FCTUC FACULDADE DE CIÊNCIAS
E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Improving the Performance of a Self-Adaptive Cloud Environment

Relatório de Tese de Mestrado em Engenharia Informática

Vítor Hugo Simões Moura da Silva
vhsilva@student.dei.uc.pt
Número de Estudante: 2007180412
Número de Indentificação: 13568475

Sob orientação de
Mário Alberto da Costa Zenha Rela
Raul André Brajczewski Barbosa

01 de julho de 2014

Sumário

Sistemas *Self-Adaptive* monitorizam continuamente e em tempo real propriedades, a fim de serem analisadas e planeadas estratégias de adaptação, que serão executadas para ajustar o seu comportamento, de modo a alcançar objectivos de qualidade. Estes sistemas têm sido aplicados em vários cenários, como sistemas de gestão de informação e sistemas que conseguem conduzir automóveis sem intervenção humana. No entanto, por serem tão complexos, impõem um significativo custo computacional. Como não existe, ainda, um conhecimento sólido sobre o consumo de recursos computacionais nas fases de adaptação, isso motivou o estudo que é objecto desta tese de mestrado.

Nesta tese, é avaliada a performance (em termos de recursos computacionais) de um modelo arquitectural presente em vários sistemas *Self-Adaptive* - *MAPE-K loop*. Com base nessa avaliação, são identificados *bottlenecks*, que ao serem corrigidos permitem uma melhoria significativa da solução adoptada. Por último, são apresentadas comparações entre as diferentes abordagens (sem melhorias e com melhorias), de modo a confirmar que a melhoria resultou numa proposta que reduz monetariamente o custo, se o sistema estiver alojado em ambientes cloud (AWS). É demonstrado que o sistema optimizado permite uma redução de 45 % nos custos de operação, sem qualquer redução nos parâmetros de qualidade.

Palavras-Chave: *Self-Adapive*, *MAPE-K loop*, recursos computacionais, objectivos de qualidade.

Dedicatória:

Esta Tese é dedicada à minha família, em especial aos meus pais, por todo o apoio que me deram ao longo do meu percurso académico.

Agradecimentos:

Queria, em primeiro lugar, agradecer aos meus pais o apoio que me deram durante o meu percurso e, sobretudo, a paciência que tiveram para que este ciclo da minha vida se fechasse. À minha irmã que me acompanhou ao longo destes anos e que sempre bons conselhos me deu.

Ao João Franco, pelo seu incansável apoio ao longo deste ano. A sua ajuda e orientação foram determinantes para a elaboração desta tese.

Aos Professores Mário Zenha Rela e Raul Barbosa, pelo rigor e espírito critico que me foram incutindo durante este ano.

Ao Francisco Correia, pela sua disponibilidade em me auxiliar na resolução de problemas que surgiram no decorrer da elaboração deste trabalho.

Ao Professor Luís Paquete, pelas soluções que me facultou para a resolução de problemas inerentes ao estudo efectuado.

À Professora Sónia Costa, pela sua disponibilidade no acto de revisão desta tese.

À minha família e amigos, que me acompanharam ao longo do meu percurso académico.

A todos vós, o meu muito obrigado.

Conteúdo

Lista de Figuras **ix**

Lista de Tabelas **xi**

Lista de Acrónimos **xiii**

1 Introdução **1**

2 Planeamento do Trabalho **5**

3 Estado da Arte **7**

3.1 Sistema - Conceito e Implicações 7

3.1.1 Função, Comportamento, Estrutura e Serviço de um Sistema . . 8

3.1.2 *Faults*, *Errors* e *Failures* de Sistema 8

3.2 Arquitectura de Software 9

3.3 Sistemas *Self-Adaptive* 10

3.3.1 Propriedades *Self-** 11

3.3.2 Objectivos de Adaptação 12

3.3.3 Algoritmo *MAPE-K loop* 13

3.3.4 Projetos Existentes 15

3.4 Ferramentas de *Benchmarking* de *web servers* 18

3.5 *Profiling* 21

3.5.1 Ferramentas de *Profiling* 21

4 Abordagem **23**

4.1 Selecção de Ferramentas a Utilizar 23

4.1.1 *Self-Adaptive System* 23

4.1.2 *Web-Server Benchmark* . . . 25
4.1.3 *Profiling* . . . 25
4.2 Caso de Estudo . . . 26
4.2.1 Especificação de *Hardware* . . . 27
4.2.2 Objectivos de Adaptação . . . 28
4.2.3 Tácticas de Adaptação . . . 29
4.2.4 Tipos de Adaptação . . . 29
4.2.5 *Workload* . . . 30
4.2.6 Introdução de Falhas . . . 31

5 Método **33**
5.1 Testes de *Benchmarking* de *web servers - Jmeter* . . . 33
5.1.1 *Scripts* . . . 33
5.1.2 Limitações . . . 34
5.1.3 Métricas Extraídas . . . 35
5.2 Testes de *Profiling - NetBeans Profiler* . . . 36
5.2.1 Limitações . . . 37
5.2.2 Detecção do Problema . . . 38
5.2.3 Resolução do problema . . . 40

6 Resultados **43**
6.1 Objectivos de Adaptação . . . 43
6.2 Recursos Computacionais . . . 47

7 Análise das Observações **53**
7.1 Objectivos de adaptação . . . 53
7.2 Recursos Computacionais . . . 55
7.3 Impacto no Custo Operacional . . . 56
7.4 Limitações . . . 57

8 Conclusão **59**

Referências **61**

Anexo **65**

A Diagrama de Gantt referente ao Planeamento do trabalho realizado 65

Lista de Figuras

2.1 *Lista de tarefas para o segundo semestre* 6

3.1 *Adaptation Loop* 14

4.1 *Raibow Framework* 24
4.2 *Representação do Sistema em estudo* 26
4.3 *Slashdot workload* 30

5.1 *Representação do sistema com Profiler agents* 36
5.2 Resultados obtidos pelo procedimento *Profiler* 39

6.1 Resultados dos Atributos de Qualidade - Teste *Baseline* 44
6.2 Resultados dos Atributos de Qualidade - Teste *Delay* 45
6.3 Comparação do consumo médio de CPU entre sistemas *Non-Adaptive* e *Adaptive* 48
6.4 Comparação do consumo médio de memória entre sistemas *Non-Adaptive* e *Adaptive* 50

A.1 *Lista de tarefas para o primeiro semestre* 66

LISTA DE FIGURAS

Lista de Tabelas

2.1 Planeamento sumário do primeiro trabalho realizado no primeiro semestre 5
2.2 Planeamento sumário do primeiro trabalho realizado no segundo semestre 6

3.1 Objectivos de Adaptação implementados pelas Propriedades Self-* (✓): implementa, (-): não implementa 13
3.2 Lista de sistemas *Self-Adaptive* seleccionados para análise 15
3.3 Propriedades *Self-** implementadas pelos sistemas: ✓ Implementa, - Não implementa. 16
3.4 Comparação de outras características de sistemas *Self-Adaptive* 17
3.5 Lista de Ferramentas de *Benchmarking* de *Web Servers* 20
3.6 Propriedades das Ferramentas de *Profiling* selecionadas 22

4.1 *Rainbow Utility preferences* 28

6.1 Dados relativos aos Atributos de Qualidade - Teste *Baseline* 46
6.2 Dados relativos aos Atributos de Qualidade - Teste *Delay* 46
6.3 Comparação do consumo de CPU - sistemas *Non-Adaptive* e *Adaptive* nos testes com e sem introdução de falhas 49
6.4 Comparação do consumo de Memória - sistemas *Non-Adaptive* e *Adaptive* nos testes com e sem introdução de falhas 51

7.1 Comparação do custo de máquinas virtuais entre sistemas *Non-Adaptive* e *Adaptive* 56

LISTA DE ACRÓNIMOS

Lista de Acrónimos

ABAS Atribute-Based Architecture Styles 9

ADL Architectural Description Languages 9

AWS Amazon Web Services ii

CPU Central Processing Unit 16

CSV Comma-separated values 24

DEI Departamento de Engenharia Informática 26

FTP File Transfer Protocol 19

GUI Graphical User Interface 21

HTML HyperText Markup Language 19

HTTP Hypertext Transfer Protocol 19

HTTPD Hypertext Transfer Protocol Daemon 5

HTTPS HTTP Secure 19

IIOP Internet Inter-Orb Protocol 19

JMS Java Message Service 19

LDAP Lightweight Directory Access Protocol 19

PDF Portable Document Format 19

POP3 Post Office Protocol 3 19

LISTA DE ACRÓNIMOS

QoS Quality of Service 15

RMI/IIOP Remote Method Invocation sobre IIOP 19

RMI/JRMP Java Remote Method Invocation 19

SLA *Service Level Agreement* 15

SLO Service Level Objective 15

SMTP Simple Mail Transfer Protocol 19

SOAP Simple Object Access Protocol 19

SSH Secure Shell 26

TCP Transmission Control Protocol 18

XML Extensible Markup Language 19

XML-RPC Remote procedure call que usa XML 19

1

Introdução

Sistemas que se auto-adaptam são capazes de ajustarem o seu comportamento, quando deparados com mudanças no ambiente ou, mesmo, no próprio sistema. Operam com mínima ou nenhuma intervenção humana e estão desenhados para atingir determinados objectivos de qualidade, como *availability* ou *performance*. Estes sistemas, além das funcionalidades básicas, requerem um conjunto de tarefas complementares independentes para monitorizar, analisar, decidir e actuar, a fim de responder a mudanças, mantendo os níveis de qualidade consistentes. No entanto, um sistema que está constantemente a monitorizar informação, planear decisões e avaliar possíveis estratégias de adaptação, necessita de consumir mais recursos computacionais, quando comparado com soluções sem adaptação. Este *overhead* computacional pode ter um impacto negativo na performance do sistema, especialmente quando este está sujeito a um aumento de trabalho (maior volume de informação a processar). Do que foi investigado, até ao momento, não existe grande informação acerca do impacto dos mecanismos de adaptação no consumo dos recursos computacionais, nem um claro conhecimento sobre a quantidade de recursos consumidos pelas diferentes fases de adaptação, no entanto Cheng et al. [11](artigo motivador do tema desta tese) afirma que um dos processos(*Delegate*) de adaptação consome menos de 2 % de CPU;

Nesta tese é analisado o algoritmo *MAPE-K loop*, proposto pela IBM [23], em termos de performance e custo de *deploying*. Especificamente, é comparado o desempenho entre soluções adaptativas e não adaptativas, tendo em consideração que ambas atingem os seus objectivos de qualidade. Por último, são identificados os *components* que consomem mais recursos computacionais e é apresentada uma solução que corrige os

problemas detectados, reduzindo, assim, o *overhead* computacional e, mantendo, os objectivos de qualidade com os valores desejados;

Esta tese contribui para a comunidade cientifica da seguinte forma:

- Validar a eficácia do *Rainbow* (sistema *Self-Adaptive* seleccionado como objecto em estudo) - Os resultados apresentados no Capítulo 6 permitem concluir que o *Rainbow* consome baixos recursos computacionais, em termos de CPU, na versão optimizada. Assim, é validada a proposta de [11], em que afirma que na aplicação *Delegate*, o consumo médio de CPU é de 2 %.

- Avaliar o *overhead* do *Controller* (processo responsável por executar as fases *Analyze*, *Plan*, *Execute* e *Knowledge*, do *MAPE-K loop*) - Neste trabalho pretendeu-se avaliar o *overhead* computacional do *Controller* devido à pouca investigação realizada sobre o assunto. O Capítulo 6 mostra que o *overhead* é baixo, registando um consumo médio de 3.5 % de CPU e 128.8 MB de memória.

- Identificar o *overhead* nas diferentes fases do *MAPE-K loop* - Este estudo fornece informação sobre quais as fases do *self-adaptive loop* requerem maior atenção em trabalhos de investigação futuros ou de implementação de novas versões. Além disso, para se proceder a qualquer melhoria ao sistema, é necessário avaliá-la através de *profiling*. Pelo o que foi observado, pode-se concluir que a fase de monitorização consome uma grande porção de recursos, comparando com as outras fases. É neste ponto que se devem unir esforços para melhorar a eficácia das técnicas de monitorização e recolha de informação;

- Comparação entre abordagens *Non-Adaptive* e *Self-Adaptive* - Mostrou-se, neste estudo, que sistemas *Self-Adaptive* partem em vantagem quando comparados com soluções *Non-Adaptive*, especialmente em termos de custo. Além disso, estes sistemas requerem poucos recursos e podem ser aplicados numa grande variedade de cenários;

- Submissão de artigo - Foi escrito um artigo que contempla o estudo realizado nesta tese e que foi submetido a uma conferência de investigação científica, na área da ciência da computação.

No procedimento de avaliação, adoptou-se pelo *Rainbow* [18] [11] como solução *Self-Adaptive* e aplicou-se o caso de estudo de uma infraestrutura de notícias, com um *workload* baseado num fenómeno real que ocorre na Internet, representando um fluxo de tráfego entre clientes e sistema (*slashdot effect*). Em cada experiência, foi recolhida informação acerca do desempenho, utilização de CPU e memória, que permitiu identificar problemas de implementação e possíveis melhorias dos mesmos.

Este documento é organizado da seguinte forma:o Capítulo 2 demonstra o planeamento do trabalho realizado, o Capítulo 3 é responsável por introduzir ao leitor o trabalho que a comunidade científica tem realizado na área, bem como apresentar sistemas *Self-Adaptive* e ferramentas que os possam avaliar. No Capítulo 4 são fundamentadas decisões que vão influenciar a realização do trabalho, fundamentadas de acordo com o que foi demonstrado no Estado da Arte. O Capítulo 5 mostra como avaliámos o sistema *Self-Adaptive*, como detectámos o seu problema e como o resolvemos. Depois, o Capítulo 6 ilustra os resultados obtidos antes e depois da melhoria do sistema. Esta informação será útil para que seja feita uma análise de resultados no Capítulo 7. Finalmente, o Capítulo 8 expõe as conclusões retiradas deste estudo.

2

Planeamento do Trabalho

A Tabela 2.1, representa a lista sumária de tarefas que foram realizadas no primeiro semestre.

Numa primeira fase, com vista à familiarização dos conceitos envolvidos no tema, a realização do estado da arte demorou cerca de dois meses, pois foi realizado trabalho de pesquisa e recolha de informação para análise. Depois de interpretada a informação do estado das arte, tomaram-se decisões relativas à escolha do sistema e às ferramentas a utilizar no segundo semestre. Foi, ainda, neste semestre que alguns testes preliminares foram realizados, o que possibilitou ter uma ideia do que estava a ocorrer no sistema. Algo que demorou cerca de dois meses.

Tabela 2.1: Planeamento sumário do primeiro trabalho realizado no primeiro semestre

	Data Inicial	Data Final
Estado da Arte	06/09/2013	22/11/2013
Abordagem	24/11/2013	19/01/2014
Revisão do Relatório	20/01/2014	28/01/2014

Por último, foi realizada a conclusão e revisão do relatório.

É apresentado em anexo o diagrama de Gantt referente ao planeamento do primeiro semestre no Anexo A.1.

Como mostra a Tabela 2.2, o segundo semestre foi dividido em cinco fases. Numa primeira fase foram feitos testes de *benchmark* ao sistema ZNNnews (Infraestrutura que fornece notícias) de forma detalhada. Isto é, além de se recolher informação acerca do *stress* definido pelo *workload*, pretendeu-se realizar medições de utilização de CPU

e memória. Para tal, foram analisados somente os processos que intervêm no sistema, isto é, o processo oracle.sh(controlador), rundelegate.sh (delegate) e os processos httpd(*Hypertext Transfer Protocol Daemon*), o que demorou cerca de um mês.

Tabela 2.2: Planeamento sumário do primeiro trabalho realizado no segundo semestre

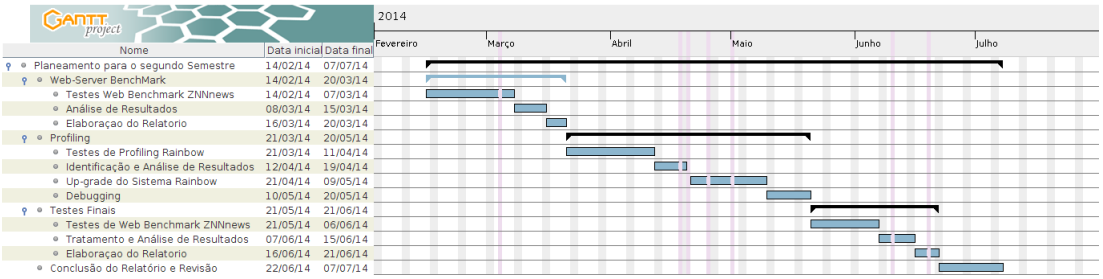
	Data Inicial	Data Final
Testes de Benchmarking	14/02/2014	20/03/2014
Profiling e Implementação	21/03/2014	20/05/2014
Testes de Aceitação	21/05/2014	06/06/2014
Redacção de Artigo	07/06/2014	20/06/2014
Conclusão e Revisão do Relatório	21/06/2014	01/07/2014

Depois de recolhida e analisada a informação, procedeu-se à segunda fase do plano, que passou por efectuar *profiling* ao algoritmo, de forma a detectar problemas de implementação para proceder à sua correcção. Processo foi cumprido de acordo com intervalo de tempo apresentado.

A terceira fase foi, de certa forma, uma repetição da primeira, no entanto com todas as mudanças efectuadas. De seguida, procedeu-se a redacção de um artigo acerca do que foi estudado nesta tese. Por fim, realizou-se a conclusão e revisão do relatório final.

A Figura 2.1 ilustra o diagrama de Gantt referente ao planeamento do segundo semestre.

Figura 2.1: *Lista de tarefas para o segundo semestre*



Em suma, foi cumprido o que tinha sido inicialmente planeado e ainda houve espaço para a redacção de um paper que foi submetido a uma conferência de investigação cientifica, na área da ciência da computação.

3

Estado da Arte

Este capítulo destina-se a clarificar conceitos teóricos e tecnologias que são contemplados ao longo desta tese, de modo a familiarizar o leitor com as metodologias desenvolvidas assim como a respectiva contextualização.

Desta forma, serão abordados temas intimamente relacionados com o conceito de sistema e respectivas implicações, arquitectura de software, sistemas *Self-Adaptive*, de modo a ser possível avaliar as condições de elegibilidade do melhor sistema.

O estudo das Ferramentas de *Web Servers Benchmarking* e *profiling* será determinante para aprovar quais as ferramentas adequadas na obtenção de dados acerca dos recursos computacionais que advêm dos sistemas *Self-Adaptive* e investigar as fases de adaptação que requerem futuros *improvements*.

3.1 Sistema - Conceito e Implicações

Este tópico tem como objectivo apresentar o conceito de sistema, de forma a esclarecer o leitor acerca da sua importância quer para um sistema *Self-Adaptive* quer para um sistema *Non-Adaptive*. Perceber o significado de função, comportamento, estrutura e serviço de um sistema é fundamental, pois seguindo este procedimento é possível alcançar a inteligibilidade da ocorrência de *faults*, *failures* e *errors*.

No final desta secção, serão apresentadas técnicas que atenuam possíveis problemas que possam ocorrer num sistema de informação.

3.1.1 Função, Comportamento, Estrutura e Serviço de um Sistema

De acordo com Avizienis et al. [3], um sistema é uma entidade que interage com outras entidades. Um simples site é exemplo disso, uma vez que para processar informação entre Cliente-Sistema, necessita da interacção entre *components* de hardware, *components* de software e humanos. Outros sistemas podem ainda interagir com mudanças no ambiente (mundo físico).

A **função** de um sistema é executar o que fora enunciado pelas especificações funcionais e não funcionais. O seu **comportamento** expressa todos os passos efectuados para realizar a sua função, sendo descrito por uma sequência de estados. A sua **estrutura** é activada para gerar o seu comportamento, passando a ser **serviço** no momento em que o comportamento gerado é interpretado pelo utilizador.

3.1.2 *Faults*, *Errors* e *Failures* de Sistema

Candea [10] relata que, no ano de 1999, o eBay esteve indisponível durante 22 horas com problemas contínuos durante vários dias, devido ao facto de o servidor da Sun ter crashado. Neste exemplo, é verificável a ocorrência de um *system fault* que originou um *system error*, o qual se concretizou através de um *system failure*.

Neste caso, a função do sistema foi "corrompida", o que foi visível ao nível do seu comportamento, não tendo sido possível a prestação do serviço esperado pelo cliente.

Sommerville [37] sugere que um ***system fault*** corresponde a uma característica do sistema que o pode conduzir a um ***system error***, ou seja, se ocorrer uma falha ao inicializar uma variável, aquela pode levar a que essa variável tenha um valor errado no momento da sua utilização.

Quando um certo evento sobrevém no sistema e o impede de entregar um serviço, tal como ele é expectável pelo utilizador, verifica-se a presença de um ***system failure***.

De modo a responder à problemática enunciada, existem técnicas que previnem e/ou minimizam os possíveis problemas que possam ocorrer, tais como:

- *Fault Avoidance*
- *Fault Detection and Removal*
- *Fault Tolerance*

Fault Avoidance baseia-se em técnicas de desenvolvimento que permitem minimizar e/ou evitar *system faults*. ***Fault Detection and Removal*** utiliza técnicas de teste e debugging que contribuem para a verificação e validação de *system faults* que, ao serem detectadas podem ser removidas antes de serem activadas pelo sistema. ***Fault Tolerance*** previne-se do facto de *system faults* não resultarem em *system errors*, ou de *system errors* não resultarem em *system failures*.

Podemos concluir que é muito importante a forma como um sistema é desenhado, pois assim é possível prevenir ou resguardar um desequilíbrio que possa ocorrer no mesmo. A construção da sua arquitectura de software pode ser decisiva para o sucesso da sua função e, como consequência, do seu comportamento, entregando assim os serviços previstos pelo cliente.

3.2 Arquitectura de Software

Geralmente, autores que já se debruçam sobre arquitectura de software têm preferido compara-la a estruturas estáticas, como edifícios, monumentos, entre outras estruturas arquitectónicas. Clements et al. [12] introduzem um ponto de vista dinâmico acerca do tema - a asa de um pássaro.

Com efeito, uma asa é composta por vários *components* como ossos, penas, músculos, nervos e vasos sanguíneos. Cada *component* tem a sua função e quando todos estão operacionais, conseguem alcançar o objectivo da asa, ajudando o pássaro a voar. Além disso, a asa possui atributos de qualidade e de relevância, tais como leveza, aerodinâmica e protecção térmica que permitem o sucesso do seu propósito.

Contrariamente à metáfora clássica, que apresenta uma estrutura estática em que o seu comportamento não varia ao longo do tempo, a asa permite a deslocação do pássaro variando a sua velocidade consoante as suas necessidades.

Actualmente, a arquitectura de software tem seguido o paradigma dinâmico em detrimento do estático, pois, sendo os sistemas dotados de uma complexidade cada vez maior, quando confrontados com mudanças, têm a necessidade de reagirem às mesmas, de modo a cumprirem com os atributos de qualidade a que se propuseram. Comportamento semelhante apresentam os sistemas *Self-Adaptive*, que, aquando da confrontação com mudanças alheias ao seu normal funcionamento, recorrem à adaptação a fim de cumprir os seus objectivos.

De acordo com Taylor et al. [38], a arquitectura de software detém ***components*** (que encapsulam um subconjunto de dados ou de funcionalidade do sistema e restringem o acesso a esse subconjunto, através de uma interface explicitamente definida, possuindo dependências declaradamente definidas no seu contexto de execução), ***connectors*** (cuja função é efectuar e regular interacções entre os *components*)e ***configuration*** (conjunto de associações especificas entre *components* e *connectors*).

Neste contexto, sistemas *Self-Adaptive* recorrem a modelos e a linguagens de arquitectura de software que permitem, em tempo real, reconfigurar a arquitectura de software de um sistema alvo, de forma a responder às suas necessidades e alcançar os seus objectivos de adaptação.

Os tipos de modelos e linguagens de arquitectura de software, como *Architectural Description Languages* e *Atribute-Based Architecture Styles*, podem, no primeiro caso e segundo Ghosh et al. [19], contribuir para a modelação e gestão de software, acrescentando o facto de essas tarefas poderem ser realizadas durante o tempo de execução do sistema. ADLs como o Acme, xADL e Darwin têm sido utilizados para representar arquitecturas de sistema facilitando a adaptação e a reconfiguração de *components* do sistema. Tipicamente, usam conjuntos de *components*, *connectors* e reutilizam *interface*. Geralmente, adicionam planos de reparação no sistema, que tem como objectivo restaurar *components* débeis.

No segundo caso, como demonstra Klein and Kazman [24], ABAS mostra como este pondera decisões arquitecturais, tendo em conta atributos de qualidade, como *performance*, *availability*, entre outros, nas suas decisões.

Na próxima secção serão apresentados sistemas *Self-Adaptive* que poderão basear-se numa das duas opções anteriormente descritas.

3.3 Sistemas *Self-Adaptive*

Tem surgido, ao longo da ultima década, um grande investimento na concepção e melhoramento de sistemas *Self-Adaptive*. Este facto prende-se com o desejo de introduzir sistemas que se comportem de forma autónoma, quando confrontados com mudanças alheias à intencionalidade dos seus objectivos de origem.

De acordo com Oreizy et al. [31], sistemas *Self-Adaptive* modificam o seu próprio comportamento (quando necessitam) para responderem às mudanças provocadas pelo

ambiente em que operam.

Logo, afigura-se importante perceber quais as suas propriedades, quais os objectivos que pretendem atingir e qual o algoritmo em que se baseiam. No final desta secção, serão apresentados e analisados sistemas *Self-Adaptive*, cuja documentação publicada se encontra disponibilizada com livre acesso.

3.3.1 Propriedades *Self-**

No artigo "*An architectural blueprint for autonomic computing*", publicado pela IBM [23], faz-se referência ao facto de as capacidades de *Self-managing* (propriedade de um sistema se auto gerir com mínima intervenção humana ou mesmo com ausência desta) [21] cumprirem as suas funções tomando acções de acordo com o que é pressentido no ambiente em que o sistema está inserido. A função é o ciclo de controlo que recolhe informação sobre os *components* de hardware e software do sistema alvo para, de seguida, agir em conformidade. Este ciclo de controlo é composto pelas seguintes propriedades:

- *Self-configuring* - quando surgem mudanças no ambiente, reconfigura dinâmica e automaticamente o sistema; as possíveis mudanças poderão ser a implementação de novos *components* ou a remoção de *components* existentes;

- *Self-healing* - tem a capacidade de descobrir, diagnosticar e reagir a perturbações no sistema; além disso, antecipa potenciais problemas e toma as devidas decisões para prevenir possíveis falhas;

- *Self-optimizing* - pretende gerir o desempenho e a alocação de recursos do sistema, respondendo aos diferentes pedidos de diferentes utilizadores, de forma mais eficaz;

- *Self-protecting* - detecta e corrige transgressões de segurança no sistema, isto é, defende o sistema contra ataques maliciosos, e antecipa problemas facilitando a mitigação dos mesmos.

Como se pode verificar, estas propriedades estão directamente relacionadas com os objectivos de adaptação que serão explanados na secção seguinte.

3.3.2 Objectivos de Adaptação

De acordo com Villegas et al. [39], objectivos de adaptação são a razão pela qual se justifica adoptar uma abordagem *Self-Adaptive*, podendo ser definidos por uma ou mais propriedades *Self-**. O seu propósito é preservar a qualidade de serviços em termos mais específicos, ou regular, mais abrangentemente, todos os requisitos não funcionais.

Desta forma, são definidos os seguintes atributos de qualidade:

- *Maintainability*: capacidade de um sistema poder ser melhorado ou optimizado sem ser necessário efectuar mudanças internas no mesmo, mas sim adaptá-lo ou reconfigurá-lo a fim de cumprir as suas necessidades [29];
- *Functionality*: capacidade de um sistema fornecer serviços e funções requisitados [29];
- *Portability*: capacidade e facilidade de execução de um sistema, quando confrontado com uma mudança de plataforma (por exemplo, a execução do sistema em ambientes Unix vs Windows) [29];
- *Usability*: medida de qualidade do sistema em relação à forma como ele interage com os utilizadores [29];
- *Availability*: probabilidade de um sistema activo e em execução, estar apto para responder ao que lhe foi requisitado [37];
- *Survivability*: capacidade de um sistema alcançar o seu propósito, na presença de ataques, falhas ou acidentes [14];
- *Reliability*: capacidade de um sistema entregar correctamente os serviços esperados pelo utilizador, sobre um dado período de tempo[37];
- *Performance*: O'Brien et al. [29] defendem que performance é composta por *response time* e *throughput*; Assim:
  - *Response time*: tempo de resposta de um pedido feito pelo cliente;
  - *Throughput*: quantidade de informação processada por unidade de tempo.

Contudo, Barbacci et al. [4] afirmam que todos os atributos de qualidade afectam o *cost*, que segundo Beloglazov and Buyya [6] é, em termos energéticos, a quantidade de energia consumida por uma máquina física. Por outro lado, no tocante às máquinas virtuais, o custo de um sistema significa a quantidade de dinheiro gasto por cada máquina activa, tendo em conta que alugamos o serviço a uma *store*, como a *Amazon AWS*.

Tabela 3.1: Objectivos de Adaptação implementados pelas Propriedades Self-* (✓): implementa, (-): não implementa

Propriedades / Objectivos de adaptação	*Self configuring*	*Self healing*	*Self optimizing*	*Self protecting*
maintability	✓	✓	-	-
functionality	✓	-	✓	✓
portability	✓	-	-	-
usability	✓	-	-	-
availability	-	✓	-	-
survivability	-	✓	-	-
reliability	-	✓	-	✓
performance	-	-	✓	-

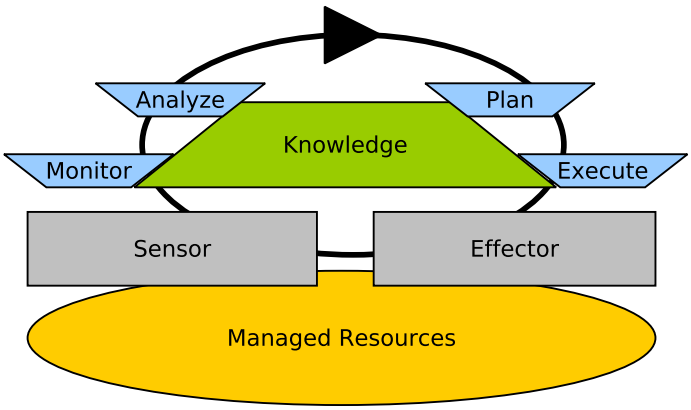
Baseado em Salehie and Tahvildari [35], a Tabela 3.1 contempla os objectivos de adaptação que podem ser implementados pelas propriedades *Self-**. Esta informação será muito útil aquando da caracterização de sistemas *Self-Adaptive*, uma vez que fornecerá indicações sobre aquele que será capaz de implementar mais atributos de qualidade no sistema.

3.3.3 Algoritmo *MAPE-K loop*

Como já foi referido na Secção 3.3, sistemas *Self-Adaptive* têm como objectivo de automatizar processos, descartando a intervenção humana que, muitas vezes, pode ser propensa a erros, despendendo tempo e dinheiro. Neste sentido, a IBM introduziu o MAPE-K loop [23], que se tornou o algoritmo referência na comunidade *Self-Adaptive* [35]. O acrónimo MAPE-K significa *[M]-Monitor*, *[A]-Analyze*, *[P]-Plan*, *[E]-Execute* e *[K]-Knowledge*.

A Figura 3.1 mostra o funcionamento de um algoritmo *closed-loop*. Além disso, observa-se que a figura é composta por processos, fluxo de dados e interfaces, em que

Figura 3.1: *Adaptation Loop*



Monitor, *Analyse*, *Plan*, *Execute* e *Knowledge* correspondem aos processos, *Sensor* e *Effector* às interfaces e *Managed Resources* ao sistema a ser adaptado. O fluxo de dados é toda a informação que interage com as interfaces e processos.

A função das interfaces e dos processos são as seguintes:

- *Sensor* - recolhe os dados, em tempo real, do sistema alvo (*Managed Resources*), enviando os mesmos para o processo *Monitor*;
- *Monitor* - faz o tratamento dos dados, de acordo com o input requisitado pelo processo *Analyse*;
- *Analyse* - analisa a informação enviada pelo processo anterior e determina se é necessária ou não adaptação no sistema alvo, de acordo com os objectivos de adaptação definidos;
- *Plan* - decide qual a estratégia de adaptação a tomar de acordo com os objectivos de adaptação definidos;
- *Execute* - executa o que foi decidido pela fase anterior;
- *Effectors* - interface que põe em prática o processo anterior no sistema alvo;
- *Knowledge* - partilha informação com as diferentes fases do MAPE-K loop, incluindo o modelo, propriedades recolhidas em tempo real e resultados de análises do sistema alvo.

3.3.4 Projetos Existentes

Tendo em consideração o que foi abordado nas subsecções anteriores, podemos, nesta fase, avaliar os sistemas *Self-Adaptive* presentes no mundo cientifico e empresarial. É importante salientar que os sistemas que apresentaremos foram escolhidos com base na sua maturação e na informação a que pudemos ter acesso.

Deste modo, a Tabela 3.2 apresenta uma breve descrição dos sistemas recolhidos. Os dados a analisar baseiam-se no artigo "Self-Adaptive Software:Landscape and Research Challenges" [35] e no artigo "A Framework for Evaluating Quality-Driven Self-Adaptive Software Systems" [39].

Tabela 3.2: Lista de sistemas *Self-Adaptive* seleccionados para análise

Sistemas	Breve Descrição
Oceano [15]	Protótipo de uma infraestrutura, focada na área de *e-business*, que oferece bons níveis de *availability*, *scalability* e é fácil de gerir. Permite que múltiplos clientes possam estar alojados num conjunto de recursos partilhados. O seu grande objectivo é garantir bons níveis de *performance* e *cost* agregados aos atributos que já oferece. Implementa o algoritmo *MAPE-K loop*.
Rainbow [18]	Consiste numa infraestrutura de adaptação que recorre à arquitectura de software de um sistema alvo, de modo a poder adaptá-lo reconfigurando a mesma. Trata-se de uma infraestrutura flexível pois reage de acordo com a estrutura e objectivos do sistema alvo em questão. Auxilia-se da linguagem Acme para manipular a dita reconfiguração. Implementa o algoritmo *MAPE-K loop*.
MADAM [16]	Facilita o desenvolvimento de aplicações adaptativas para computação móvel, através de modelos de representação arquitecturais durante o tempo de execução, permitindo, assim, que os *components* genéricos de middleware se possam adaptar. Implementa o algoritmo *MAPE-K loop*.
M-Ware [27]	Middleware que almeja alinhar automaticamente a administração de um sistema automatizado de uma empresa com os seus objectivos de negócio, incluindo maximizar a utilidade e políticas de negócio. Implementa o algoritmo *MAPE-K loop*.

Realizada a descrição dos diversos sistemas, Salehie and Tahvildari [35] demonstram as propriedades *Self-** suportadas pelos sistemas na seguinte tabela.

Tabela 3.3: Propriedades *Self-** implementadas pelos sistemas: ✓ Implementa, - Não implementa.

Propriedades / Sistemas	Self-Configuring	Self-Healing	Self-Optimizing	Self-Protecting
Oceano	✓	-	✓	-
Rainbow	✓	✓	✓	-
MADAM	✓	-	✓	-
M-Ware	✓	-	✓	-

A partir da observação e análise da Tabela 3.3, verifica-se que todos os sistemas suportam as propriedades *Self-Configuring* e *Self-Optimizing*, o que significa que os sistemas, de acordo com a Tabela 3.1 da Subsecção 3.3.2, estão aptos a cumprirem objectivos de adaptação como *maintability*, *functionality*, *portability*, *usability*, e *performance*. Todavia, o sistema Rainbow implementa também a propriedade *Self-Healing* juntando, portanto, a *reliability* ao leque de atributos de qualidade que pode implementar.

Por outro lado, nenhum dos sistema presentes têm como propriedade *Self-Protecting*, o que, numa primeira análise, poderia ser um ponto negativo a ter em conta. Porém, de acordo com os atributos de qualidade (*functionability* e *reliability*) a ela subjacentes, verifica-se que a combinação entre *Self-Healing* com uma das restantes pode executar o que *Self-Protecting* propõe.

Partindo desta afirmação, podemos comprovar que o sistema Rainbow consegue aplicar todos os objectivos de adaptação propostos anteriormente.

Por último, Villegas et al. [39] complementam as tabelas anteriores, apresentando outras características dos sistemas estudados como a entrada de referências, outputs medidos, acções de controlo e estrutura do sistema. A Tabela 3.4 pretende demonstrar o que foi enunciado. Entrada de referências são um conjunto de valores e tipos. De acordo com mudanças nas condições de execução do sistema, este conjunto especifica o estado a ser alcançado e mantido no sistema gerido pelo mecanismo de adaptação correspondente. Nos sistemas recolhidos, referências de entrada são especificadas como forma de contrato (SLA - Service Level Agreement, SLO - *Service Level Objective* e QoS - *Quality of Service*) e acções políticas.

Tabela 3.4: Comparação de outras características de sistemas *Self-Adaptive*

Sistema	Entrada de Referências	Outputs Medidos	Acções de Controlo	Estrutura do Sistema
Oceano	SLAs	SLOs/Propriedades lógicas de elementos computacionais	Operações Discretas que afectam computacionalmente a infraestrutura	*Adaptive control*/Estrutura modificável
Rainbow	SLAs-SLOs-QoS	SLOs/Propriedades lógicas de elementos computacionais	Operações Discretas que afectam a gestão da arquitectura de software do sistema	*Adaptive control*/Estrutura modificável
MADAM	SLAs-SLOs-QoS	SLOs/Propriedades lógicas de elementos computacionais	Operações Discretas que afetam a gestão da arquitectura de software do sistema	*Adaptive control*/Estrutura modificável
M-Ware	QoS; acções políticas	SLOs/Propriedades lógicas de elementos computacionais, contexto externo	Operações Discretas que afectam a gestão da arquitectura de software do sistema	Adaptive control/Estrutura modificável

SLAs são contratos estipulados entre fornecedores e clientes [34]. A partir daí, há uma partilha de recursos entre as partes, de forma a garantir aos clientes qualidade de serviços. Bons níveis de *reliability*, *response time* e *throughput* são exemplos de requisitos a ter em conta, de forma a respeitar o acordo [7] [1]. SLO especifica os níveis de qualidade de serviço, daí que, tipicamente, SLA, SLO e QoS sejam conceitos que se complementam [8]. Relativamente ao sistema M-Ware, acções políticas pretendem descrever princípios ou estratégias para um plano de acção concebido para alcançar um conjunto de objectivos identificados pelos gestores do sistema [28]. Caso tais princípios sejam violados, as acções politicas podem ser correlacionadas com o modelo de tomada de decisão empregada por agentes lógicos baseados em acção [25].

Outputs medidos correspondem a um conjunto de valores que são calculados no sistema gerido. Urge comparar estas medidas com as referências de entrada com o intuito de avaliar se o estado desejado foi alcançado. Especificamente, outputs medem as propriedades lógicas de elementos computacionais (ex. tempo de processamento de pedidos, no que toca a software e, carga do CPU - *Central Processing Unit* no que atinente a hardware) e condições externas de contexto (ex. mudanças de ambiente).

Acções de controlo caracterizam-se em contextos de *MAPE-K loop* que afectam

o sistema gerido para ter o efeito desejado. Desta forma, acções de controlo podem ser operações discretas que afectam a infraestrutura de execução computacional do sistema gerido (ex. modificar o processo de escalonamento no CPU) e podem ser operações discretas que afectam a arquitectura do sistema gerido (ex. operações de reconfiguração da arquitectura do sistema gerido).

Tipicamente, a estrutura de um sistema que se auto-adapta está sub-dividida em dois sistemas, um que controla e outro que é gerido. A análise da estrutura dos dois sub-sistemas justifica-se pela necessidade de aferir se uma determinada abordagem implementa o controlador de adaptação integrado com o sistema gerido. Visa ainda averiguar o efeito da separação de responsabilidades nestes dois sub-sistemas na concretização dos objectivos de adaptação. No que diz respeito à estrutura do controlador, verifica-se que os sistemas recolhidos se baseiam num paradigma de controlo adaptativo, ou seja, todos seguem uma estrutura *MAPE-K loop*. Para a estrutura do sistema alvo, a opção identificada é uma estrutura modificável, tendo capacidades de reflexão(permite examinar e modificar a estrutura e comportamento de um objecto em tempo de execução). A titulo exemplificativo, um sistema que consegue reconfigurar os seus *components* arquitectónicos, possui esta característica.

Em suma, sistemas *Self-Adaptive* têm um grande potencial, pois, de acordo com as suas propriedades, contribuem com qualidade na função e objectivo de um sistema alvo. Verificou-se, ainda, que o sistema *Rainbow* é o sistema mais flexível, pois actua de acordo com as propriedades de um sistema alvo. Os outros sistemas, focam-se em áreas especificas. No entanto, a forma como reagem às suas necessidades é semelhante à do Rainbow, tendo sempre como meta alcançar os objectivos de adaptação definidos previamente.

3.4 Ferramentas de *Benchmarking* de *web servers*

Depois de analisadas as diferentes abordagens *Self-Adaptive*, revela-se pertinente identificar a ferramenta de *Benchmarking* de *web servers* que mais apta a avaliar um sistema, uma vez ser necessário medir os seus objectivos de qualidade, bem como a utilização de CPU e memória.

Actualmente, os sistemas de informação são suportados por uma configuração muito mais complexa, em comparação com épocas mais remotas, uma vez que o fluxo de

informação transaccionado tem vindo sempre a aumentar. Assim, um sistema recorre a um *Load Balancer*, de forma a distribuir a informação que recebe dos clientes pelos *web servers* auxiliares, com o objectivo de reduzir a carga de trabalho do mesmo. Desta modo, testes de *Benchmarking* permitem avaliar se um sistema consegue responder, com qualidade e com um tempo de resposta aceitável (< 2 segundo), aos seus clientes, mesmo quando o fluxo de informação requisitado é elevado.

Além disso, é fundamental que as ferramentas possuam mecanismos de medição de utilização de CPU e memória por processo, uma vez que essas métricas são responsáveis por validar o propósito desta tese. Sendo a satisfação dos clientes um dos focos principais de todas as empresas do mundo, houve, em tempos, uma certa dificuldade em saber se os mesmos estavam satisfeitos antes disso ser notado nas subida ou descida das vendas. Houve quem sugerisse fazer inquéritos de satisfação, mas, com o fluxo temporal, as empresas adoptaram técnicas de benchmarking, por estas se afigurarem mais rentáveis.

Benchmarking proporciona uma análise de processos, cuja informação é relevante na decisão de ocorrência ou não de mudanças no sistema. Dito de outro modo, detecta possíveis inconvenientes que surjam no sistema, de forma a corrigi-los e a melhorá-los.

Apesar do propósito desta tese ser, numa primeira fase, a análise de *overhead* em sistemas *Self-Adaptive*, não será provido de utilidade ter um sistema muito poupado na utilização dos seus recursos, se o mesmo não responder a atributos de qualidade como, *reliability*, *availability* e *performance*. Daí decorrer a necessidade de testar o sistema, de modo a analisar o seu comportamento, sob influência de carga e stress.

Face ao exposto, a tabela que que se segue elenca as ferramentas de *Benchmark*, *open source*, mais populares. Neste âmbito, foram equacionados documentação e artigos de apoio, na escolha das ferramentas.

JMeter é uma ferramenta desenvolvida em Java, cuja função é testar a carga do comportamento funcional e medir o desempenho de *web servers* [9] [26]. Trata-se de uma ferramenta altamente extensível pois suporta muitas funcionalidades e plugins (como o PerfMon [2]). Além disso, reporta dados relativos a *response time* e a *throughput* que podem ser exportados para ficheiros .csv, assim como graficamente.

PerfMon é um plugin que através de um ou mais agentes que se encontram à escuta numa porta específica, comunicam com o *JMeter* por protocolo TCP(*Transmission Control Protocol*), de forma a poderem monitorizar dados como o desempenho do CPU

Tabela 3.5: Lista de Ferramentas de *Benchmarking* de *Web Servers*

Ferramentas Seleccionadas	Apache JMeter [20]	The Grinder [33]	FunkLoad [13]
Linguagem de Programação	Java	Python ou Jython	Python
Análise de Dados	Análise gráfica	Motor de estatística	Análise gráfica
Testes Distribuídos	Suporta	Suporta	Suporta
Extensibilidade	Altamente extensível	Altamente extensível	Altamente extensível
Plataforma	Todas	Todas	Todas
Protocolos	HTTP, HTTPS, FTP, LDAP, SOAP, POP3, TCP	HTTP, HTTPS, FTP, LDAP, SOAP, POP3, TCP, XML-RPC, IIOP, RMI/IIOP, RMI/JRMP, JMS, SMTP	HTTP, HTTPS

e Memória. Os resultados obtidos podem ser exportados para ficheiros .csv, bem como graficamente.

The Grinder é outra ferramenta, cuja função é idêntica à do *JMeter*. Contudo, a configuração dos testes é feita através de *scripting*. De forma a obter a informação graficamente, *The Grinder* possui uma ferramenta que exporta a mesma, o *GrinderAnalyzer tool* [5] [33].

Por último, resta referir ***FunkLoad*** que é uma ferramenta escrita em python que reporta os resultados em HTML(*HyperText Markup Language*) ou PDF(*Portable Document Format*), contendo a configuração de teste, estatísticas e gráficos dos testes. Além disso, monitoriza a utilização de CPU, média de carga, memória usada e tráfego da rede sob forma de gráfico [13].

Os resultados a obter pelos testes de *Benchmarking* serão a base de suporte para os testes de *Profiling*, uma vez que podem reduzir o *overhead* dos mesmos. O facto de se saber quais os processos que têm pior desempenho em termos de CPU e memória leva a que haja uma maior simplificação dos testes de *Profiling*, evitando redundância

de informação e perda de tempo.

Na secção seguinte serão apresentados ferramentas de *Profiling*.

3.5 *Profiling*

Esta secção tem como objectivo introduzir ferramentas passíveis de identificar, ao nível do código, níveis de eficiência de um determinado software. Assim, o papel do *Profiler* é fornecer ao utilizador informação geral do programa em questão, ou simplesmente de um pedaço de código do mesmo. Mais detalhadamente, *Profiling* estima o custo de tempo de execução ou de memória durante a execução de um programa, reportando informação quantitativa do mesmo, tal como tempo de execução, estatísticas de chamadas a um respectivo método e o caminho de execução mais longo seguido [36].

Testes de *Profiling* irão dar um grande contributo a esta tese, uma vez que, a partir deles, tornar-se-á possível o cenário de identificação das fases responsáveis pela utilização excessiva de recursos computacionais do *MAPE-K loop*.

3.5.1 Ferramentas de *Profiling*

O facto dos sistemas analisados na Subsecção 3.3.4 da Secção 3.3 serem escritos em Java, para os casos de *Oceano* [15], *Rainbow* [18] e *MADAM* [16], e em C para o caso de *M-Ware* [27], conduziu a que se optasse pela análise de ferramentas de *Profiling* capazes de suportar essas linguagens.

Partindo desse propósito, verificou-se que há um vasto leque de ferramentas de *Profiling*. Todavia, aquelas que apresentam maior maturidade são, de acordo com o livro ***Java Performance*** de Hunt and John [22], ***Oracle Studio Performance Analyzer*** e ***NetBeans Profiler***.

A Tabela 3.6 contém informação relevante no sentido influenciar a escolha da ferramenta de *Profiling* a utilizar. Ambas as abordagens têm boa precisão de medição dos tempos de execução, uma vez que oferecem ao utilizador a hipótese de escolher *inclusive time* ou *exclusive time*. *Inclusive time* refere-se ao tempo consumido por um determinado método e por todos aqueles que dele derivam. *Exclusive time* apenas reporta o tempo de execução dissipado pelos métodos previamente seleccionados, ou, em caso de desejo do utilizador, por todos os métodos utilizados pelo programa. Resta

Tabela 3.6: Propriedades das Ferramentas de *Profiling* selecionadas

Profiler	Oracle Studio Performance Analyzer [30]	NetBeans Profiler [32]
Plataforma	Solaris Sparc, Solaris x86/x64, Linux x86/x64	Solaris Sparc, Solaris x86/x64, Linux x86/x64, Windows, Mac OS
Linguagens que analisa	C, C++, Fortran e Java	Java
Medição de tempo de execução	Inclusive/Exclusive	Inclusive/Exclusive
Métricas que analisa	CPU despendido, memória alocada (não inclui Java)	CPU despendido, memória alocada
Manipulação e visualização de dados	GUI, Linha de Comandos	GUI

salientar o surgimento uma nova versão, que inclui a técnica de medição *Inclusive profiler*, no *NetBeans Profiler*. Esta informação é objecto de referência no site oficial do *NetBeans* [32].

A garantia de medições de CPU e memória protagonizado pelo *NeatBeans Profiler* atribuem-lhe uma apreciação favorável, pois o *Oracle Studio Performance Analyzer* apenas garante medições de CPU.

Relativamente às linguagens a analisar, verifica-se que *Oracle Studio Performance Analyzer* tem um leque mais alargado de opções, uma vez que pode fazer *profiling* sob linguagens C, C++, Fortran e Java, o que não sucede com *NetBeans Profiler*, pois este só faz profiling a Java. *NetBeans Profiler*, por sua vez, pode ser utilizado em mais plataformas do que *Oracle Studio Performance Analyzer*, o que pode ser mais um factor a ponderar.

Por último, verifica-se que ambos oferecem estilos de manipulação de visualização de dados sob forma gráfica. Porém, *Oracle Studio Performance Analyzer* oferece ao utilizador de explorar a ferramenta através de comandos a efectuar na consola.

4

Abordagem

Tomando como ponto de partida a análise do Estado da Arte, em que foram examinados vários sistemas *Self-Adaptive* e ferramentas que os possam avaliar, este Capítulo tem como objectivo elucidar o leitor relativamente às opções a tomar.

Neste Capítulo, fundamenta-se a selecção do sistema *Self-Adaptive*, da ferramenta de *Benchmarking* de *web servers* e da ferramenta de *Profiling*. Além disso, será apresentado um caso de estudo que conduzirá à consecução do objectivo desta tese.

4.1 Selecção de Ferramentas a Utilizar

Esta secção pretende, em primeiro lugar, indicar a escolha do sistema para análise e melhoramento. Em segundo lugar, é fundamental identificar no algoritmo os blocos de código que interferem negativamente no sistema. Para isso, é necessária uma ferramenta de *profiling* que detecte, durante o tempo de execução, possíveis lacunas no algoritmo.

4.1.1 *Self-Adaptive System*

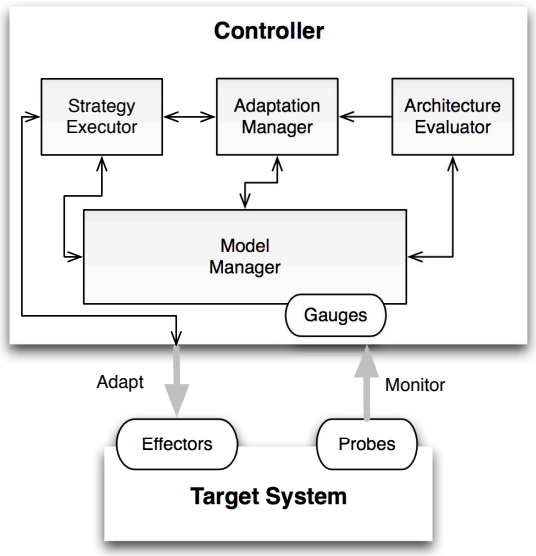
Na Secção 3.3, foram abordados os sistemas *Self-Adaptive* que apresentavam maior grau de maturidade, relativo às propriedades *Self-**, aos objectivos de adaptação que suportam e a outras características como a qualidade de serviço entre clientes e sistema.

Dos quatro sistemas estudados, nomeadamente *Oceano*, *Rainbow*, *MADAM* e *M-Ware*, aquele que mais se destacou foi o *Rainbow*, uma vez que é aquele que sustém mais propriedades *Self-** e, consequentemente, aquele que mais objectivos de adaptação pode implementar. Além disso, é o único sistema que nos permite ter acesso ao seu

código e é aquele que apresenta maior flexibilidade, pois pode operar dentro de vários contextos, enquanto que os outros optam por uma área especifica.

As Tabelas 3.1, 3.3 e 3.4 ilustram a premissa enunciada. É, portanto, inequívoco que o sistema escolhido como objecto de estudo é o sistema *Rainbow*.

Figura 4.1: *Raibow Framework*



A Figura 4.1 ilustra o a *framework* do *Rainbow* baseada no *MAPE-K loop*. Fazendo a correspondência verificamos o seguinte:

- *Monitor* - reúne, durante o tempo de execução, informação do sistema alvo através das *Probes* e das *Gauges* que actualizam as propriedades no modelo de gestão arquitectural pelo *Model Manager*;
- *Analyze* - *Architecture Evaluator* é responsável por determinar, a partir do modelo arquitectural, se uma adaptação deve ser activada para alcançar os desejados valores de qualidade;
- *Plan* - *Adaptation Manager* selecciona uma estratégia de adaptação para lidar com o desvio do comportamento correcto;
- *Execute* - *Strategy Executor* é responsável por aplicar, no *Target System* (sistema alvo), através dos *Effectors*, uma sequência de acções definidas pela estratégia de adaptação;

- *Knowledge* - a informação é partilhada pelas diferentes fases de adaptação. Nesta *framework Knowledge* corresponde ao *Model Manager* que mantém o modelo de arquitectura do sistema, durante o tempo de execução.

O sistema alvo será objecto de definição na Secção 4.2.

4.1.2 *Web-Server Benchmark*

Fazendo uma análise das ferramentas recolhidas na Secção 3.4, verifica-se que *Apache JMeter*, *The Grinder* e *FunkLoad* são ferramentas semelhantes no que toca tanto à sua função como à informação que se dispõem a oferecer. Todos eles possuem meios que reportam a informação sob forma de relatório, bem como graficamente. No entanto, em termos de usabilidade, *JMeter* é a ferramenta que mais se destaca.

Relativamente aos relatórios recebidos pelas ferramentas apresentadas, verifica-se que o facto de o *The Grinder* e o *FunkLoad* não os exportarem sob formato .csv(Comma-separated values) torna a manipulação do dados mais difícil.

Por último, o facto do JMeter ser uma ferramenta bastante testada e usada pela comunidade demonstra que esta oferece mais confiança ao utilizador na sua aplicação. Deste modo, *JMeter* é a ferramenta mais viável para fazer testes de *Benchmark*.

4.1.3 *Profiling*

O facto do sistema *Self-Adaptive* escolhido ser escrito em Java não significa que se possa fazer uma escolha evidente da ferramenta de *Profiling*. Na Tabela 3.6 verifica-se que ambas as ferramentas apresentadas analisam a linguagem Java. Como tal, neste caso, esse não pode ser o factor de desempate.

Ao analisar-se, mais detalhadamente, as características das ferramentas, constata-se que *Oracle Studio Performance Analyzer* oferece mais rigor e precisão na medição de tempos de execução do que *NetBeans Profiler*. No entanto, o mesmo não consegue avaliar a memória alocada aquando do tempo de execução, algo que o *NetBeans Profiler* suporta.

A solução para este dilema passa por estabelecer o que é prioritário, se a medição dos tempos de execução do método chamado e todos os que ele chama, ou se a recolha de informação relativa à memória alocada num determinado bloco de código. Visto que se pretende reduzir os recursos computacionais onde o algoritmo está a ser executado,

os dados acerca da memória alocada têm um grau de importância de salutar. Daí que, a escolha lógica para este dilema seja *NetBeans Profiler*.

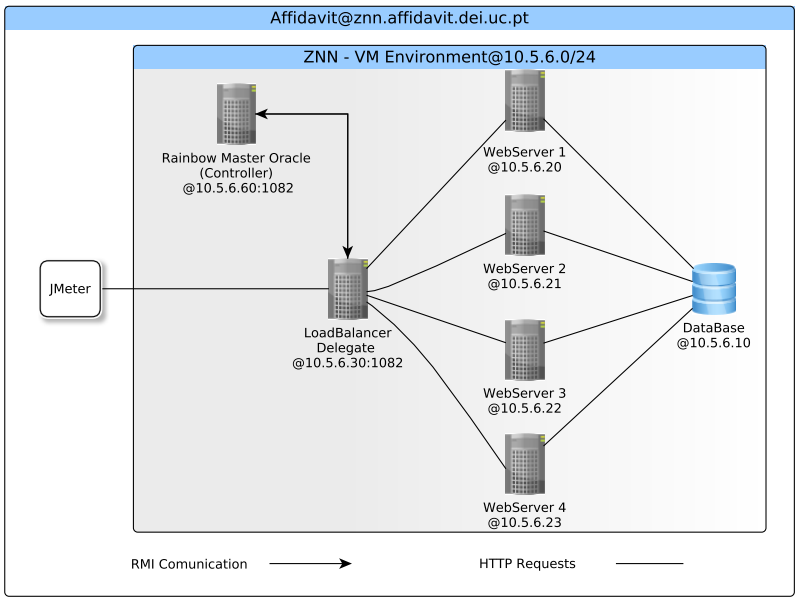
4.2 Caso de Estudo

Para que se tornasse possível testar um sistema com exequibilidade para realizar mecanismos de adaptação, seria necessário que o mesmo tivesse condições para fazer várias pesquisas requisitadas pelos diversos clientes. Daí que se tivesse optado por adoptar um sistema gerador de notícias que responda a vários pedidos de vários clientes.

Znn.com é uma infraestrutura que simula um site de notícias, disponibilizando informação estática, como conteúdos em formato de texto e conteúdos multimédia.

Tendo em conta que a motivação desta tese foi baseada no artigo "Evaluating the effectiveness of the Rainbow self-adaptive system" de Cheng et al. [11], faz todo o sentido que o sistema alvo seja esta infraestrutura.

Figura 4.2: *Representação do Sistema em estudo*



A Figura 4.2 apresenta, sob forma de diagrama, o sistema em estudo - a máquina "Affidavit@znn.affidavit.dei.uc.pt". Como se pode observar, a unidade de controlo de adaptação (*Rainbow Controller*) foi configurada numa máquina virtual distinta para

assegurar que a recolha dos valores dos recursos usados não fossem afectados por outros processos ou tarefas. Além disso, Rainbow requer a configuração de um outro *component* (*Delegate*), com o objectivo de implementar as *Probes* e os *Effectors* no sistema alvo. *Rainbow Controller* e *Delegate* comunicam através do protocolo RMI.

Este caso de estudo é composto por uma arquitectura N-tier com um conjunto de *web servers* auxiliares que servem o conteúdo referido, entre a base de dados e os clientes (Jmeter). O *Load Balancer* tem a função de distribuir a carga pelos *web servers* disponíveis. O número de *web servers* activos vai depender das estratégias de adaptação adoptadas responsáveis por cumprir os objectivos de adaptação. A interacção entre estes *components* é feita através de HTTP *requests*.

4.2.1 Especificação de *Hardware*

O sistema contém uma máquina física alojada no DEI(Departamento de Engenharia Informática), que irá comunicar com as diversas máquinas virtuais por SSH(*Secure Shell*). Assim, começando pela máquina física, verifica-se que esta contém as seguintes características:

- CPU com 8 cores - (Intel(R) Xeon(r)) E2620 @2.40GHz;
- Memória - 11,8 GB;
- Disco - Sata (ICH10 FAMILY) 220 GB;
- Sistema Operativo - Ubuntu 10.04 (lucid).

Cada máquina virtual usa um core do CPU da máquina física. As restantes características são as seguintes:

- Memória - 512 MB de memória reservada;
- Disco - SATA Controller 16 GB reservados;
- Sistema Operativo - Debian (64 bits).

4.2.2 Objectivos de Adaptação

Como a função do ZNN.com é fornecer notícias aos seus clientes, pretende-se que este os sirva com fiabilidade e com um custo operacional mínimo. Desta forma, o sistema almeja cumprir os seguintes atributos de qualidade:

- *Availability* - probabilidade de o sistema não falhar ou passar por uma reparação quando este é invocado pelo utilizador; neste caso de estudo, consideramos que uma falha ocorre quando um pedido é respondido com um incorrecto *HTTP status code*, é perdido ou demora mais de 2 segundos a ser respondido;

- *Performance* - carga actual e percentual do sistema; medida como quantidade de trabalho recebido pelo sistema sobre a carga máxima que é suportada por todos os servidores disponíveis;

- *Cost* - mede o número de máquinas virtuais necessárias em cada cenário do teste.

O *Rainbow* requer a definição de pesos em múltiplos cenários de objectivos de qualidade. A Tabela 4.1 define a relativa importância entre as dimensões de qualidade. É observado que *availability* é considerada duas vezes mais importante do que *cost* ou *performance*.

Tabela 4.1: *Rainbow Utility preferences*

	Peso
Availability	50%
Cost	25%
Performance	25%
Total	100%

4.2.3 Tácticas de Adaptação

O *Controller* pode seleccionar uma das seguintes tácticas para proceder a adaptações:

- **Activar servidor** - dentro da *pool* de servidores, se existir um servidor suplente pronto a ser activado, esta estratégia activa-o; considerando os objectivos de adaptação, esta táctica aumenta os níveis de *availability* e de performance, pois activa mais um servidor; no entanto, é prejudicial em termos de custo operacional;

- **Desactivar o servidor mais lento** - se existirem pelo menos dois servidores activos, esta abordagem desactiva o mais lento (isto é, aquele que tiver maior média de *response time*); já que um servidor é desactivado, esta táctica diminui o custo operacional e pode ter melhorias na *performance*, porque desactiva um servidor que, até à data, se encontrava a prejudicar o sistema em termos de *performance*.

- **Desactivar o servidor menos fiável** - se existirem pelo menos dois servidores activos, aquele que for menos fiável, será o eleito para desactivação; neste caso, além de se diminuir o custo operacional, aumentam-se os níveis de *performance* e de *availability*, pois é desactivado o servidor que está, ou pode estar sujeito a falhas.

4.2.4 Tipos de Adaptação

Usaram-se como tipos de adaptação os seguintes mecanismos:

- Adaptação Simples - adopta as três tácticas de adaptação apresentadas na subsecção anterior, a saber: "Activar servidor", "Desactivar o servidor mais lento" e "Desactivar o servidor menos fiável";

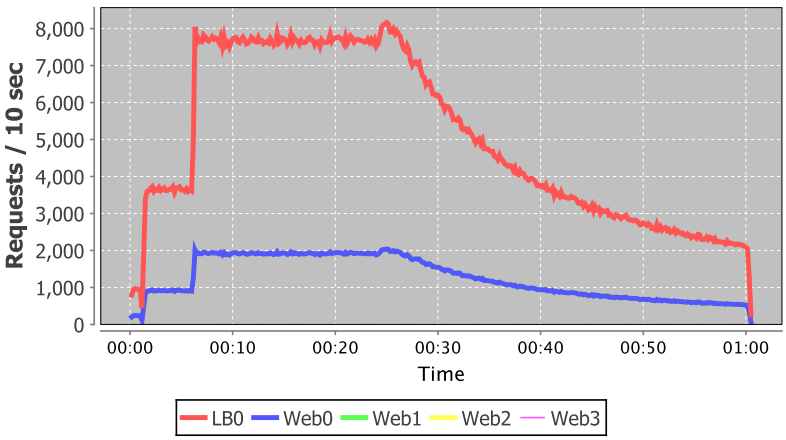
- Adaptação Complexa - além de adoptar as mesmas tácticas que a sua homóloga, consegue prever, através de modelos estocásticos, o impacto que elas têm na *availability*, promovendo uma melhor decisão de adaptação.

4.2.5 *Workload*

Idealmente, um plano que teste este tipo de sistemas deve estimular diferentes adaptações e, ao mesmo tempo, ser realista. Partindo desta premissa, é interessante verificar a ocorrência de variações de carga de informação introduzidas no sistema, começando com um nível baixo, no qual vai crescendo até chegar a um nível alto, para que, de seguida, decresça novamente de forma linear.

Slashdot effect foi um *workload* baseado numa recolha de amostras feito por mjuric e adaptado por Cheng et al. [11]. A Figura 4.3 ilustra o mesmo, cuja escala adaptada, passando de um intervalo de tempo de vinte e quatro para uma hora, sendo requisitado o total de $\simeq$ 1.5 milhões de pedidos nesse intervalo. Os *web servers* 1, 2 e 3 estão desactivados, tendo valores iguais a zero.

Figura 4.3: *Slashdot workload*



Assim, o intervalo de tempo será dividido da seguinte forma:

- 1 minuto de baixa actividade ($\simeq$ 39 pedidos/segundo);
- 5 minutos de uma aumento acentuado de tráfego na rede ($\simeq$ 152 pedidos/segundo);
- 18 minutos de um pico elevado de pedidos ($\simeq$ 313 pedidos/segundo);
- 36 minutos de diminuição linear de pedidos ($\simeq$ 168 pedidos/segundo).

4.2.6 Introdução de Falhas

Em adição ao *workload* base apresentado na subsecção anterior, será também introduzido um *delay* no minuto 10, aproximadamente. Este atraso irá afectar o funcionamento do sistema, daí ser importante saber como ele reage a tal realidade.

Espera-se, neste caso, que os resultados dos objectivos de qualidade sejam favoráveis às abordagens *Self-Adaptive* em detrimento das abordagens *Non-Adaptive*.

5

Método

Avaliar o desempenho de um sistema *Self-Adaptive* com o objectivo de identificar e distinguir problemas nas diferentes fases do *MAPE-K loop*, requer que sejam aplicados métodos e técnicas distintas. Neste Capítulo, serão apresentadas as técnicas e os métodos aplicados no presente estudo e o modo como foi implementada a análise do desempenho.

5.1 Testes de *Benchmarking* de *web servers - Jmeter*

Adoptou-se o Apache Jmeter como ferramenta de medida de stress. O *Jmeter* envia pedidos HTTP para o *Load Balancer server*, que os redirecciona para os servidores auxiliares disponíveis. Posteriormente, estes servidores encaminham a devida reposta para o *Jmeter* em que são guardados os resultados das respostas entregues com sucesso e o tempo de ocorrência da operação desde o pedido até à entrega.

Além disso, Jmeter contempla *OS Process Samples* [17] que podem ser usados para executar comandos na máquina local. Este servidor foi utilizado para executar *scripts .sh*, que recolhem informação acerca da utilização de CPU e memória nos servidores requisitados.

5.1.1 *Scripts*

Os *scripts* foram desenvolvidos para recolher informação de CPU e memória de processos relevantes tais como *httpd*, *mysql* e processos relativos às aplicações *Self-Adaptive*.

Essencialmente, estes *scripts* invocam o comando *top* do sistema *Unix* para retirar informação relevante apresentada na seguinte lista:

- *PID* - *ID* de cada processo;
- *%CPU* e *%MEM* - percentagem de utilização de CPU e memória de cada processo;
- *Command* - nome do comando que aponta para o respectivo *PID*;

O primeiro passo é recolher e guardar os *PIDs*, através do *command* requisitado, numa lista de *PIDs*. Esta lista é essencial para não se perder a informação relativa aos processos pai e filhos que podem ser gerados, por exemplo, pelos processos *httpd*. De seguida, esta informação é tratada de modo a que seja feito o somatório das percentagens de CPU e memória dos processos correspondentes aos IDs anteriormente guardados.

O facto dos *scripts* reportarem a percentagem da utilização de memória prende-se com a necessidade de não ser acrescentada complexidade na implementação nos mesmos, uma vez que cada *web server* tinha a mesma quantidade de memória disponível (512 MB), como é demonstrado na Subsecção 4.2.1.

O segundo passo é redireccionar e armazenar a informação num ficheiro csv. Este procedimento ocorre com uma granularidade de um segundo durante o tempo de execução do programa.

5.1.2 Limitações

Inicialmente, foi utilizado o plugin *Perfmon* do *Jmeter*, como, anteriormente, havia sido referenciado pelo Estado da Arte. No entanto, verificou-se, nos primeiros testes realizados, que os resultados extraídos não correspondiam aos resultados do *top* do sistema *Unix*. Isto deve-se ao facto do *Perfmon* ser uma aplicação Java, em que o seu *command* era designado por Java (assim como as do *Rainbow*), o que levava à introdução de ruído na análise realizada.

Constatada esta realidade, optou-se, então, pela realização dos *scripts* acima descritos.

5.1.3 Métricas Extraídas

Durante o tempo de execução, é gerado um log pelo *Apache HTTP Server*, o qual contém informação como o tempo de resposta de cada pedido, o *web server* responsável por responder a esse pedido, a indicação da entrega com sucesso ou não do pedido. Como é mencionado na Subsecção 4.2.2, o insucesso de uma resposta a um pedido significa que ocorreu um *HTTP status code* incorrecto, o pedido foi perdido ou a resposta demorou mais de dois segundos. Assim sendo, os dados foram calculados em janelas de dez segundos, isto é, a cada dez segundos do tempo de execução os dados eram tratados da seguinte forma:

- *Throughput* - média do número de pedidos extraídos por cada *web server* disponível (*Load Balancer* e *web servers* auxiliares activos);
- *Response Time* - média do tempo de resposta de cada *web server* disponível (*Load Balancer* e *web servers* auxiliares activos);
- *Availability* - uma relação entre pedidos respondidos com sucesso e o total de pedidos realizados, de cada *web server* disponível (*Load Balancer* e *web servers* auxiliares activos);
- *Cost* - número de *web servers* auxiliares activos.

Em adição ao que foi descrito, calculou-se, relativamente à globalidade do teste e do sistema, o total de:

- pedidos realizados;
- pedidos respondidos com sucesso;
- *Availability*;
- *Cost*.

Por outro lado, os dados relativos à percentagem de CPU e memória dos processos *httpd* do *Load Balancer* e dos processos das aplicações adaptativas (*Controller* e *Delegate*) foram retirados segundo a segundo, durante o tempo execução do teste, como é explicado na Subsecção 5.1.1. Consequentemente, cada ficheiro *csv* agrega a seguinte informação:

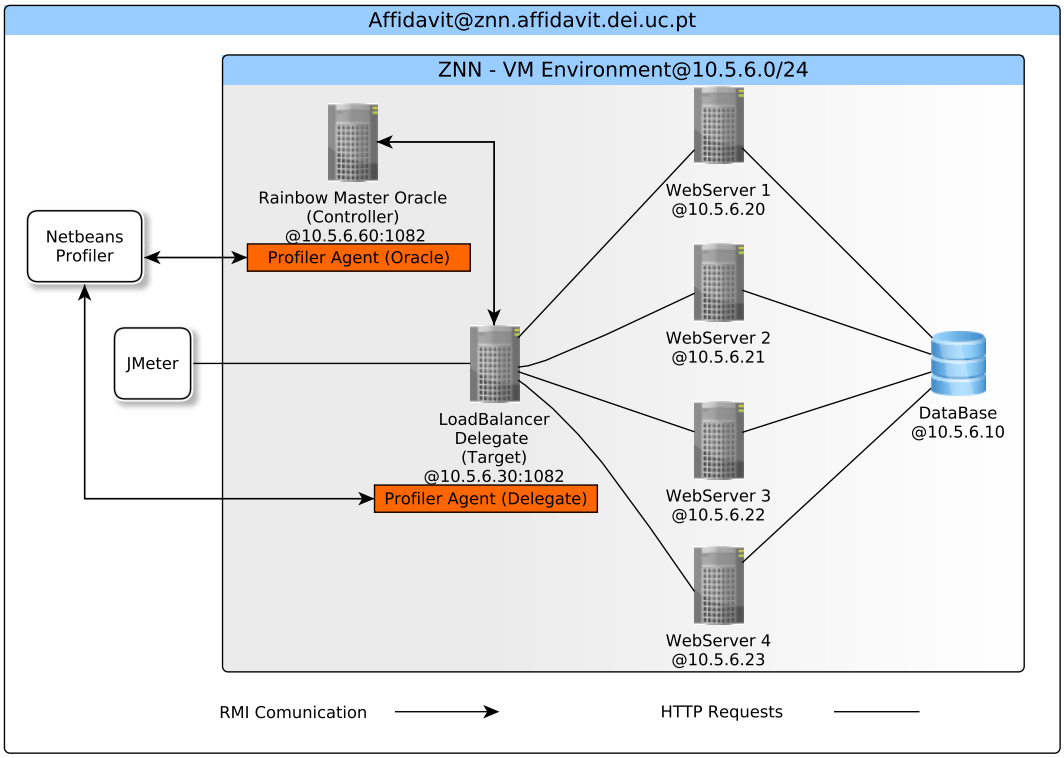
- instante em milissegundos;
- percentagem de utilização de CPU;
- percentagem de utilização de memória;

5.2 Testes de *Profiling* - *NetBeans Profiler*

Neste estudo, optou-se por uma análise dinâmica do Rainbow, de modo a identificar qual a *class*, método ou *package* responsável por consumir mais recursos computacionais.

O facto do *NetBeans Profiler* oferecer uma análise remota ao sistema veio facilitar o nosso objectivo, pois o sistema era executado em máquinas virtuais distintas.

Figura 5.1: *Representação do sistema com Profiler agents*



A Figura 5.1 mostra o funcionamento da ferramenta integrada no sistema. Como se pode observar, e tratando-se de uma análise remota, é fulcral que haja comunicação

entre as máquinas virtuais e a máquina física. Para isso, o *NetBeans Profiler* comunica através de RMI, entre o *Profiler agent* (responsável por recolher dados do processo Java que está a executar) e o *NetBeans IDE* (que recebe e trata a informação proveniente do agente).

A fim de recolhermos informação indicadora do consumo de CPU usado pelo sistema, foram aplicados os seguintes tipos de *profiling*:

- *Exclusive time* - reporta o tempo gasto, em percentagem, de todos os métodos ou classes, durante o tempo de execução;
- *Inclusive time* - reporta o tempo gasto, em percentagem, de todos os métodos que são chamados por um método específico.

No atinente à memória utilizada, foram aplicadas as seguintes opções:

- Memória - reporta a quantidade de bytes alocados por um objecto específico que foi referenciado por um método; esta é a perspectiva dinâmica da memória utilizada durante o tempo de execução;
- *Garbage Collector* - invoca o *Garbage Collector* e actualiza os resultados da alocação de memória; esta opção actualiza os resultados no preciso momento em que é operacionalizada.

5.2.1 Limitações

A grande limitação deste software é o facto do procedimento ser singular, isto é, durante a execução do sistema, só é possível medir uma métrica de cada vez. Por exemplo, ao analisarmos a utilização de CPU na aplicação *Delegate*, não é possível realizar, simultaneamente, na aplicação *Controller*.

Além disso, só é possível medir singularmente CPU e memória, o que significa que se pretendermos fazer a medição do sistema, tendo em conta as duas métricas, um determinado workload e um determinado tipo de adaptação, deverão ser efectuados seis vezes (2 (*Delegate* e *Controller*) $\times$ *Exclusive time*, 2 (*Delegate* e *Controller*) $\times$ *Inclusive time* e 2 (*Delegate* e *Controller*) $\times$ (Memória + *Garbage Collector*)).

5.2.2 Detecção do Problema

Já na posse da explicação do processo de obtenção de dados, deparamo-nos com a fase de análise dos mesmos. Os resultados de *profiler* são referentes à abordagem adaptativa complexa, sendo *Baseline* (teste sem introdução de falhas) o *workload* definido.

Elegeu-se este estudo porque, relativamente à aplicação *Delegate*, os valores dos recursos computacionais consumidos eram semelhantes entre as abordagens adaptativas, o que não se verificou, relativamente ao consumo de memória, na aplicação *Controller*, em que a abordagem complexa manifestou um aumento expressivo.

O facto do caso de estudo ser o teste *Baseline*, significa que não há diferenças significativas da abordagem complexa, no que toca à utilização dos seus recursos, quando esta é executada em diferentes tipos de teste.

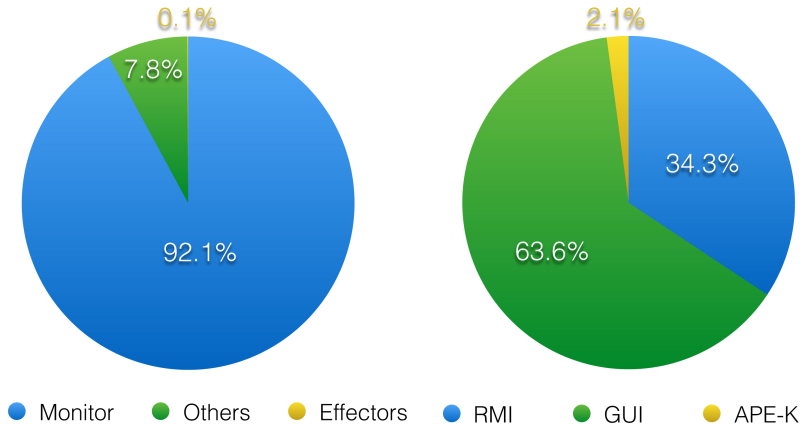
A Figura 5.2 revela que existe um consumo excessivo, quer de CPU, quer de memória, por parte de algumas fases do *MAPE-K loop*.

Como se pode observar, a figura é composta por quatro ilustrações que remetem para as análises de consumo de CPU e memória. Relativamente à legenda apresentada, verifica-se na Figura 5.2 (b) a ocorrência do termo *APE-K*, que equivale às fases *Analyze*, *Plan*, *Execute* e *Knowledge* do *MAPE-K loop*. Como o somatório do consumo de CPU destas fases era tão reduzido, optou-se por agregá-las, dando origem a este termo. A legenda referente ao *Others* remete para outras funcionalidades do *Rainbow*, como a comunicação entre as aplicações *Delegate* e *Controller* (RMI), o GUI, entre outras funcionalidades adjacentes ao *Rainbow*, mas que em nada interferem no algoritmo.

Passando à análise de resultados, há que destacar que a média de utilização de CPU por parte da aplicação *Delegate* ultrapassa os 50% e que a quantidade de memória utilizada pela aplicação *Controller* é mais de metade do que a máquina virtual tem disponível (512 MB).

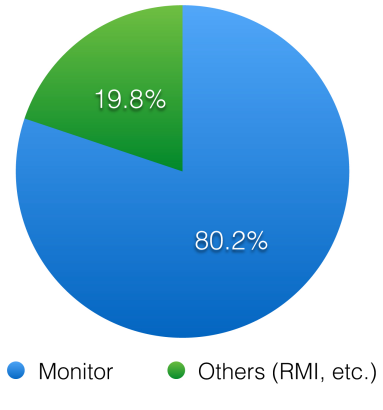
A Figura 5.2 (a) ilustra que o consumo de 91,2% de CPU dos 58.72% utilizados em média pela aplicação *Delegate* é da responsabilidade da fase *Monitor*. A causa de tal consumo, reside no modo como a informação estava a ser tratada, isto é, a cada segundo, uma string do log do *Apache HTTP Server* era percorrida de forma a ser recolhida a informação descrita na Subsecção 5.1.3. Este tratamento de strings recorre de muitas chamadas (aproximadamente 1.5 milhões de chamadas) de *substrings*, *splits*, *matchers*

Figura 5.2: Resultados obtidos pelo procedimento *Profiler*

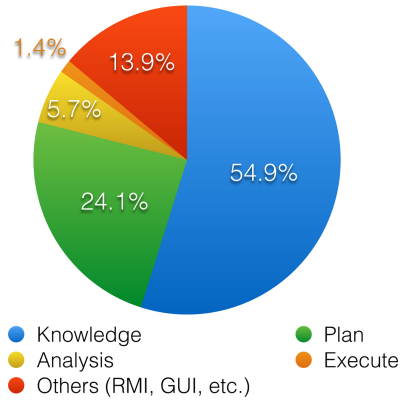


(a) Média de utilização de CPU pela aplicação *Delegate*: 58.72%

(b) Média de utilização de CPU pela aplicação *Controller*: 9.9%



(c) Média de utilização de memória pela aplicação *Delegate*: 147.03 MB



(d) Média de utilização de memória pela aplicação *Controller*: 320.69 MB

e *patterns*. Além disso, a forma utilizada para libertar a memória era a chamada explícita do *Garbage Collector* (System.gc()) por três vezes. Em cada invocação, eram percorridos todos o objectos, descartando aqueles que não estivessem referenciados. Este procedimento era executado, no total, aproximadamente 4.5 milhões de vezes, o que aumentava brutalmente o *overhead* computacional do CPU.

Por outro lado, a Figura 5.2 (d) demonstra que o consumo de 54.9% de 320.69 MB de memória utilizados pela aplicação *Controller* são da responsabilidade da fase *Knowledge*.

Sempre que o *Rainbow* tem a necessidade de reconfigurar a arquitectura do sistema alvo, faz uma cópia do mesmo, de forma a preservá-la. Todavia, no decorrer desta operação, os recursos não eram libertados, o que levou a este significativo consumo de memória.

Na subsecção seguinte, serão apresentadas as soluções para a resolução destes problemas.

5.2.3 Resolução do problema

Tendo sido demonstradas as causas de *overhead* computacional do *Rainbow*, procedeu-se à resolução das mesmas.

Relativamente à aplicação *Delegate*, visto que se tratava de um problema de optimização de código, foi contactado um especialista da área da algoritmia - o Professor Doutor Luís Paquete do Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologias da Universidade de Coimbra. Ao analisar o problema, chegámos à conclusão de que se as strings provenientes do *Apache HTTP Server* seguissem sempre a mesma lógica, seria possível reduzir a complexidade introduzida na JVM (Java Virtual Machine). Assim sendo, tornou-se possível retirar directamente da string apenas os dois primeiros dados requisitados pelo *Rainbow*, pois os restantes poderiam mudar a sua posição nas strings futuras. Por outro lado, foi aconselhado a retirar as chamadas explícitas do *Garbage Collector*, uma vez que a JVM se encarregava de executar essa tarefa automaticamente, poupando, consequentemente, o trabalho do CPU. No entanto, a memória poderia aumentar, apesar de não se tratar de um aumento significativo.

Na verdade, a documentação do System.gc() afirma que este método pode ser usado em qualquer circunstância. Porém, nos fóruns de programação é referido que ele só deve ser utilizado quando o programador tem a certeza onde e quando o deve fazer. Caso contrário, a sua invocação em nada contribui para o programa, chegando até a prejudicá-lo, como se verificou neste caso.

No caso da aplicação *Controller*, como não tínhamos acesso a essa parte do código, reportámos o problema às entidades responsáveis pelo desenvolvimento do sistema (equipa de programadores do *Rainbow* - Carnegie Mellon University), que, prontamente, enviaram uma nova versão da biblioteca.

No próximo Capítulo, serão apresentados os resultados obtidos depois da melhoria realizada ao *Rainbow*. Além disso, será feita uma análise dos mesmos, comparativamente com a que tínhamos anteriormente.

6

Resultados

De modo a validar os resultados inerentes ao melhoramento do sistema, a versão actual deve ser comparada com a anterior. Para isso, foram realizados testes que avaliam os objectivos de adaptação definidos na Subsecção 4.2.2. Pretende-se com isto, demonstrar se foram alcançados valores satisfatórios dos atributos de qualidade na abordagem melhorada. Para mais, realizaram-se experiências que analisam o consumo de recursos computacionais pelas diferentes abordagens.

A abordagem que não suporta adaptação é designada por *Non-Adaptive*, enquanto que as restantes por *Adaptive* (antes do melhoramento) e *Improved Adaptive* (depois do melhoramento).

Este Capítulo tem o objectivo de revelar os resultados obtidos.

6.1 Objectivos de Adaptação

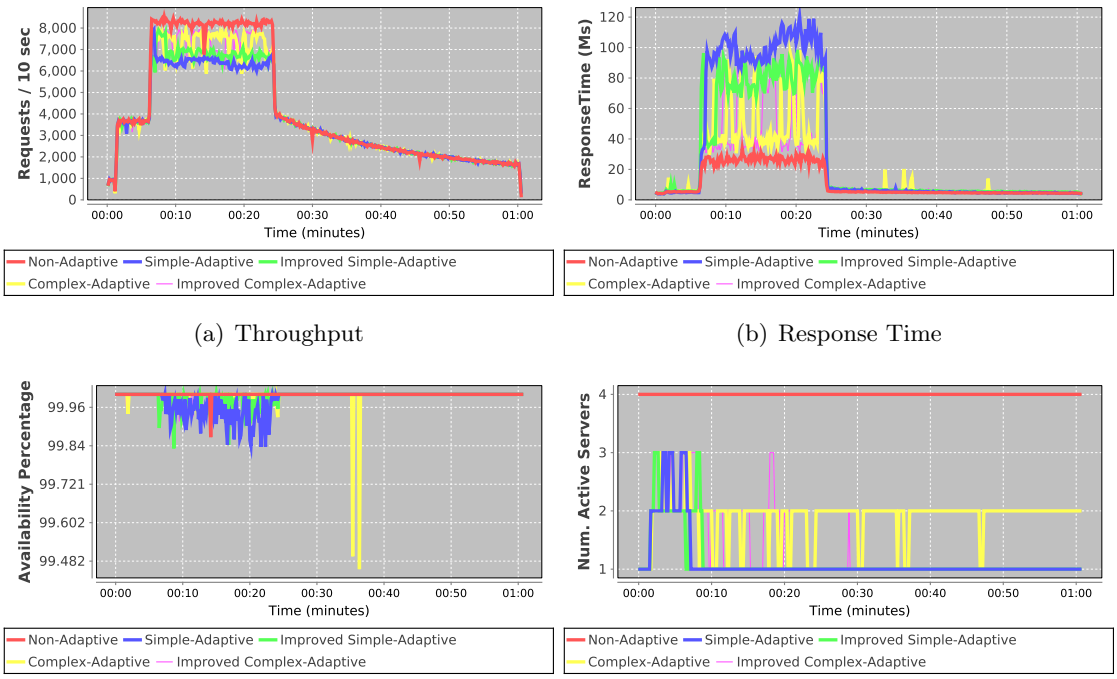
As soluções *Self-Adaptive* efectuam mudanças no sistema alvo para alcançar e manter os seus atributos de qualidade desejados. Os objectivos de adaptação têm como propósito satisfazer os atributos de qualidade referidos na Subsecção 4.2.2 e, após a melhoria do código, não devem sofrer alterações nos seus valores comparativamente com a versão anterior.

Esta secção é responsável por demonstrar os resultados das diferentes abordagens adaptativas, bem como a abordagem não adaptativa.

São, ainda, apresentados dois casos de teste. O primeiro retrata o *workload* descrito na Subsecção 4.2.5, enquanto que o segundo introduz uma falha (atraso), num dos *web*

servers auxiliares, aos 10 minutos do tempo de execução. Estes casos de teste são denominados por *Baseline* e *Delay*, respectivamente.

Figura 6.1: Resultados dos Atributos de Qualidade - Teste *Baseline*



(a) Throughput

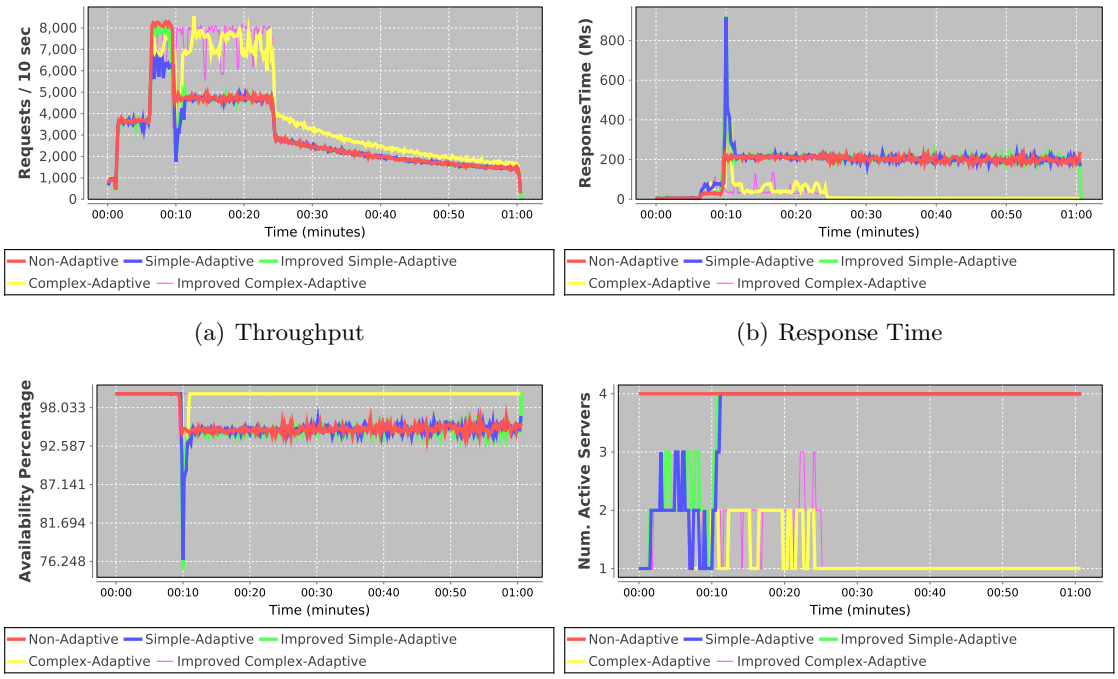
(b) Response Time

(c) Availability

(d) Número de Servidores Activos

As Figuras 6.1 e 6.2 ilustram o comportamento das diferentes abordagens, em função de *throughput*, *response time*, *availability* e *cost* (número de *web servers* activos). A cada 10 segundos, é calculada a média dos resultados obtidos.

Figura 6.2: Resultados dos Atributos de Qualidade - Teste *Delay*



(a) Throughput

(b) Response Time

(c) Availability

(d) Número de Servidores Activos

As Tabelas 6.1 e 6.2 apresentam os valores totais do que foi medido como, o número total de pedidos realizados e respondidos com sucesso, o total de *availability* e o número de servidores que foram utilizados durante o teste.

Tabela 6.1: Dados relativos aos Atributos de Qualidade - Teste *Baseline*

				Recursos Activos				
	Pedidos	**Pedidos Respondidos com sucesso**	**Availability (%)**	**LB**	**DB**	**Web Servers**	**Oracle**	**Total**
Non-Adaptive	1538524	1538515	99.99	1	1	4	0	6
Simple Adaptive	1346597	1346229	99.97	1	1	1.12	1	4.12
Improved Simple Adaptive	1387534	1387392	99.99	1	1	1.12	1	4.12
Complex Adaptive	1439647	1439576	99.99	1	1	1.89	1	4.89
Improved Complex Adaptive	1452550	1452502	99.99	1	1	1.36	1	4.36

Tabela 6.2: Dados relativos aos Atributos de Qualidade - Teste *Delay*

				Recursos Activos				
	Pedidos	**Pedidos Respondidos com sucesso**	**Availability (%)**	**LB**	**DB**	**Web Servers**	**Oracle**	**Total**
Non-Adaptive	1119625	1076588	96.15	1	1	4	0	6
Simple Adaptive	1069137	1025852	95.95	1	1	3.6	1	6.6
Improved Simple Adaptive	1093439	1047870	95.83	1	1	3.6	1	6.6
Complex Adaptive	1415223	1413422	99.87	1	1	1.26	1	4.26
Improved Complex Adaptive	1465970	1465627	99.97	1	1	1.36	1	4.36

6.2 Recursos Computacionais

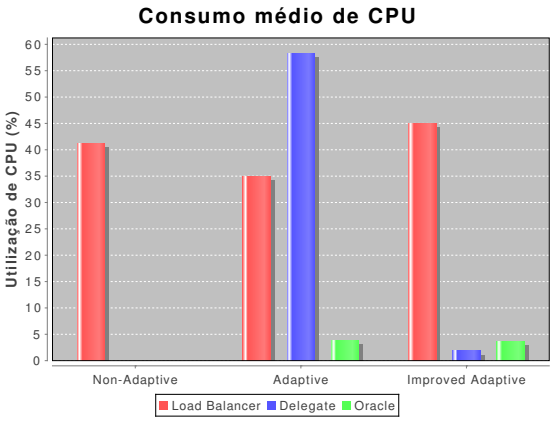
Os testes que se focaram no consumo de recursos computacionais são ilustrados nesta secção. Os dados reunidos são relativos à utilização de CPU e memória por todas as abordagens em estudo.

É medido na solução *Non-Adaptive*, apenas a intervenção do *Load Balancer*, enquanto que nas abordagens adaptativas, somam-se a ela, as medições das aplicações *Delegate* e *Oracle*. O que se pretende, é identificar se os recursos despendidos pelo *Load Balancer*, *Delegate* e *Oracle* ultrapassam a quantidade de recursos que uma máquina virtual tem disponível. Além disso, é importante validar os resultados do próprio *Load Balancer* entre as abordagens, isto é, teoricamente, o consumo computacional do *Load Balancer* não se deve alterar entre as abordagens.

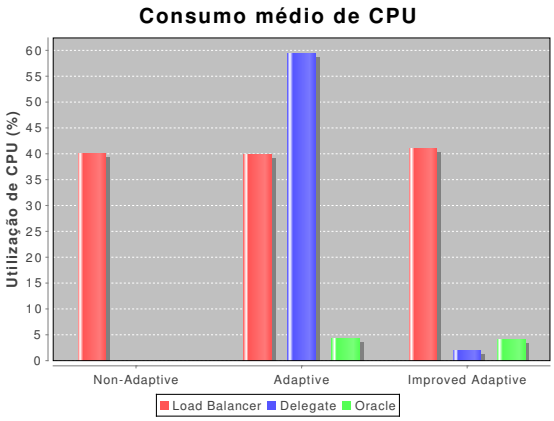
As amostras foram colhidas em tempo real, com uma granularidade de um. As Figuras 6.3 e 6.4 ilustram o consumo médio de recursos computacionais de CPU e memória, respectivamente.

As Tabelas 6.3 e 6.4 introduzem informação adicional à que foi retratada nas figuras referidas, como o desvio padrão e o percentil 95. O percentil 95 indica o valor mais elevado durante 95 % do teste. Por exemplo, o percentil 95 de CPU no *Load Balancer* da solução *Non-Adaptive* é 83.9 %, o que significa que durante 95 % do teste, a utilização de CPU é menor que 83.9 %.

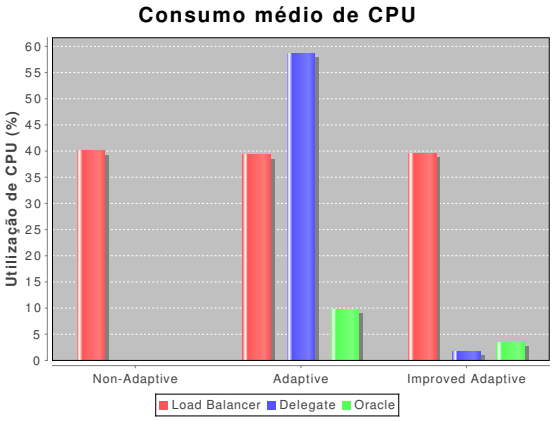
Figura 6.3: Comparação do consumo médio de CPU entre sistemas *Non-Adaptive* e *Adaptive*



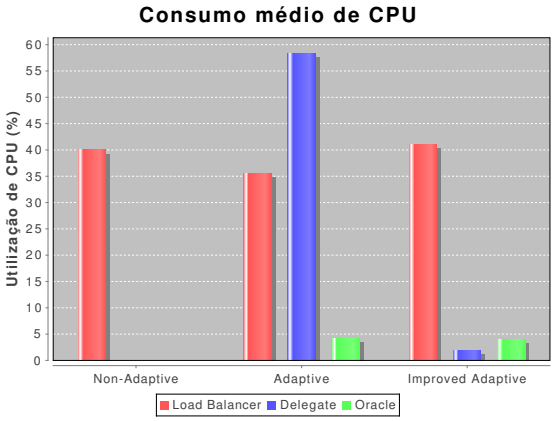
(a) Simple-Adaptive no teste Baseline



(b) Simple-Adaptive no teste Delay



(c) Complex-Adaptive no teste Baseline

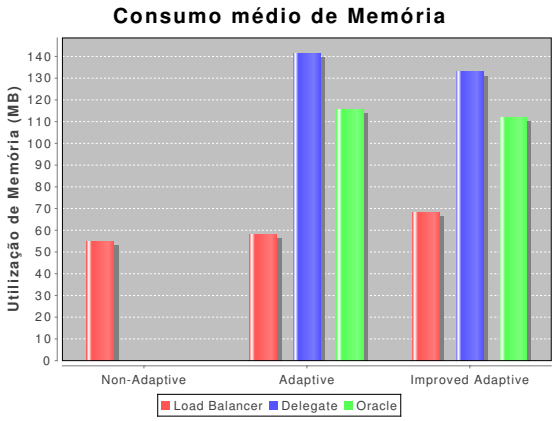


(d) Complex-Adaptive no teste Delay

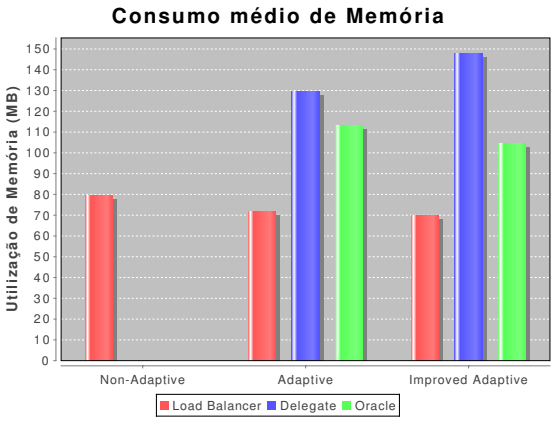
Tabela 6.3: Comparação do consumo de CPU - sistemas *Non-Adaptive* e *Adaptive* nos testes com e sem introdução de falhas

			CPU(%) - Teste Baseline			CPU(%) - Teste com Delay		
			Média	Desvio Padrão	Percentil 95	Média	Desvio Padrão	Percentil 95
Abordagem Simples	Load Balancer	Non Adaptive	41.26	22.65	96	40.08	16.54	66
		Adaptive	35.05	19.15	75.5	39.97	18.39	73
		Improved Adaptive	45.03	21.78	87.7	41.08	16.62	70
	Delegate	Adaptive	58.3	19.49	81.3	59.43	15.44	79.5
		Improved Adaptive	1.91	2.29	4	2.01	2.01	4
	Oracle	Adaptive	3.89	5.14	14	4.33	5.35	16
		Improved Adaptive	3.77	4.89	14	4.18	5.55	16
Abordagem Complexa	Load Balancer	Non Adaptive	40.12	18.55	83.9	40.08	16.54	66
		Adaptive	39.35	21.24	88	35.67	21.94	88.6
		Improved Adaptive	39.57	21.15	89.4	41.08	16.62	70
	Delegate	Adaptive	58.72	23.08	81.8	58.4	22.13	81
		Improved Adaptive	1.78	2.60	4	1.93	2.92	4
	Oracle	Adaptive	9.87	16.28	39.90	4.26	5.5	14
		Improved Adaptive	3.50	5.78	14	4.13	5.82	16

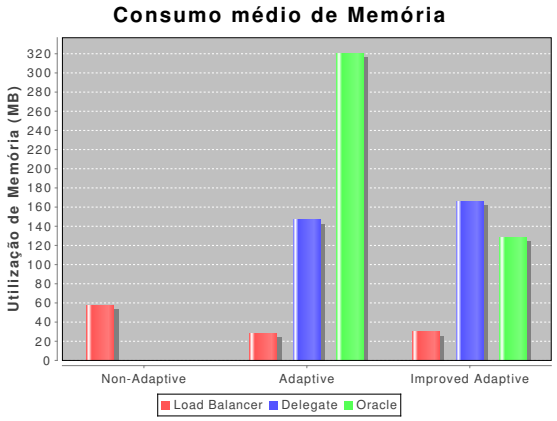
Figura 6.4: Comparação do consumo médio de memória entre sistemas *Non-Adaptive* e *Adaptive*



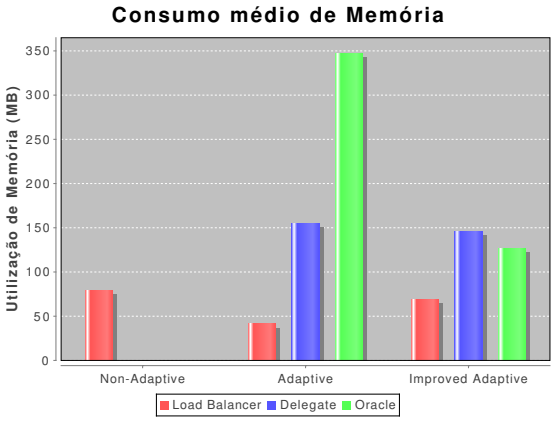
(a) Simple-Adaptive no teste Baseline



(b) Simple-Adaptive no teste Delay



(c) Complex-Adaptive no teste Baseline



(d) Complex-Adaptive no teste Delay

Tabela 6.4: Comparação do consumo de Memória - sistemas *Non-Adaptive* e *Adaptive* nos testes com e sem introdução de falhas

			Memória(MB) - Teste Baseline			Memória(MB) - Teste com Delay		
			Média	Desvio Padrão	Percentil 95	Média	Desvio Padrão	Percentil 95
Abordagem Simples	Load Balancer	Non Adaptive	54.95	13.31	61.14	79.77	1.48	80.38
		Adaptive	58.28	14.29	64.6	71.99	9.91	76.02
		Improved Adaptive	68.35	11.47	77.22	69.88	2.3	71.17
	Delegate	Adaptive	141.44	14.08	145	129.85	13.6	133
		Improved Adaptive	133.04	6.39	136	147.91	0.28	148
	Oracle	Adaptive	115.75	3.15	120	113.34	3.64	119
		Improved Adaptive	112.21	3.25	117	104.64	3.4	109
Abordagem Complexa	Load Balancer	Non Adaptive	57.9	12.99	62.97	79.77	1.48	80.38
		Adaptive	28.33	8.20	48.64	41.83	8.64	50.3
		Improved Adaptive	30.25	10.57	52.22	69.84	2.3	71.17
	Delegate	Adaptive	147.03	35.03	162	155.82	35.49	172
		Improved Adaptive	166.14	29.96	181	146.63	20.2	157
	Oracle	Adaptive	320.69	113.09	455	347.52	57.19	405
		Improved Adaptive	128.80	11.23	138	126.86	4.52	133

7

Análise das Observações

Tendo em conta os resultados obtidos no capítulo anterior, este, tem o objectivo de fazer uma análise e retirar conclusões. As Secções 7.1 e 7.2 são responsáveis por ajuizar os resultados dos objectivos de adaptação e do consumo de recursos computacionais, respectivamente.

A Secção 7.3, avalia o impacto financeiro das diferentes abordagens adaptativas numa situação real. Por último, são apresentadas as limitações deste estudo, na Secção 7.4.

7.1 Objectivos de adaptação

De acordo com o que foi apresentado no Capítulo anterior, observamos que a Figura 6.1 ilustra o comportamento das diferentes soluções, no caso de teste *Baseline*. A Figura 6.1 (a) (*throughput*) retrata o número de pedidos processados, onde se pode verificar que os valores são semelhantes entre as abordagens em estudo. Todavia, durante o período em que no número de pedidos enviados ao sistema é máximo (entre o 7 º e o 23 º minuto), ocorre um pequeno aumento de *throughput* na solução *Non-Adaptive*. Isto significa que esta abordagem, para atingir níveis de *availability* semelhantes às restantes, recorre à utilização dos quatro *web servers* auxiliares activos, enquanto que as outras abordagens apenas precisam de dois *web servers*. Este indicador revela que a solução *Non-Adaptive* tem um maior custo do que as restantes, como pode ser observado na Figura 6.1 (d).

Como mostra a Figura 6.1 (b), os valores do tempo de resposta são idênticos entre as diferentes abordagens. No entanto, ocorrem grandes oscilações nas abordagens adaptativas, durante o pico de pedidos enviados(entre o 7 º e o 23 º minuto). Isto deve-se ao facto, de nesse intervalo sucederem operações de adaptação, afectando, até, os valores de *availability*, como ilustrado na Figura 6.1 (c).

A Tabela 6.1 contém dados que complementam a análise feita à Figura 6.1. Pode-se concluir que quase todos os valores dos atributos de qualidade são semelhantes. Há, no entanto, uma diferença evidente no que toca ao número de servidores activos nas diferentes abordagens. Como descrito na Figura 4.2 da Secção 4.2, a configuração do sistema disponibiliza sete máquinas virtuais que podem ser utilizadas pelas diferentes soluções. Todavia, é requisitado que sejam utilizadas as máquinas *Load Balancer*, *Database* e pelo menos um *web server* auxiliar. No caso das soluções adaptativas, é necessária mais um máquina para alocar a aplicação *Oracle*.

De modo a alcançar os objectivos de adaptação apresentados em cima, foram necessárias 6, 4.12, 4.12, 4.89 e 4.36 máquinas virtuais pelas abordagens *Non-Adaptive*, *Simple Adaptive*, *Improved Simple Adaptive*, *Complex Adaptive* e *Improved Complex Adaptive*, respectivamente. Pode-se, ainda concluir, que as soluções melhoradas são as mais adequadas, uma vez que são capazes de alcançar os objectivos de adaptação operando com menos recursos (menos máquinas virtuais).

Relativamente ao caso de teste *Delay*, é demonstrado na Figura 6.2 (a), que os valores de *throughput* são semelhantes em todas as fases do teste, excepto na fase em que há maior volume de informação processada. Esta fase tem um factor adicional - a introdução de um atraso aos 10 minutos. O facto de, apenas, as soluções complexas conseguirem prever o impacto de cada táctica adaptativa na *availability*, permite seleccionar a que melhor comportamento pode trazer ao sistema.

O que se verifica, é que aquando da introdução de um atraso, as abordagens complexas desactivam o servidor menos fiável, ultrapassando o problema com maior eficácia, enquanto que as abordagens simples optam pela adição de servidores, mantendo o que está a falhar ligado. A Figura 6.2 (d), ilustra o enunciado, em que são activos os quatro servidores (sendo um deles com falha). Já a abordagem *Non-Adaptive*, apesar de utilizar todos os *web servers* auxiliares disponíveis, contempla um com falha, o que faz com que, os resultados não sejam satisfatórios, comparativamente com as abordagens adaptativas complexas.

As Figuras 6.2 (c) e (b), comprovam o que foi enunciado, pois somente as abordagens complexas conseguem recuperar da falha e voltam a obter bons resultados de *availability* e *response time*.

A Tabela 6.2 reforça o que foi referido acima. Como se pode observar, as abordagens simples com *delay* optam por utilizar os quatro *web servers*, o que implica ter o custo operacional (máquinas virtuais activas) mais elevado (6.6 máquinas activas), comparativamente com as outras abordagens, além de que obtém os piores resultados de *availability*. Conclui-se, ainda, que as abordagens complexas oferecem maior consistência, pois além de apresentarem os melhores níveis de *availability*, utilizam apenas 4.26 e 4.36 máquinas virtuais × hora nas soluções *Complex* e *Improved Complex*, respectivamente.

A solução *Non-Adaptive* utiliza 6 máquinas virtuais. Porém, os valores de *availability* são relativamente baixos, comparativamente com as abordagens complexas, porque a solução *Non-Adaptive* não recupera do atraso introduzido no minuto 10.

7.2 Recursos Computacionais

Passando à análise dos valores obtidos, em termos de recursos computacionais, verifica-se que na Tabela 6.3 são apresentados os resultados de utilização de CPU, antes e depois da optimização do *Rainbow*. Observa-se que, os resultados são semelhantes de teste para teste (*Baseline* e *Delay*), de tipo de adaptação para tipo de adaptação(Simples e Complexa) e de processo para processo (*Load Balancer*, *Delegate* e *Oracle*). Por exemplo, os valores médios relativos ao processo *Load Balancer* são idênticos, quando comparamos as diferentes abordagens (adapatativas e não adaptativas) com as soluções melhoradas. No entanto, constata-se de que houve uma expressiva redução de consumo de CPU na aplicação *Delegate* nas soluções melhoradas, na ordem dos 97 %, aproximadamente. Há registos de valores de desvio padrão superiores à média, devido ao facto de ocorrer uma grande oscilação de utilização de CPU durante o tempo de execução.

Em suma, as soluções melhoradas apresentam os valores mais baixos de consumo de CPU nos processos *Oracle* e *Delegate*. Quanto ao processo *Load Balancer*, é verificado que obtém valores semelhantes, comparativamente com as outras abordagens.

A Tabela 6.4 indica que o consumo médio de memória, no processo *Load Balancer*, é menos do que 70 MB, em todas as abordagens apresentadas. Além disso, no processo

Delegate, o consumo é menor do que 170 MB, nas diferentes soluções adaptativas. No entanto, houve uma grande redução de consumo, no que diz respeito à aplicação *Oracle*, nas versões melhoradas da solução complexa. De facto, comparativamente, com as versões anteriores, as melhoradas conseguiram uma redução de consumo na ordem dos 60 %.

Os valores do percentil 95, nas versões melhoradas apresentam-se relativamente próximos, sendo o mais alto (181 MB) registado no processo *Delegate* do teste *Baseline*, o que significa que em 95% do teste foram alcançados valores inferiores a 181 MB de consumo de memória.

Tendo em conta toda a análise realizada, verificou-se que a abordagem complexa melhorada é aquela que mais benefícios pode trazer ao sistema, uma vez que, alcança bons resultados nos seus objectivos de adaptação, mesmo com a presença de falhas e utilizando poucos recursos computacionais.

7.3 Impacto no Custo Operacional

Das observações enunciadas acima, usando a abordagem melhorada é possível alcançar os objectivos de qualidade desejados com um decréscimo significante de recursos computacionais. Com o *overhead* eliminado, é possível dispensar uma máquina virtual da configuração do sistema. De acordo com as especificações de hardware definidas na Subsecção 4.2.1, em que cada máquina virtual contém 512 MB dedicados de memória, e como as máquinas virtuais são replicadas, pode-se agregar o *Load Balancer* com o *Delegate* e *Oracle*, e portanto, o consumo médio de CPU seria de 47.14 % e de memória 343.33 MB.

Tabela 7.1: Comparação do custo de máquinas virtuais entre sistemas *Non-Adaptive* e *Adaptive*

	Número de Máquinas Virtuais	**Preço por ano (US$)**
Non Adaptive	6	3698.28
Adaptive	4.26	2634.24
Improved Adaptive	3.36	2066.4

A Tabela 7.1 ilustra o custo actual que cada abordagem complexa poderia ter por ano, se o sistema foi instanciado na *Amazon Web Services*. Concluindo, por ano, a solução *adaptive* representaria uma redução de 28,77 % no custo, enquanto que a solução melhorada (com a agregação de processos) alcançaria uma redução de 44.12 % no custo, o que é bastante apelativo para utilizadores deste tipo de sistema.

7.4 Limitações

As limitações detectadas ao longo da realização deste estudo são as seguintes:

- A tarefa responsável pela fase de monitorização pode incrementar o *overhead* do CPU. Neste caso de estudo, eram monitorizados quatro propriedades diferentes em *runtime* (*response time*, *throughput*, pedidos entregues com sucesso e qual servidor tratou os pedidos). Em cenários mais complexos, com o incremento de propriedades a monitorizar, o sistema pode incorrer num aumento de *overhead* dos recursos computacionais. Esta análise deve ser tida em conta para trabalho futuro.

- Custo escondido das máquinas virtuais - Cada máquina que é activada ou desactivada demora mais do que 3 segundos a começar a responder a pedidos. No entanto, nos resultados apresentados, considerou-se que uma máquina é activa se começar a responder a pedidos. Assim, o custo de activação ou de desactivação (em VMs.hora) é escondido nesses resultados, porque é pouco significativo.

8

Conclusão

Nesta tese foi avaliado o desempenho de um sistema *Self-Adaptive*, que resultou na identificação de *overhead* dos recursos computacionais em cada fase de adaptação. Além disso, foram apresentadas melhorias que garantem valores satisfatórios dos objectivos de qualidade estabelecidos.

Foi, também, avaliado neste estudo um sistema *Self-Adaptive* especifico (*Rainbow*), no qual se detectou a existência desnecessária de *overhead* computacional. Depois de investigadas quais as fases que mais recursos consumiam do algoritmo *MAPE-K loop*, optimizaram-se as fases *Monitor* e *Knowledge*. Na fase *Monitor* detectou-se um elevado consumo de CPU, que foi corrigido através de uma cuidada análise de código e contacto com especialistas na área da algoritmia. Relativamente à fase *Knowledge*, observou-se um elevado consumo de memória, cujo problema se reportou à equipa responsável pelo desenvolvimento do *Rainbow*, que prontamente disponibilizou uma nova versão do sistema, já que essa parte do código, estava inacessível. Resultou daí uma versão optimizada desta solução *Self-Adaptive*, que reduziu o custo operacional(utilizando menos máquinas virtuais) e, manteve os valores dos objectivos de qualidade pretendidos.

Para trabalho futuro, validar-se-á a proposta que resulta desta tese, ou seja, eliminando uma máquina virtual, como se comportará o *Rainbow* quer em termos dos seus objectivos, quer em termos computacionais. Além disso, analisar-se-á um sistema alvo mais complexo e diferente, de forma a comparar os resultados desta iteração com a próxima. O acréscimo de atributos de qualidade no leque de objectivos de adaptação poderá ter um impacto indesejado no que toca ao consumo computacional. Uma análise sobre este tema também ajudará a tornar o *Rainbow* num sistema ainda mais consis-

tente. Por último, investigar a utilização de recursos computacionais nas diferentes fases de adaptação, noutros sistemas *Self-Adaptive*.

Referências

[1] B. Abrahao, V. Almeida, J. Almeida, a. Zhang, D. Beyer, and F. Safai. Self-Adaptive SLA-Driven Capacity Management for Internet Services. *2006 IEEE/IFIP Network Operations and Management Symposium NOMS 2006*, pages 557–568, 2006. doi: 10.1109/NOMS.2006.1687584. 17

[2] Apache Software Foundation. Perfmon Metrics Collector. `http://jmeter-plugins.org/wiki/PerfMon/`, July 2013. 19

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004. ISSN 1545-5971. doi: 10.1109/TDSC.2004.2. 8

[4] M.R. Barbacci, R.J. Ellison, C.B. Weinstock, and W.G. Wood. *Quality Attribute Workshop Participants Handbook*. Special report. Carnegie Mellon University, Software Engineering Institute, 2000. 13

[5] Travis Bear. GrinderAnalyzer Tool. `http://track.sourceforge.net/analyzer.html`, 2007. 20

[6] Anton Beloglazov and Rajkumar Buyya. Energy Efficient Allocation of Virtual Machines in Cloud Data Centers. *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 577–578, 2010. doi: 10.1109/CCGRID.2010.45. 13

[7] P Bhoj, S Singhal, and S Chutani. SLA management in federated environments. *Computer Networks*, 35(1):5–24, January 2001. ISSN 13891286. doi: 10.1016/S1389-1286(00)00149-3. 17

[8] RH Bordini, Mehdi. Dastani, and Jürgen. Dix. *Programming multi-agent systems*, volume 4411 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-71955-7. doi: 10.1007/978-3-540-71956-4. 17

[9] J Buret and N Droze. An overview of load test tools. 2003. URL `http://clif.ow2.org/load_tools_overview.pdf`. 19

[10] George Candea. The basics of dependability. Technical report, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE, Suisse, 2007. 8

[11] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the Rainbow self-adaptive system. *20[1] S.-W. Cheng, D. Garlan, and B. Schmerl, "Evaluating the effectiveness of the Rainbow self-adaptive system," 2009 ICSE Work. Softw. Eng. Adapt. Self-Managing Syst., pp. 132–141, May 2009.09 ICSE Workshop on Software Engineering for Adaptive and Self-*, pages 132–141, May 2009. doi: 10.1109/SEAMS.2009.5069082. 1, 2, 3, 26, 30

[12] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA, 2003. ISBN 978-0-201-70372-6. 9

[13] Benoit Delbosc. Funkload Documentation. 2011. URL `https://funkload.nuxeo.org`. 20

[14] R.J. Ellison, R. J. Ellison, D.A. Fisher, D. A. Fisher, R. C. Linger, R. C. Linger, H. F. Lipson, H. F. Lipson, T. Longstaff, T. Longstaff, N.R. Mead, and N. R. Mead. Survivable network systems: An emerging discipline. Technical report, 1997. 12

[15] S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Oceano - sla based management of a computing utility. In *In Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, pages 855–868, 2001. 15, 21

[16] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, March 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.61. 15, 21

[17] Apache Software Foundation. OS Process Sampler. `http://jmeter.apache.org/usermanual/component_reference.html#OS_Process_Sampler`, July 2003. 33

REFERÊNCIAS

[18] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.175. 3, 15, 21

[19] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems — survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, January 2007. ISSN 01679236. doi: 10.1016/j.dss.2006.06.011. 10

[20] Emily Halili. *Apache JMeter*. Packt Publishing, 2008. ISBN 1847192955, 9781847192950. 20

[21] J.L. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. Self-Managing Systems: A Control Theory Foundation. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, number January, pages 441–448. IEEE, 2000. ISBN 0-7695-2308-0. doi: 10.1109/ECBS.2005.60. 11

[22] Charlie Hunt and Binu John. *Java Performance*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2011. ISBN 0137142528, 9780137142521. 21

[23] Autonomic Computing IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, (June), 2006. 1, 11, 13

[24] Mark Klein and Rick Kazman. Attribute-Based Architectural Styles. Technical Report December, CMU/SEI, Boston, MA, October 1999. 10

[25] Manbeen Kohli. An Enhanced Goal-Oriented Decision-Making Model for Self-Adaptive Systems. Master's thesis, University of Waterloo, Canada, 2011. 17

[26] J Krizanic and A Grguric. Load testing and performance monitoring tools in use with AJAX based web applications. *MIPRO*, 2010. 19

[27] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Middleware for enterprise scale data stream management using utility-driven self-adaptive information flows. *Cluster Computing*, 10(4):443–455, October 2007. ISSN 1386-7857. doi: 10.1007/s10586-007-0040-9. 15, 21

[28] Jorge Lobo, Randeep Bhatia, and Shamim Naqvi. A policy description language. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, pages 291–298, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence. ISBN 0-262-51106-1. 17

[29] Liam O'Brien, Paulo Merson, and Len Bass. Quality Attributes for Service-Oriented Architectures. In *International Workshop on Systems Development in SOA Environments (SDSOA'07: ICSE Workshops 2007)*, pages 3–3. IEEE, May 2007. ISBN 0-7695-2960-7. doi: 10.1109/SDSOA.2007.10. 12

[30] Oracle. Oracle Solaris Studio 12.2: Performance Analyzer. `http://docs.oracle.com/cd/E18659_01/html/821-1379/`, 2011. 22

[31] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999. ISSN 1094-7167. doi: 10.1109/5254.769885. 10

[32] NetBeans Org. NetBeans Profiler. `https://netbeans.org/kb/docs/java/profiler-intro.html`, 2009. 22

[33] Calum Fitzgerald Philip Aston. The Grinder. `http://grinder.sourceforge.net/`, 2000. 20

[34] Claudia Raibulet and Marco Massarelli. Managing Non-functional Aspects in SOA through SLA. *2008 19th International Conference on Database and Expert Systems Applications*, pages 701–705, September 2008. doi: 10.1109/DEXA.2008.56. 17

[35] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software. *ACM Transactions on Autonomous and Adaptive Systems*, 4 (2):1–42, May 2009. ISSN 15564665. doi: 10.1145/1516533.1516538. 13, 15, 16

[36] Björn Skoglund. Code profiling as a design tool for application specific instruction sets. Master's thesis, Linköping University, Department of Electrical Engineering, 2007. 21

[37] Ian Sommerville. *Software Engineering: (Update) (8th Edition)*. Addison Wesley, 8 edition, June 2006. ISBN 0321313798. 8, 12

[38] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009. ISBN 0470167742, 9780470167748. 10

[39] Norha M. Villegas, Hausi a. Müller, Gabriel Tamura, Laurence Duchien, and Rubby Casallas. A framework for evaluating quality-driven self-adaptive software systems. *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems - SEAMS '11*, 1:80, 2011. doi: 10.1145/1988008.1988020. 12, 15, 16

Anexos

Anexo A

Diagrama de Gantt referente ao Planeamento do trabalho realizado

As seguintes figuras representam o planeamento do trabalho realizado. A Figura A.1 refere-se ao primeiro semestre.

A. DIAGRAMA DE GANTT REFERENTE AO PLANEAMENTO DO TRABALHO REALIZADO

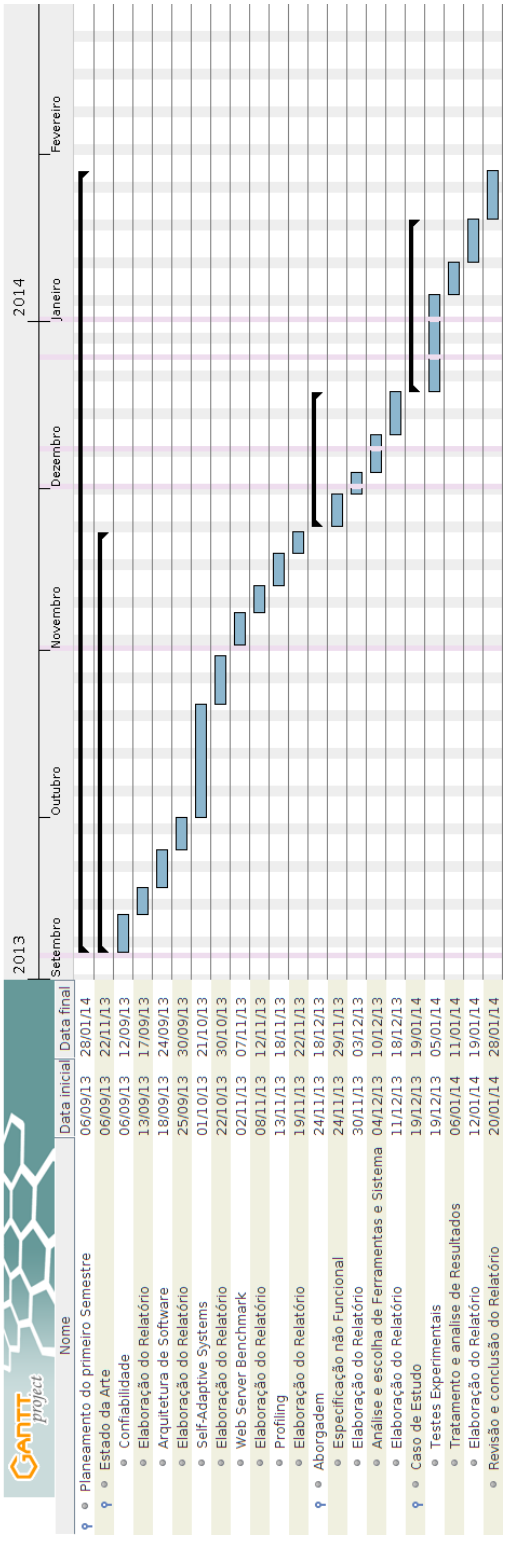


Figura A.1: *Lista de tarefas para o primeiro semestre*