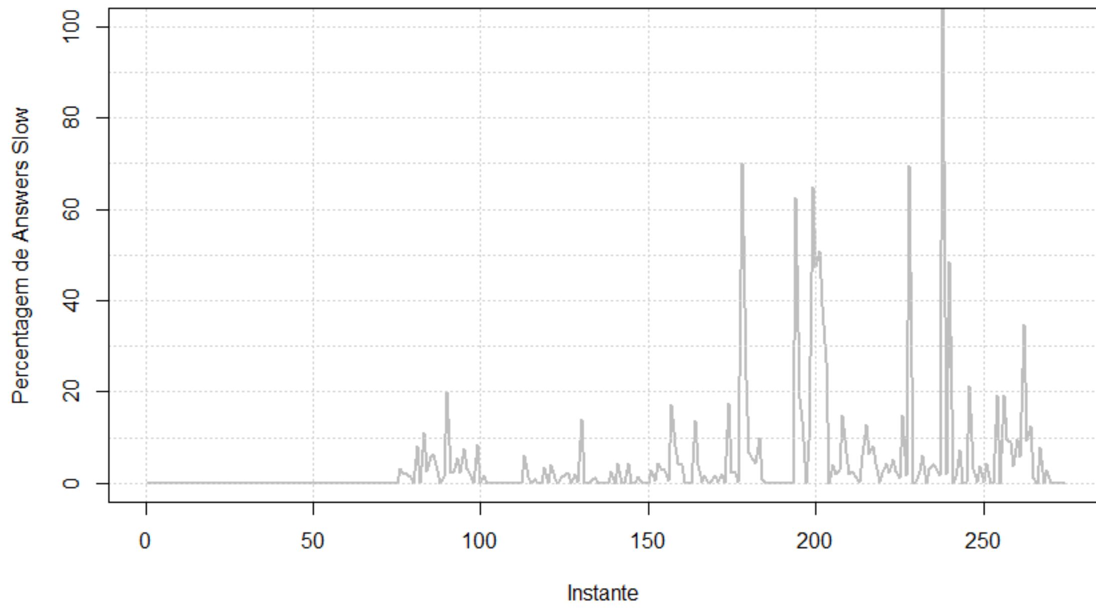


Figura 39 - Comparação dos algoritmos durante o teste 2, relativamente à latência

Na Figura 40 encontra-se a comparação entre os três algoritmos relativamente à métrica de percentagem de *answers slow*. Os piores resultados são do algoritmo de latência e os melhores são do algoritmo de 2 níveis. O algoritmo de CPU e Memória, tem alguns picos ocasionais, sendo os valores máximo de 100%. O algoritmo de Latência tem como máximo um pico de 100% e ainda possui alguns valores a rondar os 60%. Por ultimo, o algoritmo de 2 níveis tem apenas um pico e esse não ultrapassa os 4%, tendo portanto bons indicativos sobre o seu desempenho.


Algoritmo CPU e Memória
Percentagem de Answers Slow
0
20
40
60
80
100
0
50
100
150
200
250
300
350
400
450
Instante


Algoritmo Latência
Percentagem de Answers Slow
0
20
40
60
80
100
0
50
100
150
200
250
Instante

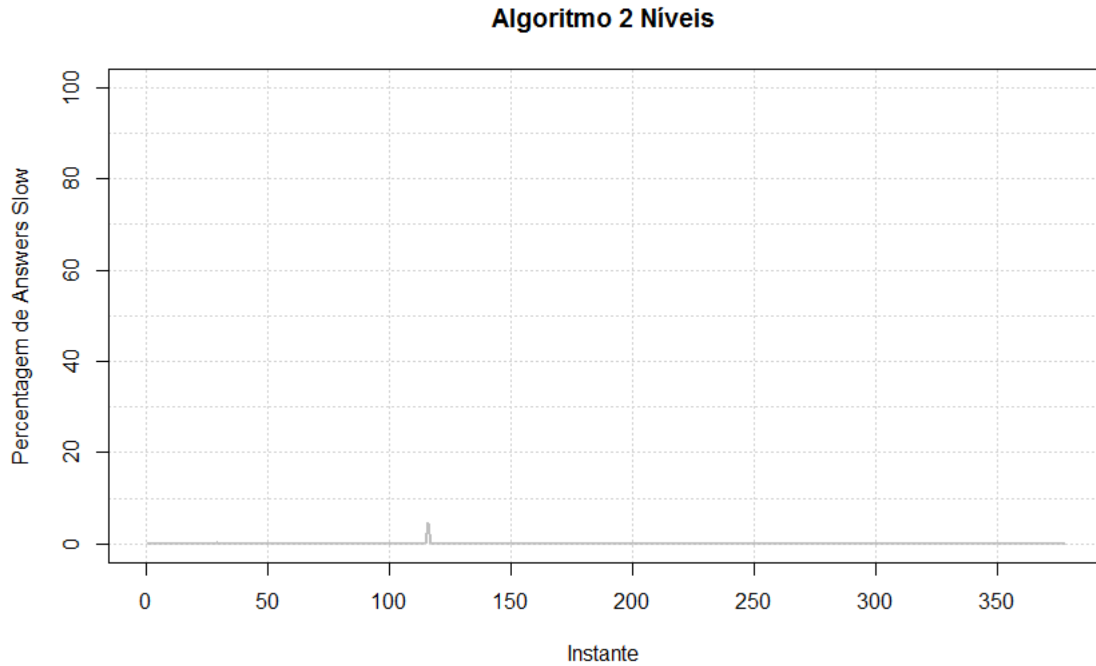
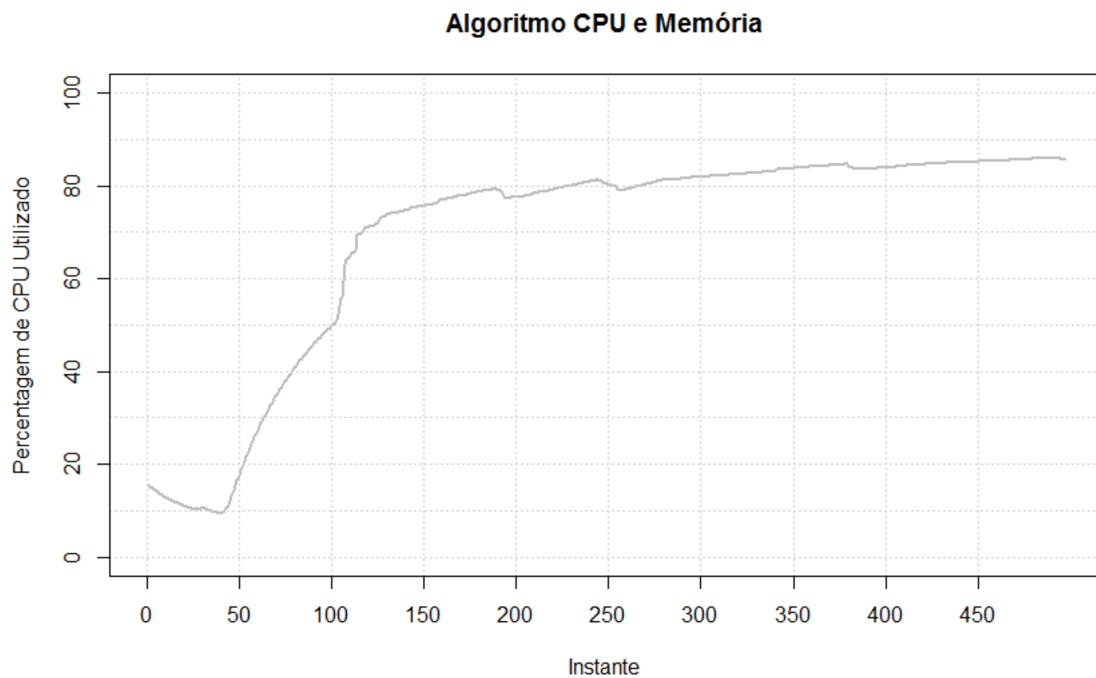


Figura 40 - Comparação dos algoritmos durante o teste 2, à percentagem de *answers slow*

Na métrica de utilização de CPU, ilustrada na Figura 41, é visível que o algoritmo de Latência utiliza menos CPU do que os restantes concorrentes. Analisando as métricas de latência e *answers slow* percebe-se que o algoritmo não utilizou completamente os recursos à sua disposição.



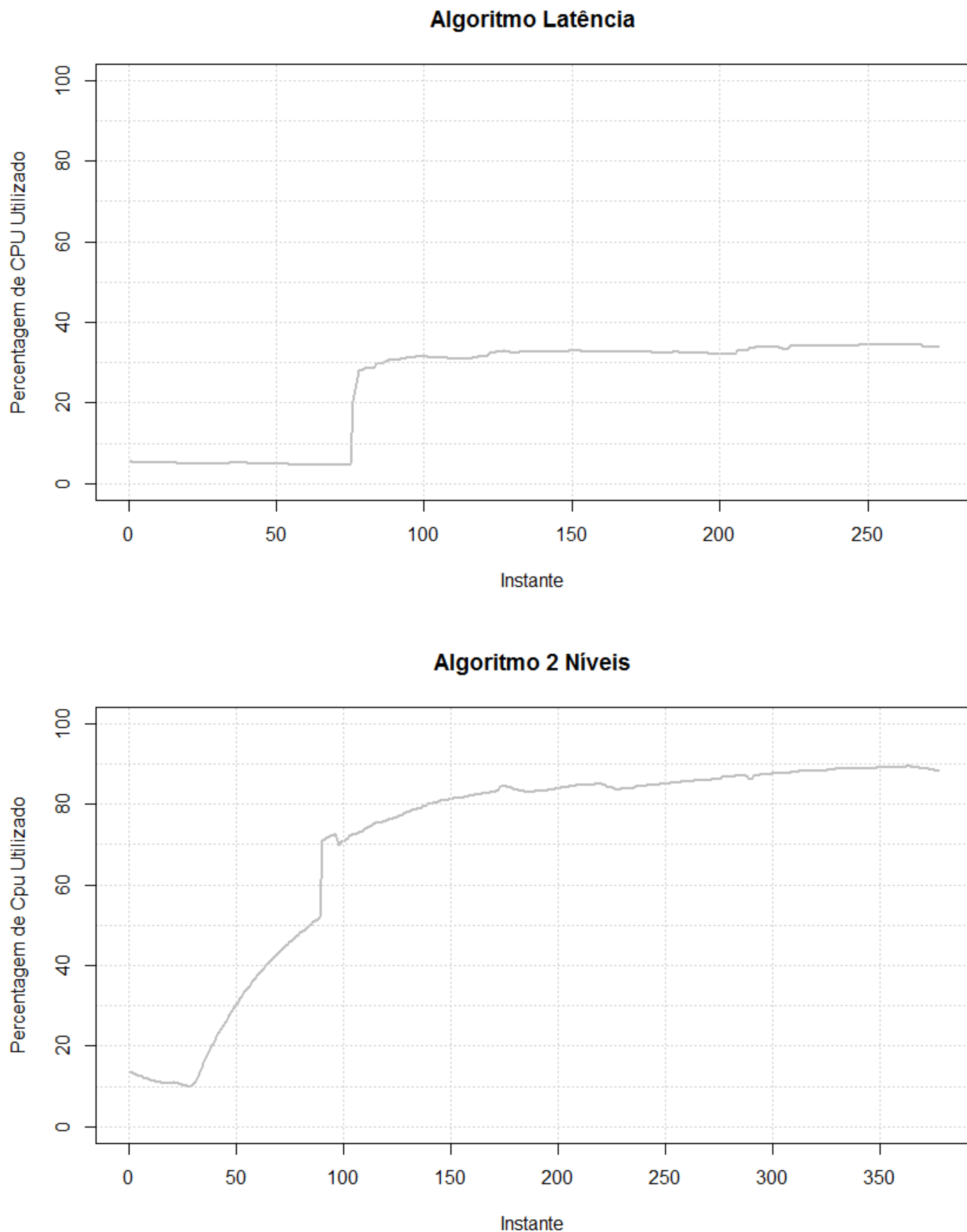
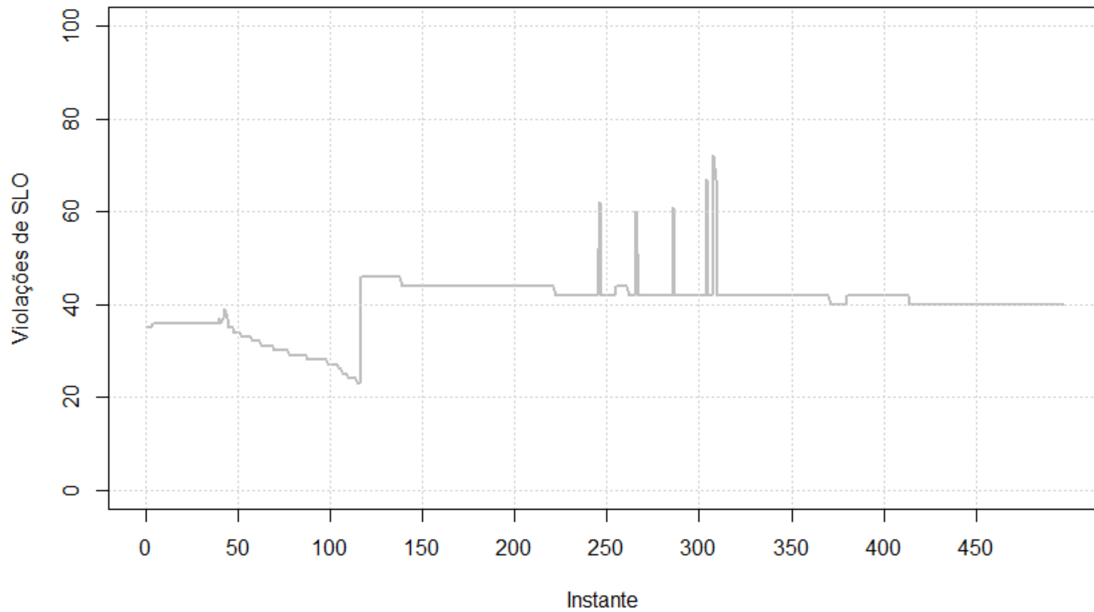
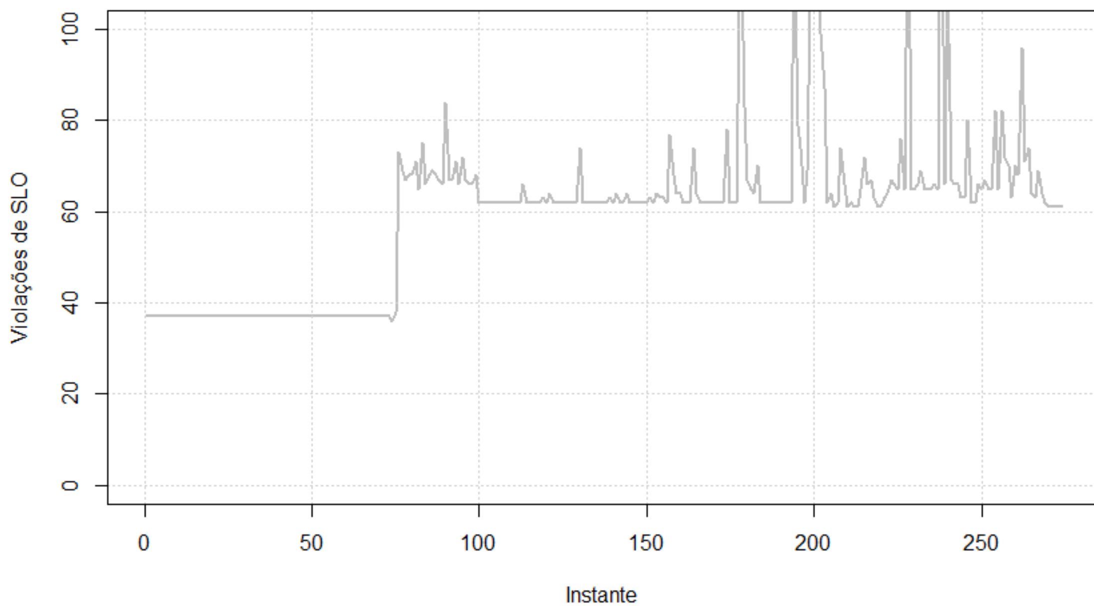


Figura 41 - Comparação dos algoritmos durante o teste 2, relativamente ao CPU utilizado

Em jeito de conclusão a métrica de violações de SLO, ilustrada na Figura 42, permite perceber que o algoritmo de Latência tem os piores resultados, tendo valores a rondar as 60 violações, fora os picos excecionais que atingem as 100 violações de SLO. Em segundo lugar vem o algoritmo de CPU e Memória em que a escalabilidade do serviço veio penalizar, com os valores a rondar as 40 violações existindo picos ocasionais na ordem das 60 violações de SLO. E em primeiro lugar regista-se algoritmo de 2 níveis, onde os valores rondam as 20 violações tendo resultados superiores aos

diretos concorrentes. Esta métrica permite constatar que os algoritmos de CPU e Memória, Latência ao escalarem não tomaram a melhor decisão para o serviço colocando em causa a sua *performance* e qualidade.





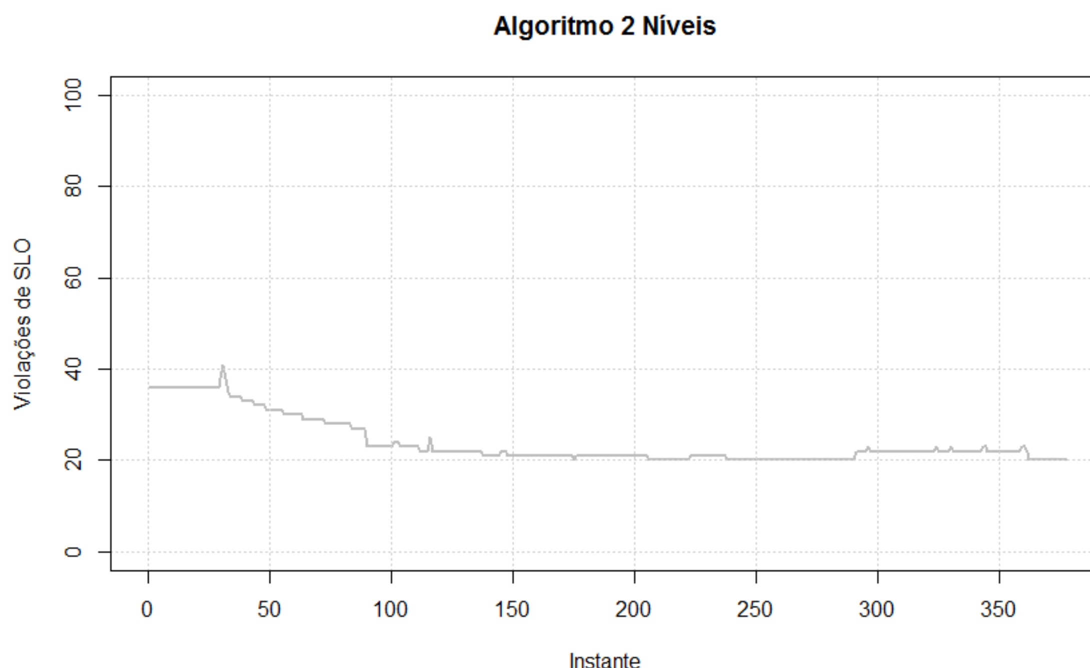


Figura 42 -Comparação dos algoritmos durante o teste 2, relativamente às violações de SLO

O teste 2 mostra mais um bom desempenho do algoritmo desenvolvido. Uma vez mais adecisão tomada em não escalar o serviço foi a mais acertada, pois o serviço conseguiu suportar o volume de tráfego com penas uma instância.

Conclusões

O algoritmo de escalabilidade permite adaptar o serviço que o irá ajudar a ter a estrutura necessária para responder aos pedidos. Nesta tarefa o algoritmo deve ser o mais transparente possível para o serviço, tendo o cuidado de ter um impacto positivo. A partir dos testes realizados foi possível tirar ilações sobre os algoritmos. Conclui-se que a decisão de escalar, tomada num momento errado ou numa altura em que o serviço não necessita de facto de ser escalado, pode comprometer o bom desempenho do serviço. Através das métricas de violações de SLO consegue-se concluir que o algoritmo de 2 níveis garante qualidade ao serviço sem que para isso aloque recursos desnecessários, dando claramente sinais muito positivos. Mas para concluir a análise sobre o desempenho dos algoritmos foram também verificadas as métricas de latência e *queries* por segundo relativas ao cliente, para perceber a influência que os algoritmos têm sobre o serviço na perspetiva de quem efetua pedidos. Assim nas Figura 43 e Figura 44 estão representadas a latência e *queries* por segundo, respetivamente, de cada algoritmo, onde o v1 é o algoritmo de CPU e Memória, o v2 o algoritmo de Latência, o v3 o algoritmo de 2 níveis, o t1 é o teste 1 e o t2 é o teste

2. Assim no id do teste v3t1 representa o desempenho do algoritmo de 2 níveis durante o teste 1. Relativamente à métrica de latência no teste 1 o melhor resultado verifica-se com o algoritmo desenvolvido onde a média é de aproximadamente de 50ms, em segunda posição o algoritmo de latência com média de aproximadamente de 60ms – contudo tem valores a atingir os 160ms – e o pior resultado é o de CPU e Latência com uma média de 120ms, praticamente o dobro dos testes anteriores e ainda tem valores a alcançar os 185ms. No teste 2, mais uma vez o algoritmo desenvolvido a ter melhores resultados, com uma latência média a rondar os 70 ms e como máximo tem 110 ms, em segundo lugar encontra-se o algoritmo de CPU e Memória com uma média de 90ms e com valores máximos a atingir os 140ms; em último lugar vem o algoritmo de Latência com uma média de aproximadamente 120ms e os valores a atingirem 164ms de máximo.

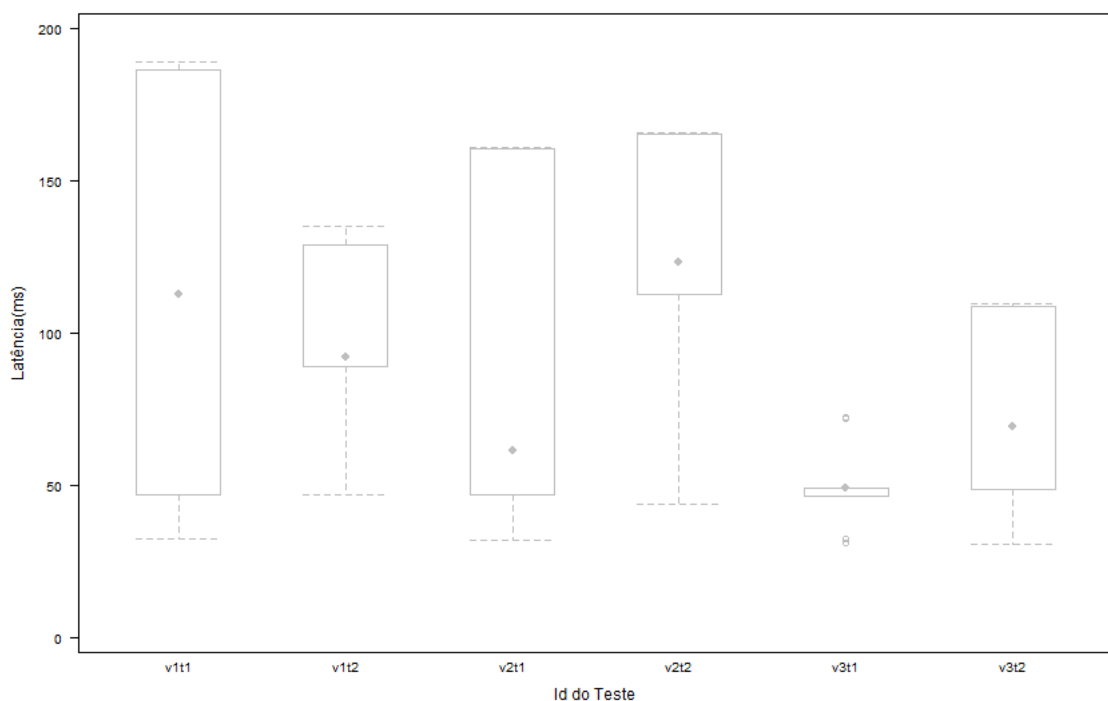


Figura 43 - Latência nos clientes durantes testes de *performance* do serviço

Na métrica de *queries* por segundo, no teste 1 o melhor resultado pertence ao algoritmo de 2 níveis onde consegue aproximadamente 2000 qps, em segundo lugar o algoritmo de Latência com média de 1600 qps, com valores mínimos a atingir 500 qps e o pior resultado verifica-se no algoritmo de CPU e Latência onde a média ronda os 800qps. Relativamente ao teste 2 o melhor resultado é do algoritmo desenvolvido com uma média a rondar os 1500 qps, tendo valores mínimos a alcançar os 1000qps, o segundo algoritmo é o CPU e Memória com uma média de 850 qps e em último lugar o algoritmo de Latência com valores médios a rondar os 700 qps.

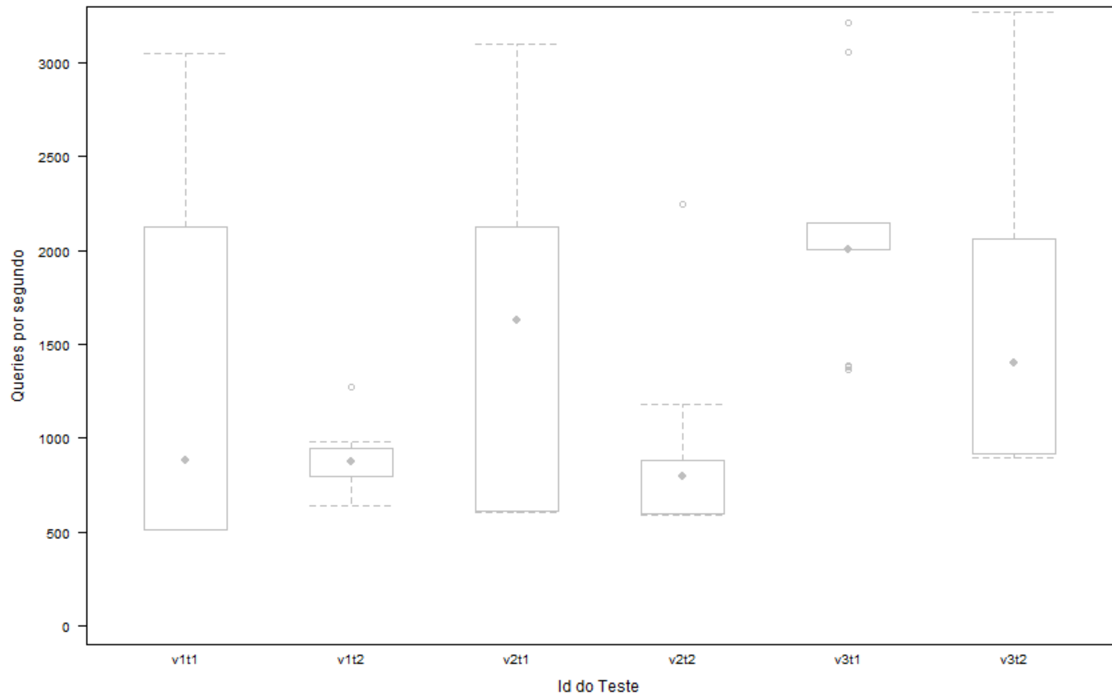


Figura 44 – *Queries* por segundo nos clientes durantes testes de performance do serviço

Perante estes resultados é possível concluir que o algoritmo de 2 níveis toma melhores decisões relativamente aos concorrentes apresentados, não tendo portanto impacto prejudicial para os serviços nas suas decisões.

Capítulo 6 : Conclusão

Este capítulo marca o fim do relatório e o fim de um ano de estágio, chegando à altura de ser feito um balanço e uma súmula do trabalho elaborado. O estágio tinha como objetivo a implementação de um algoritmo de escalabilidade para o serviço DNS em contexto *cloud*. Para atingir o objetivo proposto foi necessário proceder às seguintes etapas – familiarização com o tema, especificação da arquitetura, desenvolvimento e análise do algoritmo.

O estágio iniciou-se com a familiarização ao tema das tecnologias envolvidas no projeto., abordando essencialmente as temáticas de *cloud computing* e DNS, permitindo perceber que a migração do DNS para a *cloud* traz vantagens, como por exemplo, tornar o serviço elástico e facilitar a gestão do mesmo. Com a elaboração de um método de obtenção do serviço DNS, tornou-se possível, de certa forma, uma melhor integração no projeto MCN, esclarecendo como os diferentes componentes trabalham. Apreendido o funcionamento dos componentes do projeto, aprofundei o conhecimento no serviço de DNS, entendendo a funcionalidade e a forma como os componentes comunicam. Tomando estes conceitos como ponto de partida, foram efetuados testes às duas soluções de DNS, com a diferenciação do comportamento da base de dados, tendo uma solução com a base de dados centralizadas – acedida pelas diferentes servidores de DNS – e outra com a base de dados distribuída – existindo uma base de dados *master* que envia o conteúdo para as outras base de dados *slave* que se encontram nos servidores de DNS.

Analisando os resultados dos testes é possivel concluir que a arquitetura distribuída é a melhor opção para o serviço poder efetuar uma boa escalabilidade, obtendo esta quase sempre uma latência reduzida e um valor de *queries* por segundo bastante aceitável. A boa performance da arquitetura é visível quando se obtém uma latência de aproximadamente de 600 ms com apenas dois servidores de DNS e 25 utilizadores a efetuarem um total de 1.250.000 *queries*. A avaliação à arquitetura para além de ter permitido conclusões sobre qual a melhor arquitetura, elucidou igualmente sobre quais as métricas que irão auxiliar no escalamento do serviço, auxiliando na definição do algoritmo de escalabilidade. De referir que as avaliações efetuadas às arquiteturas

permitiram-me colaborar na elaboração de um artigo científico para o qual contribuí com alguma da descrição e análise à arquitetura.

Na especificação do algoritmo de escalabilidade existiu um estudo prévio de casos práticos deste tipo de algoritmos, para aferir técnicas e práticas utilizadas. Após o estudo foi possível perceber que existem 3 tipos de algoritmos de escalabilidade: reativos, preditivos e compostos (que utilizam as duas técnicas), os preditivos e compostos devido à dependência historial necessita de treino para recolher dados e mesmo após essa recolha podem acontecer muitas falhas porque o serviço de DNS é muito dinâmico e é difícil prever os recursos necessários para responder aos pedidos. A técnica reativa pode ter falhas e levar a que recursos sejam mal aproveitados muito devido aos falsos "picos". Tendo em conta estas desvantagens e as características do serviço foi especificado um algoritmo semirreativo com 2 níveis. No primeiro nível têm KPIs que são reativos e servem de indicadores de performance e, num segundo nível um algoritmo de decisão, que com base em métricas verifica se a melhor decisão é escalar.

Na análise o algoritmo desenvolvido foi comparado com dois algoritmos reativos (um com base em métricas de recursos e o outro com base em qualidade do serviço). Esta comparação deve-se a várias soluções encontradas – do estudo realizado que antecedeu a especiação do algoritmo – utilizarem métricas de recursos (e.g. CPU, memória, *bandwidth*) e métricas de qualidade (e.g. latência) do serviço. Os resultados da análise mostraram que os algoritmos reativos têm falhas que colocam em causa o estado do serviço. No entanto,um segundo nível de decisão vem colmatar essa falha.

Finalizando e fazendo uma análise ao estágio e aos seus objetivos, pode-se concluir que os objetivos inicialmente definidos foram alcançados. O estudo inicial serviu para conhecer as tecnologias utilizadas e a arquitetura do projeto. A especificação da arquitetura contribuiu para perceber a dinâmica do serviço e serviu de base para as métricas e *thresholds* do algoritmo. A análise entre os algoritmos permitiu perceber que o algoritmo desenvolvido tem a qualidade esperada e se cumpre com os objetivos.

6.1. Trabalho Futuro

Uma vez que o projeto ainda tem pelo menos mais quatro meses antes de terminar, existem alguns acertos que o algoritmo poderá necessitar antes de ser integrado. Primeiramente deve ser testado num ambiente de testes mais fidedigno para serem realizados testes com mais carga e averiguar se mesmo com mais pedidos o algoritmo mantém o desempenho esperado. Consoante a análise retirada deste teste, o algoritmo poderá precisar de ajustes, mas possui vários *thresholds* e características configuráveis, o processo torna-se mais simples.

Apêndices

Apêndice A – *Template* e *Stack*

O seguinte trecho de código, no formato de YAML, representa a descrição da infraestrutura requerida para o serviço que irá ser implementado na *cloud*.

```
---
description: "YAML MCN DNSaaS Template"
heat_template_version: 2013-05-23
outputs:
  mcn.endpoint.api:
    description: "IP MCN endpoint for DNSaaS API"
    value:
      get_attr:
        - server_dns_api1_floating_ip
        - floating_ip_address
  mcn.endpoint.forwarder:
    description: "IP MCN endpoint for DNSaaS Recursor"
    value:
      get_attr:
        - dns_forwarder
        - first_address
parameters:
  flavor1:
    default: m1.medium
    description: "Flavor to use for the servers"
    type: string
  flavor2:
    default: m1.large
    description: "Flavor to use for the servers"
    type: string
  image1:
    default: DNSaaS_DNS_server_API_teste
    description: "Name of image to use for DNSaaS"
    type: string
  image2:
    default: DNSaaS_Recursor
    description: "Name of image to use for DNSaaS"
    type: string
  key_name:
    default: mcn-key
    description: "Name of an existing EC2 KeyPair to enable SSH access to the instances"
    type: string
  maas_ip_address:
    default: "192.168.112.14"
    description: "MaaS Instance"
    type: string
  private_net_id:
    default: 61aa56b9-d2e7-425a-b5e2-b303d85278b3
    description: "ID of private network into which servers get deployed"
    type: string
  private_subnet_id:
    default: bd0faaa3-9c8b-4925-a6c8-4ba8b7f9f9e5
    description: "ID of private sub network into which servers get deployed"
    type: string
```

```
  public_net_id:
    default: fde9f17b-eb51-4d4b-a474-deb583d03d86
    description: "ID of public network for which floating IP addresses will be allocated"
    type: string
resources:
  dns_forwarder:
    depends_on: [ server_dns_api1 ]
    properties:
      flavor:
        get_param: flavor2
      image:
        get_param: image2
      key_name:
        get_param: key_name
      name: dnsaas-recursor
      networks:
        -
          port:
            Ref: dns_forwarder_port
      user_data_format: SOFTWARE_CONFIG
    type: "OS::Nova::Server"
  dns_forwarder_floating_ip:
    properties:
      floating_network_id:
        get_param: public_net_id
      port_id:
        Ref: dns_forwarder_port
    type: "OS::Neutron::FloatingIP"
  dns_forwarder_port:
    properties:
      fixed_ips:
        -
          subnet_id:
            get_param: private_subnet_id
      network_id:
        get_param: private_net_id
      replacement_policy: AUTO
    type: "OS::Neutron::Port"
  server_dns_api1:
    properties:
      flavor:
        get_param: flavor1
      image:
        get_param: image1
      key_name:
        get_param: key_name
      name: dnsaas-server-api1
      networks:
        -
          port:
            Ref: server_dns_api1_port
      user_data_format: SOFTWARE_CONFIG
    type: "OS::Nova::Server"
  server_dns_api1_floating_ip:
    properties:
      floating_network_id:
        get_param: public_net_id
      port_id:
```

```
      Ref: server_dns_api1_port
    type: "OS::Neutron::FloatingIP"
  server_dns_api1_port:
    properties:
      fixed_ips:
        -
          subnet_id:
            get_param: private_subnet_id
      network_id:
        get_param: private_net_id
      replacement_policy: AUTO
    type: "OS::Neutron::Port"
  dns_forwarderDeployment:
    properties:
      config:
        get_resource: dns_forwarderConfigDeployment
      input_values:
        dns_ip1:
          get_attr:
            - server_dns_api1
            - first_address
      server:
        get_resource: dns_forwarder
    type: "OS::Heat::SoftwareDeployment"
  dns_forwarderProvisioning:
    properties:
      config:
        get_resource: DNSConfigProvisioning
      input_values:
        maas_ip:
          get_param: maas_ip_address
      server:
        get_resource: dns_forwarder
    type: "OS::Heat::SoftwareDeployment"
  server_dns_api1Provisioning:
    properties:
      config:
        get_resource: DNSConfigProvisioning
      input_values:
        maas_ip:
          get_param: maas_ip_address
      server:
        get_resource: server_dns_api1
    type: "OS::Heat::SoftwareDeployment"
  dns_forwarderConfigDeployment:
    properties:
      config: |-
          #!/bin/bash
          echo -n ".=$dns_ip1"  > /dnsRecursor/forwarder_zones
          /bin/sed    sed    -i    --    "s/forward-zones-recurse=.*/forward-zones-recurse=168.192.in-
addr.arpa=$dns_ip1,92.130.in-addr.arpa=$dns_ip/g" /etc/powerdns/recursor.conf
          service pdns-recursor force-reload
      group: script
      inputs:
        -
          name: dns_ip1
    type: "OS::Heat::SoftwareConfig"
  DNSConfigProvisioning:
```

```
properties:
  config: |-
    #!/bin/bash
    /bin/sed -i -- "s/Server=.*/Server=$maas_ip/g" /etc/zabbix/zabbix_agentd.conf
    /bin/sed -i -- "s/ServerActive=.*/ServerActive=$maas_ip/g" /etc/zabbix/zabbix_agentd.conf
    service zabbix-agent restart
  group: script
  inputs:
    -
      name: maas_ip
type: "OS::Heat::SoftwareConfig"
```

O resultado do *template* tem o seguinte efeito na rede – ilustrada na Figura 45 – cria duas instâncias e fornece os respetivos endereços e dependências.

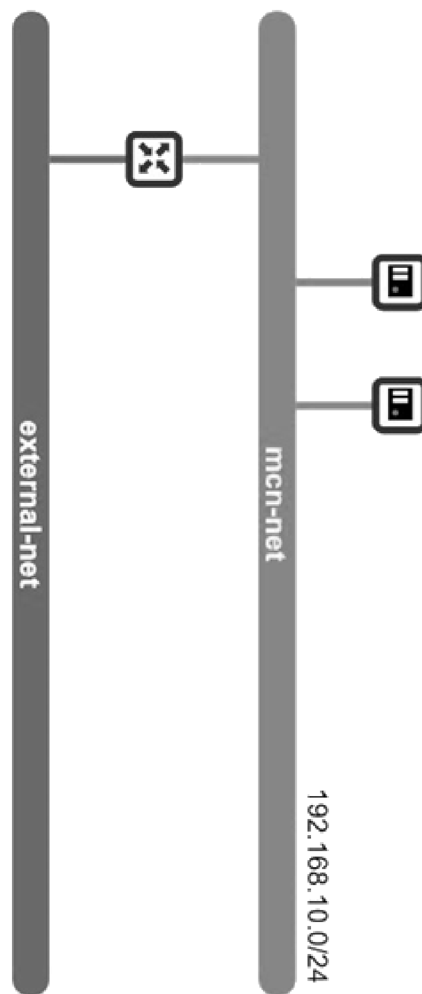


Figura 45 - Infraestrutura do *template* submetido

Apêndice B – MeTHODICAL

Este apêndice descreve de forma sucinta os diferentes passos do MeTHODICAL, conforme ilustrado na Figura 46.

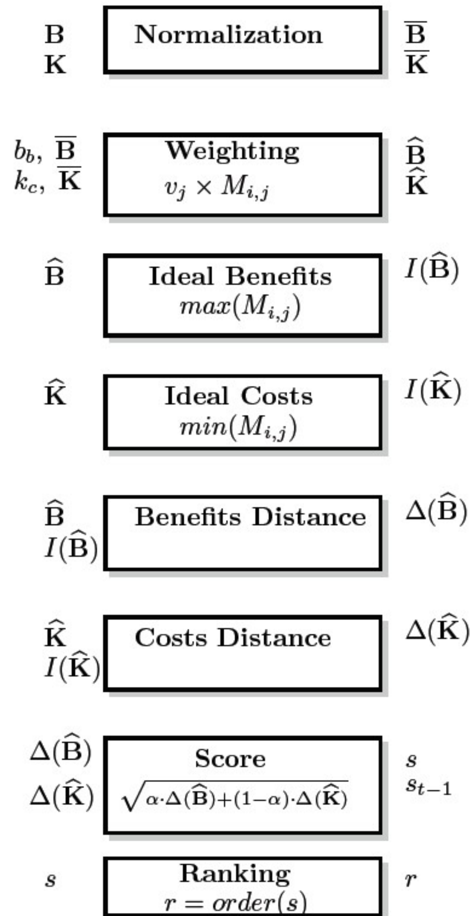


Figura 46 - Passos do Algoritmo MeTHODICAL

O primeiro passo corresponde à normalização dos diversos critérios usados na otimização. Os critérios são organizados em dois tipos: benefícios – que são para ser maximizados e custos – que são para ser minimizados. Para cada critério um valor é medido, por exemplo o critério de *queries* por segundo corresponde a um benefício visto que um valor mais alto indica melhor *performance*. Por outro lado os tempos de instanciação de uma máquina/de um serviço correspondem a custos.

Cada critério tem uma importância diferente em relação outros critérios. Por exemplo, num problema de otimização o critério de *queries* por segundo pode ser mais relevante na determinação da solução ótima do que a taxa de envio de *queries*. Assim sendo, o segundo passo corresponde à aplicação de pesos para cada critério a nível dos custos e dos benefícios.

Com os valores para as diversas métricas, os valores ideais para os benefícios e custos são determinados. Estes ideais correspondem às alternativas com os valores máximos nos benefícios e aos valores mínimos nos custos.

$$\begin{vmatrix} 10.5 & 9 & 11 \\ 15.2 & 2 & 20 \\ 11 & 5 & 23 \end{vmatrix}$$

Dado o exemplo em cima de uma matriz de benefícios o ideal corresponde ao vetor: [15.2, 9, 23]. Esta lógica aplica-se também aos custos, onde interessa ter os valores mínimos como ideais.

Após determinar os ideais, é calculada a distância de cada benefício e custos para os valores ideais. O cálculo da distância é baseado num intervalo de interesse, onde os valores mais interessantes são considerados como ótimos (ou mais próximo de). A Figura 47 e a Figura 48, ilustram graficamente a forma de determinar a distância dos benefícios e dos custos. O valor médio de um critério é considerado para permitir a correlação entre diferentes alternativas. I(x) representa o valor ideal determinado nos passos anteriores.

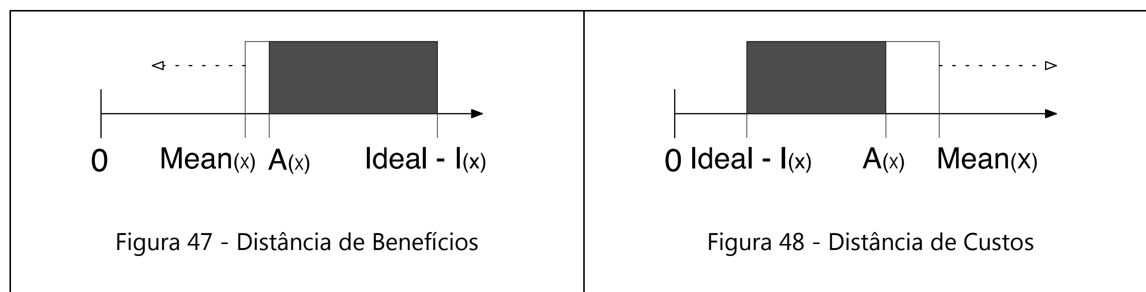


Figura 47 - Distância de Benefícios

Figura 48 - Distância de Custos

O passo da pontuação agrega os valores das distâncias dos benefícios e dos custos em termos de pontuação. É utilizado uma função para permitir balancear as distâncias de custos e benefícios. A solução ótima corresponde aquela que tem a pontuação mais baixa, visto que a distâncias são as menores para os valores ideais.

Referências

[1] MCN, "Mobile Cloud Networking", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.mobile-cloud-networking.eu/site/

[2] MCN, "Motivation", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.mobile-cloud-networking.eu/site/index.php?p=4

[3] Mashabl, "The First Cellphone Went on Sale 30 Years Ago for $4,000, março de 2014", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://mashable.com/2014/03/13/first-cellphone-on-sale/

[4] International Data Corporation, "Worldwide Smart Connected Device Forecast Market Share by Product Category", setembro de 2013, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.idc.com/getdoc.jsp?containerId=prUS24314413

[5] Independent, "John McCarthy: Computer scientist known as the father of AI", novembro de 2011, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.independent.co.uk/news/obituaries/john-mccarthy-computer-scientist-known-as-the-father-of-ai-6255307.html

[6] International Data Corporation, "Worldwide Public IT Cloud Services Spending by Segment", setembro de 2013, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.idc.com/getdoc.jsp?containerId=prUS24298013

[7] Graziano C., "A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project", 2011,Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://lib.dr.iastate.edu/cgi/viewcontent.cgi?article=3243&context=etd

[8] Caraytech Group, "GeoDNS BIND patch", janeiro de 2005, Visualizado em 20 de setembro de 2015, [Online] Disponível em: http://www.caraytech.com/geodns/

[9] C. Liu, P. Albitz, "DNS and BIND. O'Reilly Media, 2006", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://books.google.pt/books?id=HggtWI1ShvMC

[10] Chen J, "Google Public DNS: 70 billion requests a day and counting", fevereiro de 2012, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://googleblog.blogspot.pt/2012/02/google-public-dns-70-billion-requests.html

[11] Vaishnavi V., Kuechler B., "Design Science Research in Information Systems", novembro de 2012, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://desrist.org/desrist/article.aspx

[12] Bruno Sousa, David Palma, João Santos, Cláudio Marques, João Gonçalves, Hugo Fonseca, Paulo Simões, Luís Cordeiro, "Scaling DNS as a Service in the Cloud", janeiro de 2015

[13] Linthicum D., "The Great Enterprise Migration to the Cloud Begins", janeiro de 2011, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.forbes.com/sites/microsoft/2011/05/02/the-great-enterprise-migration-to-the-cloud-begins/

[14] Mell P, Grance T., "The NIST Definition of Cloud Computing", setembro de 2011, Visualizado em 27 de janeiro de 2014, [Online] http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[15] IBM, "What is Cloud?", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.ibm.com/cloud-computing/us/en/what-is-cloud-computing.html

[16] Rackspace, "Understanding the Cloud Computing Stack: SaaS, PaaS, IaaS", outubro de 2013, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.rackspace.com/knowledge_center/whitepaper/understanding-the-cloud-computing-stack-saas-paas-iaas

[17] MCN, "Overall Architecture Definition, Release 1", outubro de 2013, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.mobile-cloud-

networking.eu/site/index.php?process=download&id=124&code=93b79f8f5b99f67a6cdc28369c05b65f624cfee7
[18] Amazon, "Amazon Web Services", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://aws.amazon.com/
[19] Amazon, "Amazon EC2 Service Level Agreement", junho 2013, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://aws.amazon.com/ec2/sla/
[20] Amazon, "Amazon EC2 Pricing", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://aws.amazon.com/ec2/pricing/
[21] Microsoft, "Microsoft Azure", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://azure.microsoft.com/
[22] Amazon, "Service Level Agreements", novembro 2014, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://azure.microsoft.com/support/legal/sla/
[23] Microsoft, "Virtual Machines Pricing", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://azure.microsoft.com/en-us/pricing/details/virtual-machines/
[24] Rackspace, "Rackspace Cloud", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.rackspace.com/
[25] Lindberg V., "Rackspace's Policy On Contributing To Open Source", janeiro de 2014, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.rackspace.com/blog/rackspaces-policy-on-contributing-to-open-source/
[26] Rackspace, "Cloud Big Data Platform SLA", fevereiro de 2011, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.rackspace.com/information/legal/cloud/sla/ sla rackspace
[27] Rackspace, "Rackspace Public Cloud", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.rackspace.com/cloud/public-pricing
[28] Lunacloud, "Lunacloud cloud software", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.lunacloud.com/
[29] Kenny U, "Cloudspecs performance report lunacloud, amazon ec2, rackspace cloud", novembro de 2012, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.lunacloud.com/docs/reports/cloudspecs-performance-report.pdf
[30] Lunacloud, "Service Level Agreement", outubro de 2012, Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.lunacloud.com/en_eu/sla
[31] Lunacloud, "Pricing", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.lunacloud.com/en_eu/cloud-server-precos
[32] OpenStack, "OpenStack cloud software", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.openstack.org/
[33] OpenStack, "Companies Supporting The OpenStack Foundation", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.openstack.org/foundation/companies/
[34] OpenStack, "Manage flavors", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/user-guide-admin/content/cli_manage_flavors.html
[35] OpenStack, "Architecture", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/havana/install-guide/install/yum/content/ch_overview.html
[36] OpenStack, "Horizon", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/horizon/
[37] OpenStack, "Keystone", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/keystone/
[38] OpenStack, "Nova", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/nova/
[39] OpenStack, "Neutron", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/neutron/
[40] OpenStack, "Swift", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/swift/

[41] OpenStack, "Cinder", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/cinder/
[42] OpenStack, "Glance", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/glance/
[43] OpenStack, "Ceilometer", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/ceilometer/
[44] OpenStack, "Heat", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://docs.openstack.org/developer/heat/
[45] OpenStack, "Designate", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://wiki.openstack.org/wiki/Designate
[46] PowerDNS, "PowerDNS", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://www.powerdns.com/
[47] PowerDNS, "PowerDNS API", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://doc.powerdns.com/md/httpapi/README/
[48] PowerDNS, "Scripting The Recursor" ", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://doc.powerdns.com/md/recursor/scripting/
[49] Internet Systems Consortium, "BIND The most widely used Name Server Software", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: https://www.isc.org/downloads/bind/
[50] Z. Gong, X. Gu, and J. Wilkes, "PRESS: PRedictive elastic ReSource scaling for cloud systems", outubro de 2010, pp. 9–16.
[51] F. Almeida Morais, F. Vilar Brasileiro, R. Vigolvino Lopes, R. Araujo Santos, W. Satterfield, and L. Rosa, "Autoflex: Service Agnostic Auto-scaling Framework for IaaS Deployment Models," 2013
[52] J. Jung, Sit E., Balakrishnan H., Morris R., "DNS performance and the effectiveness of caching", 2012
[53] K. Park, V. S. Pai, L. L. Peterson, and Z. Wang, "CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups, 2004
[54] Nomium, "Network measurement tools", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://nominum.com/support/measurement-tools
[55] Zabbix, "Zabbix - the enterprise-class monitoring solution for everyone", Visualizado em 27 de janeiro de 2014, [Online] Disponível em: http://www.zabbix.com/
[56] Bruno Sousa, Kostas Pentikousis, Marília Curado, "MeTHODICAL: Towards the next generation of multihomed applications", 2 de junho de 2014
[57] Maitreya Natu, Sangameshwar Patil, Vaishali Sadaphal, Harrick, "Automated Debugging of SLO Violations in Enterprise Systems", 9 de janeiro de 2010