Instituto Superior de Engenharia do Porto

Tolerância a Falhas em Sistemas de Tempo-Real Distribuídos e Embebidos

Óscar Emanuel de Brito Valente

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas de Tempo-Real

Orientador: Prof. Doutor Luís Miguel Pinho Nogueira

Júri:

Presidente:

Prof.ª Dr.ª Maria de Fátima Coutinho Rodrigues, ISEP

Vogais:

Prof. Dr. Jorge Manuel Neves Coelho, ISEP

Prof. Dr. Luís Miguel Pinho Nogueira, ISEP

Porto, Setembro de 2013

Resumo

Este documento descreve um modelo de tolerância a falhas para sistemas de tempo-real distribuídos. A sugestão deste modelo tem como propósito a apresentação de uma solução fiável, flexível e adaptável às necessidades dos sistemas de tempo-real distribuídos.

A tolerância a falhas é um aspeto extremamente importante na construção de sistemas de tempo-real e a sua aplicação traz inúmeros benefícios. Um *design* orientado para a tolerância a falhas contribui para um melhor desempenho do sistema através do melhoramento de aspetos chave como a segurança, a confiabilidade e a disponibilidade dos sistemas.

O trabalho desenvolvido centra-se na prevenção, deteção e tolerância a falhas de tipo lógicas (*software*) e físicas (*hardware*) e assenta numa arquitetura maioritariamente baseada no tempo, conjugada com técnicas de redundância. O modelo preocupa-se com a eficiência e os custos de execução. Para isso utilizam-se também técnicas tradicionais de tolerância a falhas, como a redundância e a migração, no sentido de não prejudicar o tempo de execução do serviço, ou seja, diminuindo o tempo de recuperação das réplicas, em caso de ocorrência de falhas. Neste trabalho são propostas heurísticas de baixa complexidade para tempo-de-execução, a fim de se determinar para onde replicar os componentes que constituem o *software* de tempo-real e de negociá-los num mecanismo de coordenação por licitações. Este trabalho adapta e estende alguns algoritmos que fornecem soluções ainda que interrompidos. Estes algoritmos são referidos em trabalhos de investigação relacionados, e são utilizados para formação de coligações entre nós coadjuvantes.

O modelo proposto colmata as falhas através de técnicas de replicação ativa, tanto virtual como física, com blocos de execução concorrentes. Tenta-se melhorar ou manter a sua qualidade produzida, praticamente sem introduzir *overhead* de informação significativo no sistema. O modelo certifica-se que as máquinas escolhidas, para as quais os agentes migrarão, melhoram iterativamente os níveis de qualidade de serviço fornecida aos componentes, em função das disponibilidades das respetivas máquinas. Caso a nova configuração de qualidade seja rentável para a qualidade geral do serviço, é feito um esforço no sentido de receber novos componentes em detrimento da qualidade dos já hospedados localmente. Os nós que cooperam na coligação maximizam o número de execuções paralelas entre componentes paralelos que compõem o serviço, com o intuito de reduzir atrasos de execução.

O desenvolvimento desta tese conduziu ao modelo proposto e aos resultados apresentados e foi genuinamente suportado por levantamentos bibliográficos de trabalhos de investigação e desenvolvimento, literaturas e preliminares matemáticos. O trabalho tem também como base uma lista de referências bibliográficas.

Palavras-chave: tolerância a falhas, sistemas de tempo-real distribuídos, formação de coligação, réplicas ativas, agentes móveis, *software* baseado em componentes, configuração dinâmica de QoS, coordenação orientada à negociação.

Abstract

This document describes a fault-tolerant model for real-time distributed systems. The proposal of this model intends to present a trustworthy, flexible and adaptable solution to meet real-time distributed systems main needs.

Fault-tolerance is an extremely important feature in real-time systems design and its implementation has countless advantages. A fault-tolerance-oriented design contributes decisively to the overall system with the improvement of key-aspects like security, reliability and systems' availability.

The developed work focuses in preventing, detecting as well as tolerating both logical (software) and physical (hardware) faults and has its basis on a majorly time-based architecture, united with redundancy techniques. It also aims at the cost-effectiveness of the execution therefore using several other traditional fault-tolerance techniques like redundancy, absent jeopardizing service execution time, and always trying to shorten replica recovery time, in faulty situations. In this work are proposed low runtime complexity heuristics to determine where to replicate components that compose the real-time software and to negotiate them in an auction-based coordination. This work makes progress on some algorithms that provide a valid solution even if they are interrupted. These algorithms are referred in related investigations works, in order to accomplish coalition formations between mutual supporting nodes.

This proposed model fills in possible gaps through virtual and physical active replication techniques, applying parallel execution blocks, in the attempt of improve or maintain the produced quality, *quasi* without creating significant overhead in the system. The proposed model ensures that the chosen machines, to which agents will migrate, improve progressively the quality of service levels provided to the components, according to the respective hosts' availabilities. It always makes an effort to receive incoming components at the cost of degrading others already hosted locally, if the new quality configuration elevates the service overall quality. The cooperating coalition nodes maximize the number of parallel executions between parallel components, in order to reduce execution delays.

This thesis development leaded to the proposed model and presented results and was genuinely supported by research and development scientific works, detailed literature survey and mathematical preliminaries. This work is also supported by a list of necessary references.

Keywords: fault-tolerance, real-time distributed systems, coalition formation, active replica, mobile agents, component-based software, dynamic QoS configuration, auction-bidding coordination.

Agradecimentos

Agradeço a todas as pessoas que me têm apoiado, desde o início deste trabalho. Agradeço acima de tudo à minha família, à minha namorada Mariana Lopes, que sempre me deu coragem para enfrentar todo o tipo de obstáculos, aos meus amigos cuja compreensão foi uma fonte de inspiração, e a Deus que me dá a força, a esperança e a alegria necessária no desenvolvimento do meu trabalho. Um agradecimento especial para o meu orientador da tese, o Professor Luís Nogueira, que em todos momentos se mostrou disponível, acompanhou o meu trabalho e me ajudou em todas dúvidas que tive.

"Science never solves a problem without creating ten more"

George Bernard Shaw

Índice

1 Introdução 1

1.1 Introdução 1

1.2 Objetivos 2

1.3 Motivação 2

1.4 Exposição do Problema 3

1.5 Estrutura 3

2 Estado de Arte 6

2.1 Conceitos e termos 6
- 2.1.1 Atributos 6
- 2.1.2 Comportamentos de sistemas em falha 7

2.2 Falhas 8
- 2.2.1 Classificação 8
- 2.2.2 Modos de falha 9

2.3 Sistemas de tempo-real 10
- 2.3.1 Ambiente Real-Time 10
- 2.3.2 Deadlines 12
- 2.3.3 Tolerância a falhas 13

2.4 Atributos da *dependability* 18
- 2.4.1 Availability 18
- 2.4.2 Reliability 19
- 2.4.3 Maintainability 20
- 2.4.4 Safety 20

2.5 Técnicas de tolerância a falhas 20
- 2.5.1 Conceitos e termos 20
- 2.5.2 Tipos de técnicas 23
- 2.5.3 Recovery Block 24
- 2.5.4 N-Version Programming 25
- 2.5.5 Consensus Recovery Block 26
- 2.5.6 Distributed Recovery Block 27
- 2.5.7 Extended Distributed Recovery Block 28

2.6 Agentes inteligentes 29
- 2.6.1 AOP 31
- 2.6.2 Tipos de arquiteturas 31
- 2.6.3 Tipos de agentes 32
- 2.6.4 Coordenação 33
- 2.6.5 Aplicação em tolerância a falhas 34
- 2.6.6 Vantagens e desvantagens 35
- 2.6.7 Trabalho relacionado 36

3 Modelo de Tolerância a Falhas 40

3.1 Descrição do ambiente 40
3.1.1 Estrutura do sistema 41
3.1.2 Configuração de QoS 43
3.2 Esquema baseado em EDRB 47
3.2.1 Replicação ativa 49
3.3 Integração de MAS 53
3.4 Formação de coligações 59
3.4.1 Coordenação auction-bidding 59
3.4.2 Algoritmos *anytime* 62
3.5 Conclusões 76
4 Implementação 80
4.1 JADE 80
4.1.1 FIPA 81
4.1.2 Arquitetura 86
4.1.3 Mobilidade de agentes 87
4.1.4 LEAP 89
4.2 Modelação 90
4.2.1 Classes 90
4.2.2 Interações entre agentes 94
4.3 Testes 101
5 Conclusão 116
5.1 Conclusão da dissertação 116
5.2 Limitações 117
5.3 Trabalho futuro 117

Lista de Figuras

Figura 2.1 - Taxonomia da *dependability* 7
Figura 2.2 – Classificação das falhas 8
Figura 2.3 – Sistema de tempo-real 10
Figura 2.4 – Ambiente de um sistema de tempo-real 11
Figura 2.5 - (a) função de utilidade de uma *hard deadline*; (b) função de utilidade de uma função *soft deadline* 12
Figura 2.6 – Espaço de estados de um sistema tolerante a falhas 13
Figura 2.7 - FTU composto por dois *fail-silent* FTUs 15
Figura 2.8 - FTU com três FCUS com *voters* 16
Figura 2.9 – Relação entre MTTF, MTTR, MTBF 19
Figura 2.10 - As quatro fases de tolerância a falhas 22
Figura 2.11 - Estrutura do esquema RB 25
Figura 2.12 – Estrutura do esquema NVP 26
Figura 2.13 - Estrutura do esquema CRB 27
Figura 2.14 - Estrutura do esquema DRB 27
Figura 2.15 - Estrutura da hierarquia do esquema EDRB 28
Figura 2.16 - Estrutura do esquema EDRB 29
Figura 2.17 - Estrutura básica de um agente 30
Figura 2.18 - Uma arquitetura de subsunção para um *robot* de navegação 32
Figura 2.19 – Etapas do *contract net protocol* 34
Figura 3.1 – Espaço de estados do modelo 41
Figura 3.2 – Composição genérica EDRB num *host* da coligação 49
Figura 3.3 - Exemplo de uma rede apta para execução cooperativa 51
Figura 3.4 – Exemplo do funcionamento de um serviço numa coligação 52
Figura 3.5 – Máquina de estados finitos de um AR 56
Figura 3.6 - Máquina de estados finitos de um AE 58
Figura 3.7 - Exemplo de serviço com componentes paralelos 66
Figura 3.8 - Exemplo de coligação com componentes paralelos 68
Figura 3.9 - Algoritmo *anytime global* QoS *optimisation* 70
Figura 3.10 - *Upgrade* no algoritmo *anytime local* QoS *optimisation* 72
Figura 3.11 - *Downgrade* no algoritmo *anytime local* QoS *optimisation* 73
Figura 3.12 – (a) *upgrade* de QoS; (b) *downgrade* de QoS 74
Figura 4.1 - Modelo referência FIPA de um AMS 83
Figura 4.2 - Estrutura de mensagem FIPA-ACL 84
Figura 4.3 - Serviço *yellow pages* 85
Figura 4.4 - Ontologia da gestão de agentes na JADE 86
Figura 4.5 - Relações entre elementos da arquitetura 87
Figura 4.6 - Interação FIPA Request entre dois agentes 88
Figura 4.7 - Modos de execução da JADE-LEAP 89
Figura 4.8 - Diagrama de classes 92

Figura 4.9 - Protocolo de interação FIPA Propose.....93
Figura 4.10 - Etapas de negociação de componentes 95
Figura 4.11 - Diagrama de sequência da inicialização e execução.....97
Figura 4.12 - Diagrama de sequência da negociação 98
Figura 4.13 - Diagrama de sequência de *downgrades*.....99
Figura 4.14 – Diagrama de sequência de *upgrades*.....101
Figura 4.15 – *Schema* XSD do ficheiro service.xml.....102
Figura 4.16 - Memória RAM com configuração de QoS simples.....106
Figura 4.17 - Memória RAM com configuração de QoS de multimédia 106
Figura 4.18 - Qualidade da coligação com configuração de QoS simples.....107
Figura 4.19 - Qualidade da coligação com configuração de QoS de multimédia 109
Figura 4.20 - Níveis de QoS nos componentes de *A* a *E*.....111
Figura 4.21 – Exemplo de *output* de cenário tradicional 112

Lista de Tabelas

Tabela 3.1 - Exemplo de lista genérica de dimensões QoS, atributos e valores possíveis 45
Tabela 3.2 - Exemplo de associação conjunto-valores da descrição de QoS 46
Tabela 3.3 - Exemplo de hierarquia de descrição QoS 47
Tabela 4.1 - Recursos de máquinas utilizadas em testes.....103
Tabela 4.2 - Tempo de migração de agentes.....104
Tabela 4.3 - Diferença de tempos de migração (original *vs* clone) 104
Tabela 4.4 - Resultados de *output* da simulação 113
Tabela 4.5 - Tempos de recuperação de *timeouts* de agentes.....114

Acrónimos e Símbolos

Lista de Acrónimos

ACC *Agent Communication Channel*

ACL *Agent Communication Language*

AE Agente de Execução

AI *Artificial Intelligence*

AID *Agent IDentifier*

AMS *Agent Management System*

AOP *Agent-Oriented Programming*

AR Agente de Reconhecimento

BDI *Beliefs, Desires, Intentions*

CBSE *Component-Based Software Engineering*

CDC *Connected Device Configuration*

CPU *Central Processing Unit*

CPS *Cyber-Physical System*

DF *Directory Facilitator*

DRB *Distributed Recovery Block*

EDRB *Extended Distributed Recovery Block*

ETA *Event-Triggered Architecture*

FCU *Fault-Containment Unit*

FIPA *Foundation for Intelligent, Physical Agents*

FIT *Failure In Time*

FTU *Fault-Tolerant Unit*

GMPLS *Generalized Multi-Protocol Label Switching*

HTTP *HyperText Transfer Protocol*

IA	*Instantaneous Assignment*
IEEE	*Institute of Electrical and Electronics Engineers*
IIOP	*Internet Inter-ORB Protocol*
IMTP	*Internal Message Transport Protocol*
J2ME	*Java 2, Micro Edition*
JADE	*Java Agent DEvelopment*
JSON	*JavaScript Object Notation*
LAN	*Local Area Network*
MIDP	*Mobile Information Device Profile*
MPLS	*Multi-Protocol Label Switching*
MR	*Multi-Robot*
MRTA	*Multi-Robot Task Allocation*
MT	*Multi-Task*
MTBF	*Mean Time Between Failure*
MTP	*Message Transport Protocol*
MTS	*Message Transport Service*
MTTF	*Mean Time To Failure*
MTTR	*Mean Time To Repair*
NGU	*Never-Give-Up*
NTP	*Network Time Protocol*
NVP	*N-Version Programming*
PCPM	*Parallel Components Processing Maximization*
PDA	*Personal Digital Assistant*
PKI	*Public-Key Infrastructure*
PRS	*Procedural Reasoning System*
QoS	*Quality of Service*

RAM	*Random Access Memory*
RB	*Recovery Block*
RMA	*Remote Monitoring Agent/Remote Management Agent*
RTS	*Real-Time System*
RTDS	*Real-Time Distributed System*
RSVP-TE	*Resource reSerVation Protocol for Traffic Engineering*
SO	Sistema Operativo
SOA	*Service-Oriented Architecture*
SR	*Single-Robot*
ST	*Single-Task*
TE	*Time-Extended assignment*
TTA	*Time-Triggered Architecture*
XML	*eXtensible Markup Language*

Lista de Símbolos

σ	Grau de significância
ω	Importância relativa

1 Introdução

1.1 Introdução

Atualmente, com a crescente exigência dos utilizadores pela qualidade de serviço proporcionada pelo *software*, são cada vez mais necessários sistemas de computação de tempo-real que respeitem os tempos de resposta impostos, para garantir níveis de qualidade aceitáveis.

Os sistemas distribuídos são altamente dinâmicos e mais complexos a nível de troca de informação do que sistemas *stand-alone*. São também mais imprevisíveis, logo é importante conceber a possibilidade de ocorrerem falhas de diversos tipos, mesmo em sistemas meticulosamente desenhados.

Hoje em dia muitos dos dispositivos utilizados são móveis e *lightweight* e por conseguinte assume-se que, em circunstâncias idênticas, a sua capacidade de processamento bem como a rapidez de resposta são bastante inferiores à de outros dispositivos, de caráter fixo, *e.g.* PC ou *laptop*. É necessário ter a capacidade de proporcionar um nível de qualidade de serviço aceitável no que toca a sistemas embebidos para tornar a experiência do utilizador satisfatória. Contudo, é sempre necessário balancear o *trade-off* entre fornecimento de qualidade e o tempo de processamento, pois uma melhor qualidade exige também uma quantidade de tempo de processamento maior.

Todos os tipos de técnicas utilizadas para tolerar falhas neste tipo de ambientes devem assegurar que o sistema fica num estado devidamente seguro de forma atempada, a fim de não sobrecarregar o sistema, dotando-o com a capacidade de tolerar falhas. As principais características dos RTDS sobre as quais o modelo proposto incide são a *availability* (disponibilidade), *feasibility* (viabilidade) e *scalability* (escalabilidade)[1].

[1] Neste documento, a fim de não recorrer ao uso de circunlóquios, simplifica-se a escrita através do emprego de estrangeirismos.

Nem sempre os sistemas conseguem cumprir as *deadlines* requeridas pelo *software*, mediante as qualidades especificadas. Um atraso de processamento num módulo de *software*, denominado componente, pode levar a subsequentes atrasos em outros módulos dependentes e, desta forma, levar à ocorrência de uma falha. No entanto, esse problema pode ser contornado. Uma aplicação de tempo-real dotada com as devidas funcionalidades, pode recorrer a outros sistemas de computação, pedindo auxílio para que executem conjuntamente um serviço de forma distribuída, coordenada, incremental e adaptativa. Os nós que compõem esse supersistema cooperam entre si, tendo em vista um propósito comum: a execução da aplicação dentro dos limites temporais.

O trabalho desenvolvido debruça-se sobre cenários em que os dispositivos envolvidos têm pouca capacidade de processamento. Os algoritmos utilizados encarregam-se de achar uma solução perto do ideal incrementalmente para que a execução obedeça às *deadlines* estabelecidas pelo sistema de tempo-real.

O trabalho descrito deu origem à publicação do artigo 'Fault-Tolerant Cooperative Service Executions in Distributed Embedded Real-time Systems' na conferência inFORUM, 2013.

1.2 Objetivos

Os principais objetivos deste trabalho são a criação de um modelo genérico tolerante a falhas para sistemas distribuídos, nomeadamente em cenários onde as máquinas têm bastantes restrições ao nível de recursos (CPU, memória RAM, etc.). Especificamente, o trabalho tem o seguinte conjunto de objetivos:

- Criação de um modelo que obedeça às propriedades de um sistema distribuído que tolere falhas de um *software* de tempo-real, em cenários onde o tempo é um ponto-chave;
- Criação de uma arquitetura para o modelo;
- Aplicação de técnicas de tolerância a falhas (prevenção, deteção e tolerância) que potencializem a recuperação do sistema e que não afetem substancialmente o tempo de execução do *software*;
- Criação de uma simulação para prova de conceito;
- Testes, análise e demonstração de resultados e conclusões acerca do modelo desenvolvido.

1.3 Motivação

Tornando-se cada vez mais dependente dos sistemas computacionais, a humanidade precisa de sistemas que tolerem falhas. A tolerância a falhas é um fator impreterível em sistemas críticos - sistemas de aviação, sistemas nucleares, elevadores, sistemas bancários, ou seja, sistemas de alto risco, onde as vidas humanas estão, de certa forma, comprome-

tidas – mas também em sistemas menos críticos, em que o tempo de resposta é igualmente uma preocupação – aplicações multimédia.

O desenvolvimento deste trabalho é então instigado pela necessidade de se produzirem sistemas de tempo-real distribuídos e embebidos, também apelidados de *Cyber-Physical Systems* (CPSs), de alta confiabilidade, ou seja, que estejam dotados de mecanismos que permitam a prossecução da execução de um serviço no sistema na presença falhas computacionais. Mesmo em situações faltosas o sistema deve continuar a produzir um nível mínimo de qualidade, podendo este até ser definido pelo utilizador. Existem várias abordagens nesta área como a redundância ou o escalonamento, embora não sejam estas que determinem o sucesso da implementação, mas sim a forma como são aplicadas.

Uma execução cooperativa tornar o funcionamento do modelo mais complexo para o *designer*, devido à formação de coligações. Por outro lado, a cooperação entre nós pode trazer grandes vantagens, em termos de resultados obtidos. É importante que o modelo a conceber esteja apto para alterar a solução da coligação formada em *runtime*, isto é, deve ser independente do *design* do sistema.

1.4 Exposição do Problema

O funcionamento de um sistema de computação distribuído difere muito de um sistema uniprocessador. A tolerância a falhas também difere na maneira como é aplicada em ambas as situações. Apesar de um RTDS se tratar de sistema complexo e descentralizado é possível encontrar algumas vantagens desses aspetos. Considerando que um sistema é de cariz distribuído, deve ser também um sistema independente internamente, em termos da autonomia dos módulos que o constituem.

Existem diversos modelos concebidos com a finalidade de contornar este problema, através da criação de réplicas passivas – método em que as réplicas começam a execução só após perceberem que a réplica primária falhou. No entanto, pretende-se capacitar o modelo proposto para que reaja imediatamente em situação de falha e que, enquadrado numa estratégia *Never-Give-Up* (NGU), forneça respostas aprazadas (*timeliness*) e, se necessário, se modifique para obter uma nova configuração global aceitável. Para isso o sistema a ser desenvolvido recorre à criação de réplicas ativas, uma primária $Prim_{c_i}$ e uma secundária Sec_{c_i}, para cada componente do *software* de tempo-real.

1.5 Estrutura

Este documento encontra-se dividido em cinco capítulos distintos, na seguinte ordem: introdução, estado de arte, modelo de tolerância a falhas em RTDS embebidos, implementação e conclusão.

Este primeiro capítulo contém a introdução do trabalho que retrata sucintamente os tópicos importantes como os objetivos do trabalho e os problemas de tolerância a falhas em sistemas de tempo-real. No segundo capítulo é abordado o estado de arte: são descritos vários conceitos teóricos de tolerância a falhas, bem como algumas das abordagens mais conhecidas, que estão na génese de técnicas derivantes destas. São também referidos requisitos para desenvolver um sistema tolerante a falhas nesta área e é feita uma análise sobre como ultrapassar as dificuldades na modelação destes sistemas. São citados alguns dos trabalhos de investigação que mais se destacam até hoje, relacionados com o tema. Sobre estes é feita uma crítica, relatando vantagens e desvantagens. No terceiro capítulo o modelo de tolerância a falhas proposto é detalhado, analisado e são mencionadas as suas vantagens e desvantagens. No quarto capítulo é descrita a simulação desenvolvida, num ambiente multiagente, para prova de conceito. Os resultados obtidos são analisados e comentados. Por fim, no último capítulo de conclusão sintetiza-se o trabalho e compara-se o que era expectável de se verificar neste modelo com o que realmente se apurou. São descritas as dificuldades encontradas ao longo da investigação e da criação e desenvolvimento. É referido algum trabalho futuro que pode ser realizado para aprimorar a qualidade do trabalho desenvolvido.

Ao longo do trabalho são mencionados outros aspetos e notas, considerados de segunda ordem, que se encontram no rodapé.

2 Estado de Arte

Este capítulo aborda alguns assuntos meramente teóricos relativamente à tolerância a falhas aplicada em RTSs e refere os principais conceitos deste tema. Debruça-se também sobre as características que definem um sistema multiagente.

2.1 Conceitos e termos

Num sistema distribuído de tempo-real existem várias questões fundamentais (*issues*) que devem ser consideradas no *design* de um sistema de tolerância a falhas aplicado a RTSs e merecem especial atenção no modelo desenvolvido.

2.1.1 Atributos

Existem diversos aspetos-chave de um sistema que representam, num todo, a confiabilidade e a segurança do sistema. A unificação destes atributos forma um conceito, comumente conhecido nesta área, denominado *dependability*. A Figura 2.1 representa a taxonomia da *dependability* e sintetiza as subáreas que a constituem, no âmbito de sistemas de tempo-real.

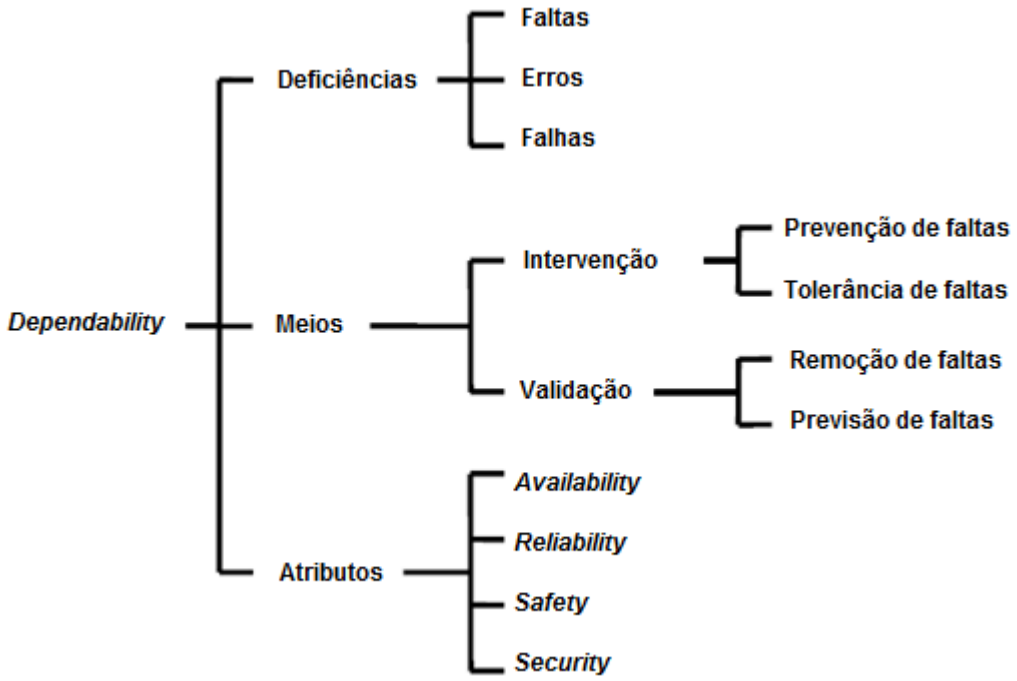


Figura 2.1 - Taxonomia da *dependability*

[1]

- *Availability* (disponibilidade) - determina a sobrecarga do sistema;
- *Feasibility* (viabilidade) - diz se a tarefa consegue cumprir a *deadline*, independentemente do tipo (*hard* ou *soft*). O não cumprimento da *deadline*, dependendo do seu tipo, torna a execução da tarefa inútil ou então o seu resultado cada vez menos útil, quanto maior o atraso;
- *Reliability* (fiabilidade) - determina a disponibilidade nos serviços ponto-a-ponto, isto é, se o sistema fornece continuamente serviço. Quanto maior a *reliability*, menor o seu impacto na experiência do utilizador;
- *Safety* (proteção) – é um aspeto que permite avaliar a capacidade do sistema evitar consequências catastróficas no ambiente em que se encontra inserido;
- *Security* (segurança) - refere-se à prevenção do acesso ou tratamento de informação não autorizado;
- *Scalability* (escalabilidade) - medida que define a capacidade do sistema lidar com maiores quantidades de dados, cujo volume tende a crescer. Com um aumento da carga de trabalho (*workload*) aumenta também o *throughput*.

2.1.2 Comportamentos de sistemas em falha

Na eventualidade de ocorrer uma falha, um determinado sistema exibirá um comportamento específico, dados as suas características e o tipo da falha. Estes fatores afetam o sistema em função da sua confiabilidade.

Os comportamentos dos sistemas podem variar entre os seguintes modos de falha:

- *Fail Stop System* – o sistema para e não produz *output*. Não recebe, nem envia mensagens, nem aciona eventos. A persistência (ver 2.2.1) destas falhas é do tipo permanente, visto que o sistema não consegue recuperar do seu estado faltoso.
- *Byzantine System* – o sistema entra num estado inconsistente, pois continua a sua atividade, mas o seu *output* é errado.
- *Fail-Fast System* – é uma mistura dos dois anteriores. O sistema inicialmente age como *Byzantine*. Porém, após um certo tempo, passa para um comportamento *Fail Stop*.

Todos estes comportamentos são erróneos, logo provocam estados que não são plausíveis no que diz respeito a sistemas de tempo-real.

2.2 Falhas

2.2.1 Classificação

Os diversos tipos de falhas referidos podem ser catalogados de acordo com perspetivas distintas. A Figura 2.2, abaixo apresentada, mostra a relação hierárquica das falhas. As áreas de maior interesse para o trabalho são a fase da falta, no ramo da origem, e ambos os tipos de persistência de faltas. Relativamente aos outros ramos, é independente da sua natureza, como do tipo de fenómeno e até do tipo de persistência.

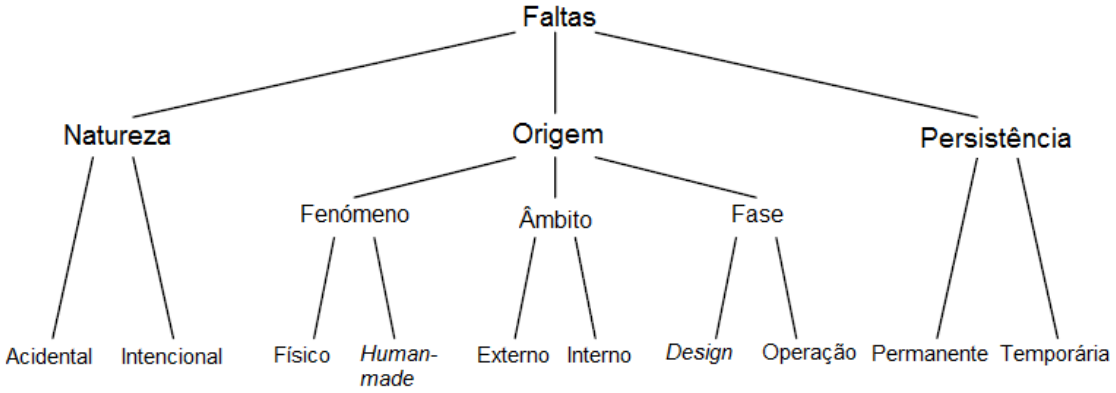


Figura 2.2 – Classificação das falhas

[1]

Existem vários tipos de falhas que provocam um comportamento errado no sistema. Kumar *et al.*, 2011, qualificam os diferentes tipos de falhas que podem ocorrer de um ponto de vista de sistemas de tempo-real:

- Falhas de rede – ocorrem devido a problemas numa partição de rede: perda ou corrupção de pacotes;

- Falhas físicas – são provocadas por falhas em CPU, memória ou disco;
- Falhas de *media* – são provocadas por *crashes* no direcionamento de *media* (dados de vídeo e de áudio);
- Falhas de processadores – falhas em processadores, devido a *crashes* nos sistemas operativos;
- Falhas de processos – são falhas que ocorrem devido a *bugs* de *software* ou escassez de recursos;
- Falhas por expiração de serviço – ocorrem quando o tempo do serviço expira enquanto está a ser consumido por uma aplicação.

Os autores abordam três tipos de falhas relativamente à sua duração e aos recursos de um sistema:

- Permanentes – falhas mais graves que ocorrem devido a falta de alimentação elétrica ou falhas em cabos;
- Temporárias:
    - *Intermittent* – falhas que surgem casualmente e que são difíceis de prever, pois normalmente só são percebidas quando o sistema está *online*;
    - *Transient* – falhas inerentes ao sistema. Devem ser solucionadas com técnicas de *rollback* e *restart*.

2.2.2 Modos de falha

Hiller, 1998, debruça-se sobre as consequências das falhas relacionando-as com a perceção que os utilizadores têm destas. O autor apelida este assunto de modos de falha (*failure mode*). De acordo com Hiller, uma falha tem sempre: um domínio, uma perceção e uma consequência.

O domínio de uma falha (*failure domain*) consiste num intervalo de valores faltosos (*value failures*) e falhas de *timing* (*timing failures*), ou seja, entregas de respostas fora de tempo, ambos não compreendidos pelo sistema.

A perceção de uma falha (*failure perception*) pode ser entendida como falha consistente (*consistent failure*), *i.e.* a falha que é percebida pelo utilizador sempre da mesma maneira, ou falha inconsistente (*inconsistent failure*), encontradas em sistemas Bizantinos, que pode ter diferentes tipos de perceções por parte do utilizador.

Relativamente às consequências das falhas (*failure consequence*), estas podem ser benignas (*benign failures*), ou seja, as suas ocorrências tornam-se transparentes para o resultado, ou catastróficas (*catastrophic failures*), isto é, falhas que são de pior magnitude, e tornam o resultado mais degradado comparativamente com o que o sistema devolveria na ausência da falha.

A relação sucessiva entre falha (*failure*), falta (*fault*) e erro (*error*) é explicitada pela cadeia fundamental de Laprie [2]:

$$\ldots \rightarrow falha \rightarrow falta \rightarrow erro \rightarrow falha \rightarrow falta \rightarrow erro \rightarrow \cdots$$

É importante referir que este ciclo é finito dado que nem todas as faltas conduzem a um erro e nem todos os erros conduzem a uma nova falha.

2.3 Sistemas de tempo-real

2.3.1 Ambiente Real-Time

Quando se se depara com um sistema de computação em tempo-real, todo o cenário envolvente torna-se diretamente dependente da linha temporal presente e a terminologia utilizada é específica.

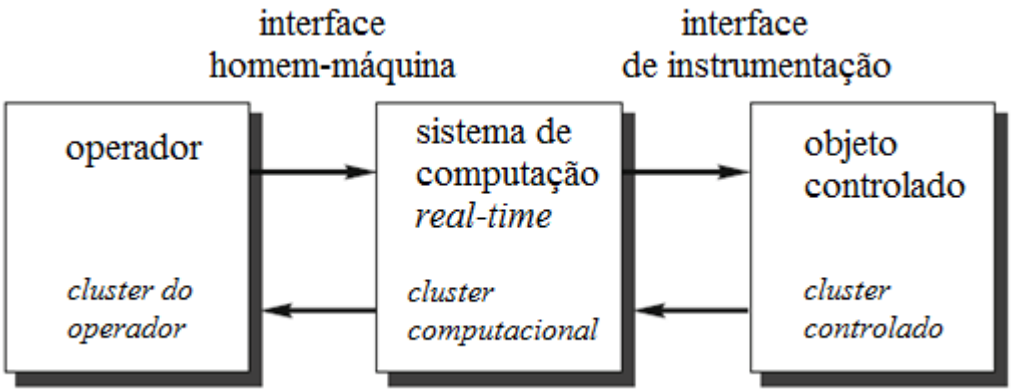


Figura 2.3 – Sistema de tempo-real

[3]

A Figura 2.3 ilustra as interações entre os intervenientes num sistema de tempo-real: o operador, o sistema de tempo-real e o objeto controlado.

Um sistema é composto por várias entidades ou *real-time entities*. Uma *real-time entity*, que consiste numa variável de estado significativa para o sistema, é composta por atributos que não tendem a alterar durante o ciclo de vida dessa mesma entidade. Uma *real-time database* (base de dados de tempo-real) é um conjunto de variáveis que definem a imagem do *cluster* controlado. Nestas variáveis está projetado o estado do sistema e estas são acedidas pelos observadores do RTS. Os observadores podem ser de um dos seguintes tipos:

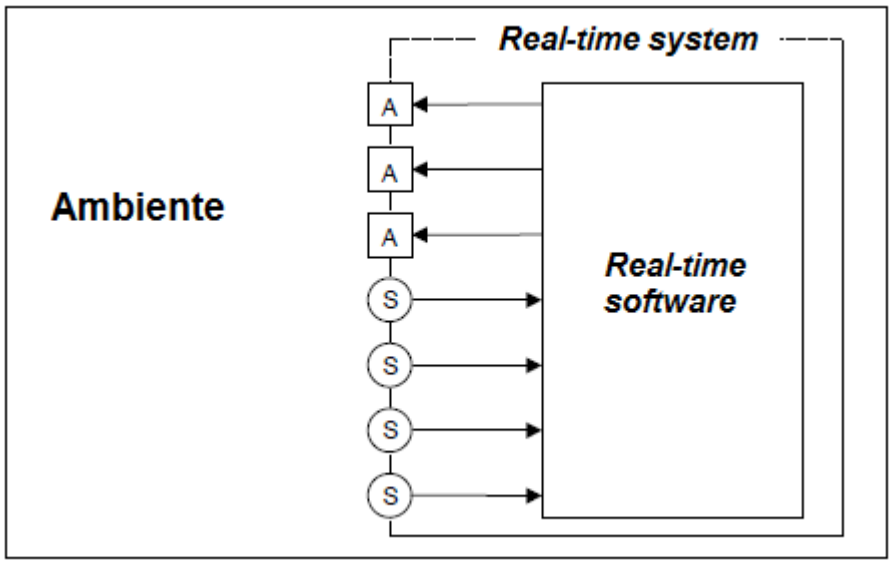
- *Event-triggered* – atualização a uma variável da *real-time database*, imediatamente após a alteração de um estado;
- *Time-triggered* – atualização a uma variável da *real-time database*, despoletado pela progressão do *real-time clock*, por um período fixo.

A equação (2.1) devolve o valor em segundos do *delay jitter*, em função da variação do *delay* Δd, de uma máquina $computer$.

$$jitter = \Delta d^{computer} \tag{2.1}$$

[3]

As interações efetuadas entre as entidades podem ser de dois tipos: leitura – um dos intervenientes é um sensor, sente o meio - ou escrita – um dos intervenientes torna-se um atuador, isto é, provoca uma alteração no meio. Estas interações são também denominadas de interfaces de instrumentação. A Figura 2.4 apresenta uma arquitetura genérica do ponto de vista de interações entre o *real-time system* e o *real-time software*.



A denota atuadores e S denota sensores

Figura 2.4 – Ambiente de um sistema de tempo-real

[3]

2.3.2 Deadlines

Num RTS a definição de valores para os prazos temporais, a que a execução deve obedecer, é uma tarefa obrigatória. Estes valores, independentemente do *design* do sistema, devem ser especificados antes da sua execução em todos os módulos que exijam uma execução em tempo-real. Sendo um fator crítico em ambientes computacionais de tempo-real, as imposições relativas ao tempo são definidas através de valores numéricos, chamados *deadlines*. Estas indicam qual o tempo máximo permitido até à obtenção ou devolução de uma resposta.

As *deadlines* também têm uma classificação própria e podem ser do tipo:

- *Hard* – as *hard deadlines* são prazos impostos, que no caso de não serem cumpridos podem ter consequências severas no meio onde estão inseridas e o resultado devolvido já não é útil. Este tipo de *deadlines* caracteriza os sistemas *Hard Real-Time*, também denominados de *safety-critical real-time computer systems*;
- *Firm* – são denominadas de *firm deadlines* os prazos das execuções cujo resultado já não é útil, se o tempo for ultrapassado. No entanto, ao contrário das *hard* não têm consequências graves para o meio;
- *Soft* – as *soft deadlines* são prazos que, se forem excedidos, a qualidade do resultado é progressivamente degradada. Este tipo de *deadlines* encontra-se presente nos sistemas *Soft Real-Time*.

A seguinte figura apresenta as funções da utilidade das *deadlines*, *soft* e *hard*, em função do atraso dos resultados.

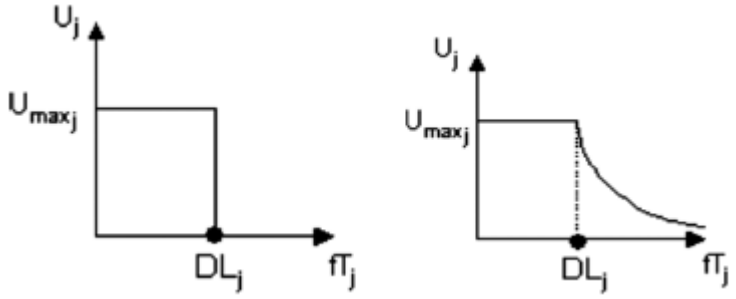


Figura 2.5 - (a) função de utilidade de uma *hard deadline*; (b) função de utilidade de uma função *soft deadline*

[4]

Como se pode ver na Figura 2.5, no caso da função das *soft deadlines* (b), após a ultrapassagem da *deadline* no eixo do tempo fT_j, a utilidade U_j tende para zero, enquanto que no caso da função das *hard deadlines* (a), a utilidade do resultado desce abruptamente para zero.

2.3.3 Tolerância a falhas

2.3.3.1 Estratégia *Never-Give-Up*

Numa perspetiva de sistemas de tempo-real, as falhas podem ser divididas em dois tipos: falhas normais e falhas raras. A Figura 2.6 mostra o espaço de estados da localização das falhas de um sistema tolerante a falhas.

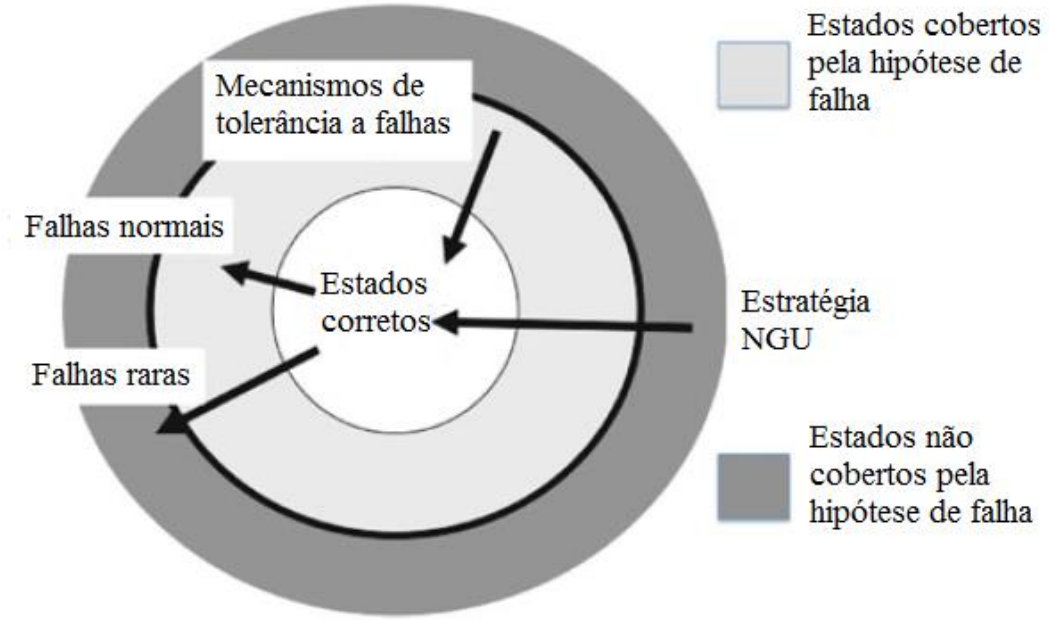


Figura 2.6 – Espaço de estados de um sistema tolerante a falhas

[3]

As falhas do tipo normal são falhas que devem ser toleradas pelo sistema e que o levam a um estado chamado *normal fault-state*. Considerando ainda a figura, contrariamente às falhas do tipo normal, as falhas do tipo raro encontram-se fora da área *fault hypothesis*. Dado que se encontram num estado não abrangido pela *fault hypothesis*, caso ocorram falhas raras, torna-se impossível controlar o sistema através do mecanismo de tolerância a falhas.

A estratégia NGU (*Never-Give-Up*) consiste numa abordagem de tolerância a falhas que tenta encaminhar perduravelmente o sistema para um estado correto. Para tal, é necessário definir quais as FCUs (*Fault-Containment Units*) existentes no sistema e criar apropriadamente as FTUs (*Fault-Tolerant Units*).

2.3.3.2 *Fault-Containment Unit* e *Fault-Tolerant Unit*

Uma FCU consiste numa unidade de falha (*unit of failure*) que faz parte do sistema de tempo-real e que é reconhecida pelo *designer*, logo constitui a *fault hypothesis*. No *design* do sistema as FCUs devem ser concebidas de forma a falhar independentemente. Caso contrário, o impacto no ambiente do RTS será desastroso, e poderá até fazer com que o sistema entre em modo de falha permanente. Caso isso suceda nem sequer é possível a aplicação da estratégia NGU posteriormente.

De forma a "mascarar" as falhas dos FCUs, são criados diversos *Fault-Tolerant Units* (FTUs) que consistem em conjuntos de FCUs. A implementação dos FTUs vai depender do tipo de implementação dos FCUs. O modo de implementação dos FTUs deve seguir as seguintes regras:

- Se o FCU for implementado numa abstração *fail-silent*, então o FCU consiste em dois FCUs;
- Se não for possível assumir nada relativamente ao comportamento do FCU, em caso de falha (falhas *Byzantine*) então é necessário proceder à implementação de quatro FCUs, ligados por dois canais de comunicação independentes para formar a *Fault-Tolerant Unit*;
- Se for possível assumir que em todas as FCUs existe um *fault-tolerant global time* e que a rede de comunicação contém falhas temporais de uma FCU, então o mascaramento de uma falha de um FCU *non-fail-silent* é exequível através de triplicação modular – TMR (*Triple Modular Redundancy*), o método mais importante de *fault-masking*.

Numa perspetiva de engenharia de *design*, no primeiro caso, um FCU *Fail-silent* consiste num subsistema computacional juntamente com um detetor de erros ou dois FCUs e um *self-checking checker*, cuja função é comparar resultados. Este último componente verifica se o FCU produz resultados corretos, no domínio de valores e de tempo, ou se não produz. Neste esquema, no caso de ser composto por dois FCUs *fail-silent*, como estes são determinísticos, ambos resultados devolvidos estarão corretos. Ainda relativamente a este caso, o número de mensagens corretas produzidas podem ser zero, no caso de falha, ou uma ou duas, no caso de sucesso. Evidentemente que se as mensagens resultantes forem idempotentes, duas mensagens replicadas terão o mesmo efeito de uma só.

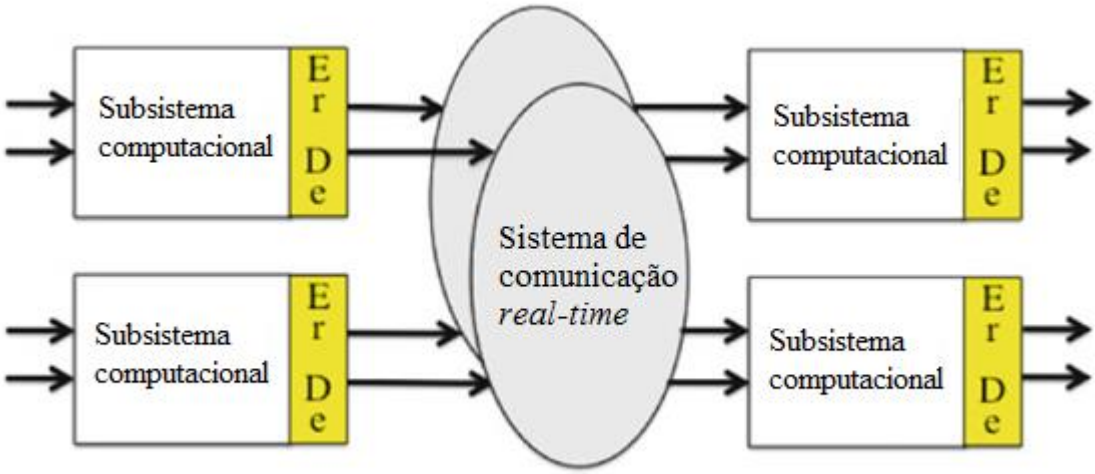


Figura 2.7 - FTU composto por dois *fail-silent* FTUs

[3]

Relativamente ao segundo ponto, a implementação denomina-se *Byzantine Resillient Fault-tolerant Unit* e consiste em quatro componentes. Este esquema de FTU é necessário para tolerar faltas do tipo *Byzantine*, também chamadas "maliciosas", e é aplicado se não for possível assumir o modo de falha e se não existir um *fault-tolerant global time* no sistema. Generalizando, segundo Kopetz, 2011, [3] um protocolo que forme um acordo *Byzantine* tem uma série de requisitos. Para tolerar falhas de k componentes são necessários os seguintes pressupostos:

- O FTU deve conter $3k + 1$ componentes;
- Cada componente deve estar ligado a todos os outros componentes do FTU por $k + 1$ caminhos de comunicação disjuntos;
- Para detetar componentes faltosos, devem ser executadas $k + 1$ "rodadas" de comunicação[2] entre componentes.

Por fim, o último tipo de implementação *Triple Modular Redundancy* permite detetar e "mascarar" valores faltosos (que se encontram fora do domínio de valores), que provavelmente não são tolerados num dado domínio aplicacional. Neste tipo de configuração, um FTU tem de consistir em três FTUs sincronizados e determinísticos. Estes, por sua vez, têm de estar ligados por dois sistemas de comunicação de tempo-real independentes. Para além disso, tem de existir um *fault-tolerant global time*, tanto nos FCUs como no sistema de comunicação e o sistema deve ter conhecimento do comportamento temporal permitido. Esta implementação também lida com uma estratégia de votos, *voting*, que serve para determinar, através de uma lógica de democracia, qual o componente faltoso. Existem duas estratégias de *voting*:

[2] Entenda-se por "rodada" de comunicação uma mensagem enviada por cada um dos componentes do sistema a todos os outros.

- *Exact voting* – neste tipo de estratégia de voto é feita a comparação dos resultados das mensagens dos 3 FCUs: se duas das três mensagens forem iguais (através do *bit pattern*), então uma dessas duas é selecionado como *output* da componente FCU;
- *Inexact voting* – neste tipo de estratégia assume-se que duas das mensagens contêm o mesmo resultado, se estes chegarem num determinado intervalo, específico à aplicação. Esta estratégia é usada se o determinismo das réplicas não for garantido.

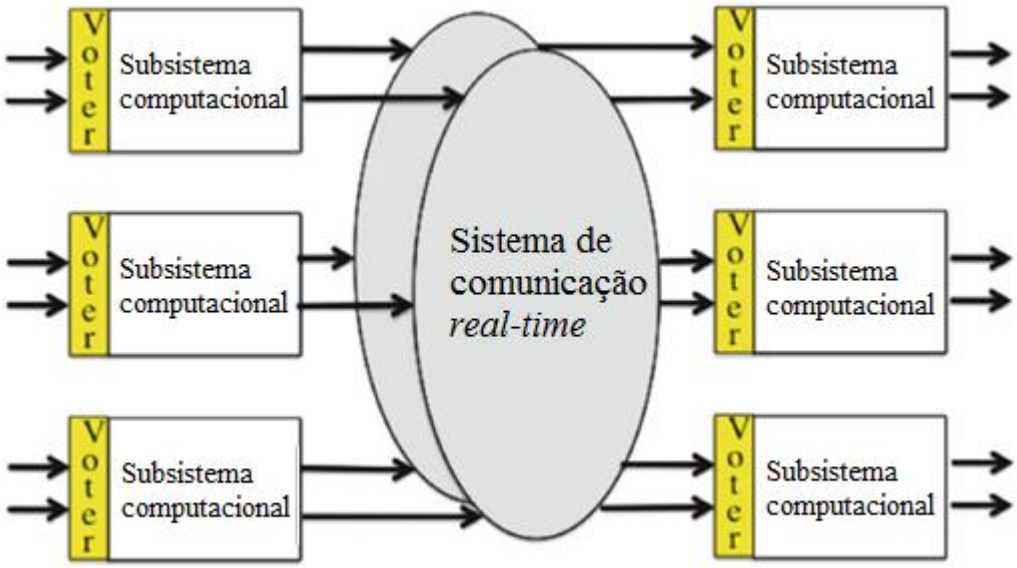


Figura 2.8 - FTU com três FCUS com *voters*

[3]

No *design* de sistemas de tempo-real, para que os mecanismos de tolerância a falhas sejam testados e se confirmem operacionais, devem-se aplicar técnicas especiais de testes, como por exemplo a tradicional injeção de falhas, *scrubbing*. Este termo refere-se a técnicas de teste que são periodicamente aplicadas a fim de detetar unidades faltosas e acumulação de erros nos FCUs. Isto porque apesar de um mecanismo *fault-tolerant* mascarar internamente o erro, para que o utilizador não o perceba, ao nível da sua experiência de utilização do *software*, uma falha do tipo permanente pode incapacitar o próprio mecanismo em situações de falha futuras.

A forma como as interações entre os FTUs são desenhadas define o tipo de arquitetura do sistema.

2.3.3.3 Arquiteturas

A arquitetura de um sistema de tempo-real pode assentar em uma de três tipos: ETA (*Event-Triggered Architecture*), TTA (*Time-Triggered Architecture*) ou arquitetura híbrida.

Na arquitetura ETA só são trocadas mensagens quando algum evento significativo sucede. Este tipo de arquitetura é consideravelmente mais frágil na vertente de tolerância a falhas, pois não é possível distinguir quando não há atividade num componente, pelo menos durante um certo intervalo de tempo. Para colmatar estas lacunas, é necessário implementar um serviço adicional *time-triggered*, ou seja, baseado no tempo do sistema, denominado *fault-tolerant global time*. Este serviço consiste genericamente num *watchdog* periódico que resolve o problema de *membership*, isto é, a cooperação entre os vários componentes.

Já na arquitetura TTA todos os componentes sabem quando um outro falha, ou, pelo menos, um componente sabe da falha de outro que lhe seja funcionalmente dependente. Isto acontece devido à presença de mensagens periódicas de sinalização de atividade (*heartbeats*) trocadas entre eles. Esta arquitetura traz vantagens relativamente à comunicação e esse aspeto torna-se essencial relativamente à noção que o sistema tem do seu próprio estado – normal ou faltoso. Desta forma, é possível saber em que intervalos o componente está "vivo", através das mensagens enviadas entre dois pontos temporais A e B. Se uma primeira mensagem M_A e uma segunda M_B são delimitadoras do intervalo em que o componente esteve ativo, cada recetor sabe quando deve receber a mensagem (*e.g.* através da definição de um *timeout*). Também é possível prever a *temporal accuracy* – tempo útil que uma *real-time entity* demora a aceder a um objeto seu - pois numa arquitetura TT o *delay* de uma "rodada" de comunicação é ou deve ser sabido *a priori*.

A junção destes dois principais tipos de arquitetura denomina-se de arquitetura híbrida (ETA + TTA). Este tipo de arquitetura é composto por interações entre componentes baseadas tanto em eventos como em ocorrências de ordem temporal.

2.3.3.4 Coligações

Torna-se uma enorme vantagem para um sistema de tempo-real quando os seus FTUs são dotados de ajuda mútua. Quando tal acontece, os componentes do sistema têm a capacidade de configurar e/ou de notificar os seus vizinhos, aquando da ocorrência de uma falha. Esta convergência na chamada coligação formada (*coalition*) torna-se uma forma de combater a falha de uma forma imediata e com prontidão, pois este serviço de *membership* (*membership service*) permite ao próprio sistema que efetue uma reconfiguração global, a fim de continuar a providenciar uma qualidade de serviço admissível para o utilizador.

2.3.3.5 Réplicas ativas *vs.* réplicas passivas

As técnicas de redundância podem ser aplicadas nos sistemas de tempo-real, geralmente, sob a forma de duas abordagens específicas: réplicas ativas ou passivas. Quando se utilizam réplicas ativas, são criadas n réplicas de um determinado componente de *software* e

são executadas paralela ou concorrentemente com o original. Este tipo de réplicas traz a vantagem de, sob condições faltosas, existir a possibilidade de tanto o componente primário como os secundários, entregarem o resultado num intervalo temporal aceitável. Contudo, as réplicas secundárias, ou de *backup*, podem atrasar a execução do componente pois consomem, inevitavelmente, recursos da máquina. Relativamente às réplicas passivas, estas podem revelar-se bastante eficazes, principalmente em sistemas *soft real-time*, pois não despendem recursos de processamento na máquina (CPU), apenas de armazenamento na memória (RAM). Todavia, as réplicas passivas podem não cumprir as *deadlines* pois dão início a uma execução tardia, tendo em conta que só começam a execução quando a réplica eleita primária falhar. De forma oposta, esse aspeto revela-se uma vantagem para as réplicas ativas, visto que estas iniciam simultaneamente a execução com o original e por conseguinte, processam mais cedo o resultado a devolver.

2.4 Atributos da *dependability*

Alguns dos atributos da *dependability* de um sistema de tempo-real refletem-se na prática, em variáveis mensuráveis chamadas métricas de performance. As métricas de performance consistem em importantes indicadores de desempenho dos sistemas informáticos e têm um papel imprescindível na avaliação do desempenho de um sistema, a vários níveis. Servem para estabelecer rácios de garantia de serviço e aplicados, *e.g.* num ambiente *auction-bidding* podem servir para a formulação de propostas pelos componentes que constituem o serviço, independentes em termos de localização.

Cada atributo da *dependability* de um RTS representa então um indicador de performance do sistema. Os atributos mais destacáveis são descritos nos seguintes pontos.

2.4.1 Availability

A expressão apresentada na equação (2.3) representa analiticamente a *availability* expectável de um sistema, que por sua vez faz uso da métrica de MTTR, $E\big(M(t)\big)$[3], que é caracterizada pela expressão

$$MTTR = E\big(M(t)\big) = \int_0^{\infty} M(t)\ dt \tag{2.2}$$

[1]

$$A = \lim_{t\to\infty} A(t) = \frac{MTTF}{MTTF + MTTR} \tag{2.3}$$

[1]

[3] Uma função composta $E(Y(x))$ traduz o valor expectável da função em questão, neste caso, para a função $M(t)$. O mesmo acontece na função $R(t)$ de *reliability*, $E(R(t))$, e na função de *safety*.

, sendo que:

$$MTTF + MTTR = MTBF \tag{2.4}$$

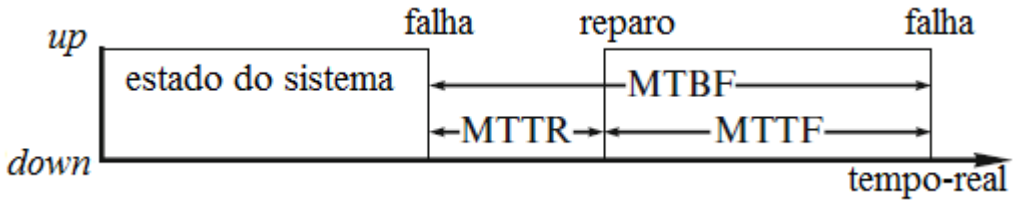


Figura 2.9 – Relação entre MTTF, MTTR, MTBF

[3]

Na Figura 2.9 o sistema tem dois estados, *up* ou *down*, encontrando-se no último no caso de ocorrer uma falha não tolerada. O tempo médio de recuperação (MTTR) é a média de tempo que leva até o sistema passar de um estado inativo para ativo, ao passo que o tempo médio de falha (MTTF) é a média de tempo que leva até o sistema falhar e entrar no estado *down*. Por fim, a soma das duas últimas medidas resulta no tempo médio entre falhas (MTBF), que consiste na média de tempo que decorre entre duas falhas distintas, não toleradas no sistema.

Assim, é possível deduzir que uma alta disponibilidade é alcançada quando se consegue um MTTF longo ou MTTR reduzido.

2.4.2 Reliability

A fiabilidade de um sistema deve ser inversamente proporcional à frequência com que as falhas irão ocorrer. Na equação (2.5) a variável de *reliability* é definida pela função $R(t)$:

$$R(t) = \exp(-\lambda(t - t_0)) \tag{2.5}$$

[3]

- Assume-se que a frequência de falhas do sistema é constante e representa-se na seguinte forma: λ falhas/h;
- O intervalo que se pretende avaliar é delimitado pelo instante inicial e instante final - $[t_0, t]$;

- Esta probabilidade de que o sistema irá fornecer continuamente serviço é medida em FITs (*Failure In Time*), sendo que 1 FIT significa que o MTTF de um determinado dispositivo físico é de $10^9 h$, isto é, uma única falha ocorre num intervalo de 115.000 anos.

Também é possível representar esta variável através da métrica MTTF, cuja expressão algébrica é dada pela equação (2.6):

$$MTTF = E\big(R(t)\big) = \int_0^{\infty} R(t)\, dt \tag{2.6}$$

[1]

2.4.3 Maintainability

Segundo Kopetz, 2011, [3] a *maintainability* é um requisito da *dependability* que é definido por uma função $M(d)$. Esta função representa a probabilidade da atividade do sistema ser restaurada dentro de um intervalo d (tempo de restauro) após a ocorrência da falha. A fim de quantificar esta métrica, é introduzida, tal como na *reliability*, uma frequência constante de reparação μ (reparos por hora) e um MTTR. De notar que, para um desenvolvimento de um sistema dotado de uma alta manutenção, é necessário proceder à criação de FRUs (*field replaceable units*) conectados por interfaces modulares, facilmente conectadas/desconectadas, que disponibilizem serviços. Apesar do custo de produção dos FRUs ser consideravelmente maior, a sua aplicação acarreta a vantagem de facilmente se poder substituir um FRU faltoso.

2.4.4 Safety

A *safety*, representada por uma função $S(t)$, é dada pela equação (2.7. Esta variável quantifica a probabilidade de que o sistema irá permanecer protegido durante um período t. Esta função utiliza a métrica MTTC, que é definida da seguinte forma:

$$MTTC = E\big(S(t)\big) = \int_0^{\infty} S(t)\, dt \tag{2.7}$$

[1]

2.5 Técnicas de tolerância a falhas

2.5.1 Conceitos e termos

A fim de combater a inoperacionalidade dos RTS foram idealizadas e modeladas algumas abordagens teóricas, comumente denominadas de técnicas de *design* de tolerância a falhas. Em *cloud computing*, *i.e.* sistemas de cariz distribuído, estas abordagens podem ser

utilizadas no âmbito da tolerância a falhas ao nível do *software* (*software-based fault-tolerance*) e/ou ao nível da rede (*network-based fault-tolerance*). O primeiro tipo de abordagem consiste em esquemas tolerantes a falhas implementados na aplicação, *middleware* ou SO. Por outro lado, o segundo tipo é composto por esquemas resilientes implementados na infraestrutura de rede. Este trabalho foca-se na tolerância a falhas ao nível de *software*. No entanto, o tema de tolerância a falhas em redes de computadores é abordado sucintamente no tópico 2.6.7.

Anderson, 1981, [5] refere dois âmbitos essenciais na área de tolerância a falhas: processamento de erros e tratamento de falhas. O processamento de erros consiste na tentativa de remoção de erros do estado em que o sistema se encontra. O tratamento de falhas é o esforço realizado no sentido de prevenir que as falhas voltem a acontecer, que no fundo corresponde a tentar quebrar o ciclo da cadeia de Laprie.

As quatro fases que devem constituir a tolerância de falhas são as seguintes:

- Deteção de erros;
- Avaliação de danos;
- Processamento de erros:
    - *Recovery* (recuperação);
    - *Compensation* (compensação);
- Tratamento de falhas:
    - *Diagnosis* (diagnóstico);
    - *Passivation* (prevenção).

De acordo com Hiller, 1998, [1] a primeira fase, deteção de erros, consiste na identificação de um estado erróneo, isto é, um estado que pode conduzir a uma falha subsequente. Se, efetivamente, um erro for detetado, procede-se à segunda fase, avaliação de danos, que consiste em determinar que áreas do sistema foram afetadas devido ao erro.

Após a avaliação dos danos causados pelo erro, é necessário passar à fase de processamento de erros. Através do processo de recuperação é possível realizar o processamento de erros por *backward recovery* ou por *foward recovery*. No primeiro caso, é necessário armazenar uma imagem do sistema (*real-time image*), que consiste num *snapshot* do sistema, antes do erro. De seguida, inicia-se o restauro do *recovery point*, que coloca o sistema de novo num estado *error-free*. No segundo caso, *foward recovery*, tenta-se encontrar um novo estado em que o sistema fique operacional. Na maior parte das vezes o sistema opera em modo degradado, o que implica que haverá uma redução da qualidade do nível de serviço fornecida. Por outro lado, na compensação de erros (*compensation*) há um esforço no sentido de cumprir a entrega dos resultados produzidos sem erros. Para isso recorre-se geralmente a técnicas de redundância.

O tratamento de falhas só acontece quando uma falha foi efetivamente tolerada. O diagnóstico de falhas (*diagnosis*) consiste em determinar a causa do erro, relativamente à sua

localização no sistema e natureza. Já a prevenção de falhas consiste no resguardo do sistema, para que as falhas não voltem a suceder.

A Figura 2.10 ilustra o modelo de interação entre as quatro fases referidas e o modo como elas se relacionam com a cadeia de faltas, erros e falhas, num sistema.

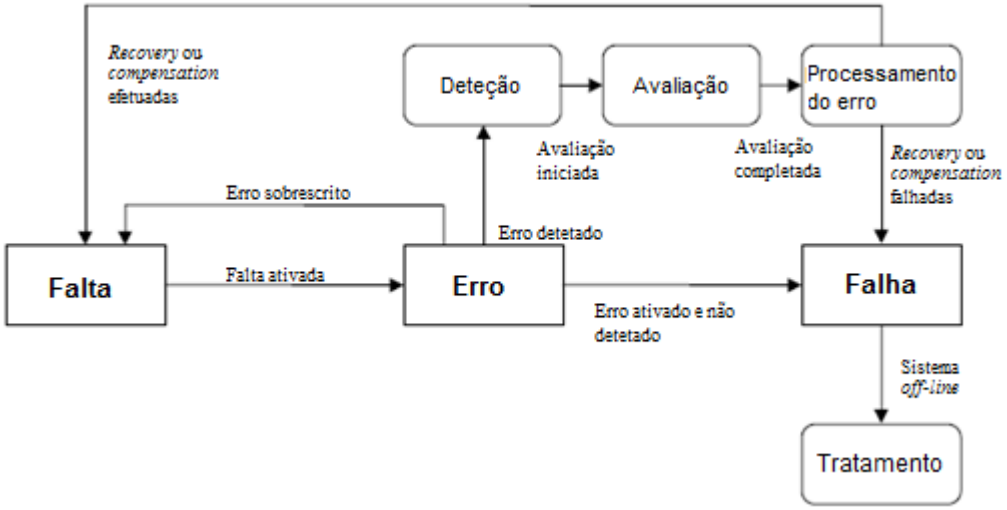


Figura 2.10 - As quatro fases de tolerância a falhas

[1]

Um procedimento de recuperação de falhas num esquema *fault-tolerant* envolve três fases principais:

- *Failure detection* (deteção da falha) – fase entre a ocorrência e a descoberta da falha;
- *Failure notification* (notificação da falha) – fase entre a deteção e a notificação dos nós responsáveis pela recuperação da falha;
- *Failure recovery* (recuperação da falha) – fase entre a notificação e o instante em que o estado anterior à falha foi restaurado.

Antes de prosseguir para os principais métodos de tolerância a falhas, convém explicar a notação utilizada para classificação de modelos de tolerância a falhas. A notação de Avizienis, 1986, [6] permite classificar uma técnica de tolerância a falhas em três domínios diferentes: espaço (H*ardware*), informação (S*oftware*) e repetição (Tempo).

O domínio de espaço refere-se à redundância de *hardware*, que pode ser estática ou dinâmica. É estática quando o sistema tenta mascarar um erro, enquanto na dinâmica é permitida a ocorrência do erro e tenta-se recuperar o módulo que falhou. O domínio de espaço é denotado pela letra H.

O domínio da informação significa redundância ao nível de *software* e quantifica quantos canais independentes de dados são utilizados. Geralmente, conjuga-se esta forma de redundância com a redundância de *hardware* dinâmica. A letra S denota este domínio.

A redundância temporal, representada pelo domínio da repetição, é constituída por duas estratégias: (a) o reinício de um programa após a deteção de um erro e (b) a execução repetida para deteção do erro. Este domínio é representado pela letra T.

De referir que a redundância utilizada é representada pela letra N e, no caso de esta ser diversificada é denotada pela letra d.

Exemplo: Um sistema que seja composto por um canal de *hardware*, por dois de *software* diversificados e por uma linha temporal, é caracterizado segundo a expressão $1H/2dS/1T$.

Num sistema de tolerância a falhas, também é possível referir o tipo de relação *robot-task*. Gerkey e Mataric, 2004, [7] para classificar um sistema na relação *robot*-tarefa, no âmbito da robótica multitarefa (MRTA), referem a seguinte taxonomia:

- *Single-Task* (ST) *robots* – máquina que só pode executar uma tarefa;
- *Multi-Task* (MT) *robots* – máquina que executa várias tarefas;
- *Single-Robot* (SR) *task* – tarefa que pode ser atribuída a uma só máquina;
- *Multi-Robot* (MR) *task* – vários *robots*, que formam a coligação, podem executar concorrentemente uma tarefa;
- *Instantaneous Assignment* (IA) – a alocação é feita sem considerar tarefas futuras;
- *Time-Extended assignment* (TE) – a alocação é feita ao longo do tempo, ou seja, tem em conta tarefas posteriores.

2.5.2 Tipos de técnicas

Uma técnica de tolerância a falhas é uma forma concreta e prática de eliminar ou minimizar o impacto que essa falha provoca no sistema onde ocorre.

Cada uma das abordagens teóricas referidas no ponto anterior expande-se para um leque de diversas técnicas que foram desenvolvidas e, muitas delas, modificadas ao longo das últimas três décadas.

Nas fases enumeradas em 2.5.1 são aplicadas várias técnicas genéricas de tolerância a falhas. Cada esquema pode aplicar uma ou mais técnicas e conjugá-las de forma adequada para que o sistema tire o máximo de proveito. As principais técnicas são as seguintes:

- *Checkpointing* – consiste em copiar o estado inteiro do serviço permitindo, em caso de falha, a sua migração (*migration*) para outras máquinas e a sua reinicialização (*restart*), após a recuperação da falha, desde o seu último *checkpoint*. Esta técnica pode ser implementada das seguintes maneiras:
    - Não coordenada;
    - Coordenada;

    - *Communication-induced* – *checkpointing* efetuado através de mensagens: global (aplicado num conjunto de processos) ou local (aplicado num só processo);
- Migração – consiste na transferência de processos e de dados entre dois locais, mesmo durante a execução do processo;
- Replicação – consiste em distribuir cópias do mesmo serviço para diferentes locais e pelo menos um deve ser alcançável, mesmo numa situação faltosa. Esta técnica pode ser feita a três níveis:
    - *Job* – replicação de processos;
    - *Data* – replicação síncrona ou assíncrona de dados, dependendo da sua consistência;
    - *Component* – replicação de componentes;
- Escalonamento/Redundância – consiste no agendamento das tarefas do sistema para alcançar um balanceamento de carga melhor. Pode ser feito de três maneiras distintas:
    - *Space scheduling* – usado para tratar falhas do tipo permanente, através de *hardware* ou *software*, *e.g.* abordagem *primary-backup*;
    - *Time scheduling* – usado para tratar falhas do tipo *intermittent*;
    - *Hybrid Scheduling* – mistura das duas maneiras anteriores, para tratar os dois tipos de falhas referidos, tempo e espaço.

Na maior parte das vezes, durante o processo de recuperação as aplicações/*middleware* não são *QoS-aware*, *i.e.* não têm em atenção o problema de assegurar o nível de QoS mínimo pretendido pelo utilizador, para que a execução do serviço permaneça dentro dos parâmetros, nas máquinas a que os componentes são atribuídos. Uma implementação deficiente pode trazer consequências devastadoras para a performance, *e.g.* devido à necessidade constante de sincronizações.

Os dois esquemas de tolerância a falhas em *software* que se foram destacando e que foram continuados ao longo das últimas décadas são o *Recovery Block* (RB) e o *N-Version Programming* (NVP). A partir destes principais esquemas surgiram bastantes derivações, nomeadamente do RB, cujo objetivo consiste em adaptarem-se às súbitas alterações, frequentemente encontradas em ambientes *real-time*.

2.5.3 Recovery Block

O esquema RB, segundo a notação de Avizienis, é classificado como $1H/NdS/NT$ e é composto por três principais componentes:

- Um módulo primário – realiza a tarefa principal, executando o bloco de código primário;
- Módulos alternativos – realizam a mesma tarefa, porém de maneira diferente do módulo primário, através da execução de blocos de código secundários;

- Um teste de aceitação (*acceptance test*) – um teste que verifica a aceitabilidade dos resultados produzidos pelos módulos para o meio.

Num esquema RB são geralmente utilizadas técnicas de processamento de erros como *backward recovery* e técnicas de tolerância a falhas como *recovery cache*, *e.g.* ao entrar num *recovery block* efetua-se *checkpointing*. Este esquema baseia-se numa arquitetura TT, visto que grande parte das falhas que surgem nos módulos são detetadas através de *timeouts*. Os *recovery blocks* podem conter *sub recovery blocks* ou *nested recovery blocks*, para facilitar a delegação de operações segundo uma hierarquia estabelecida em *design* ou *runtime*. As vantagens que este esquema acarreta são: em primeiro lugar, não é preciso saber que falhas ocorreram e que danos causaram, devido à aplicação de *backward recovery points*; em segundo, os módulos são independentes do *design*; pode haver vários *acceptance tests* através de *nested recovery blocks*, que tornam os módulos menos propícios a erros, apenas degradando o QoS. Esses *acceptance tests* fazem uma filtragem de erros vertical no domínio do componente, e evitam um efeito dominó caso sucedessem *nested conversations*. Ainda relativamente às vantagens, em sistemas embebidos não há muito *overhead* de informação nos *recovery points*, muito menos com o uso de *watchdog-timers*[4]. Por outro lado, relativamente aos aspetos negativos, a elevada complexidade dos *acceptance tests* pode torná-los lentos.

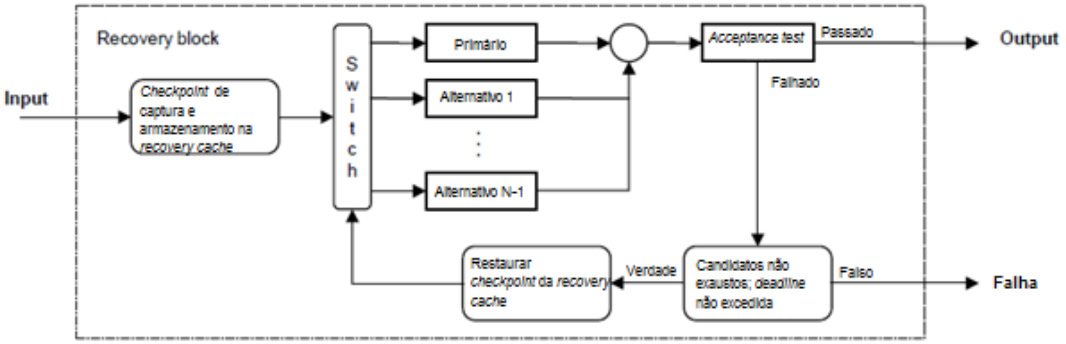


Figura 2.11 - Estrutura do esquema RB

[1]

2.5.4 N-Version Programming

O NVP assenta num esquema concetual de votação, tal como o esquema TMR. O NVP é composto por uma especificação inicial (*initial specification*), *N* versões de *software*, mecanismos de decisão e um programa supervisor. As principais vantagens deste esquema são a interdependência (*e.g.* no âmbito do desenvolvimento de *software* por diferentes equipas) das *N* versões que torna o esquema menos propenso a erros. Relativamente às

[4] Temporizador de *software* que dispara um *reset* do sistema ou uma ação corretiva, devido a uma condição faltosa, *e.g.* o programa ficar pendurado (*hang*).

desvantagens, as falhas de *design* na *initial specification* podem afetar as N versões, apesar de terem pouco impacto em falhas residuais. A abordagem de *inexact voting* pode levar a vários resultados corretos, o que pode criar um atraso na decisão de qual o resultado a retornar. Existem ainda as desvantagens de ser necessário espaço para armazenar as N versões, e de haver escassez de tempo e de capacidade de processamento para sincronizá-las juntamente com o algoritmo de decisão. Por último, este modelo, através da notação utilizada, é denotado $N(d)H/NdS/1T$.

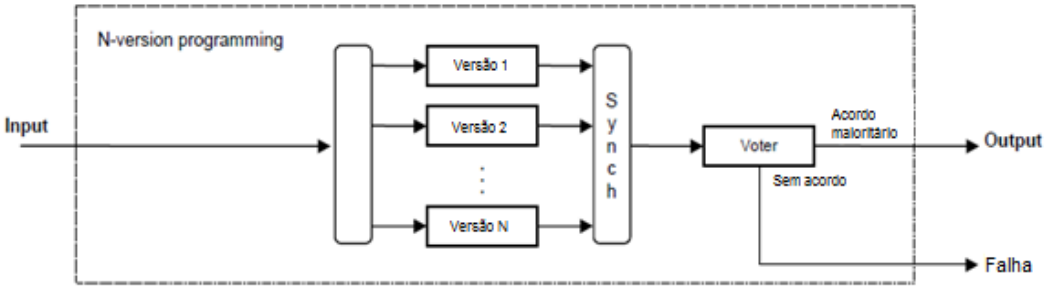


Figura 2.12 – Estrutura do esquema NVP

[1]

2.5.5 Consensus Recovery Block

O esquema CRB consiste simplesmente numa fusão entre o NVP e o RB, na medida em que os resultados (*output*) estão sujeitos a uma primeira fase de *voting*, e, caso não se verifique consenso, passam por uma segunda fase que consiste num *acceptance test*, enquadrado no esquema RB. As grandes vantagens deste esquema são as seguintes:

- Não são precisos mecanismos de *recovery cache*;
- Minimiza o problema de ter vários resultados corretos nas N versões;
- O *acceptance test* já não afeta o esquema, em termos de complexidade, pois os resultados têm de passar previamente pela filtragem do sistema *voter*.

Quanto aos aspetos negativos, é necessário um custo de desenvolvimento acrescido, pelo facto de haver um número elevado de componentes a implementar, algumas das desvantagens dos esquemas originais do RB e NVP são transferidas para este novo esquema, como a complexidade dos *acceptance tests*. Por fim, verifica-se um *overhead* apenso ao sistema, devido à quantidade e diversidade de componentes.

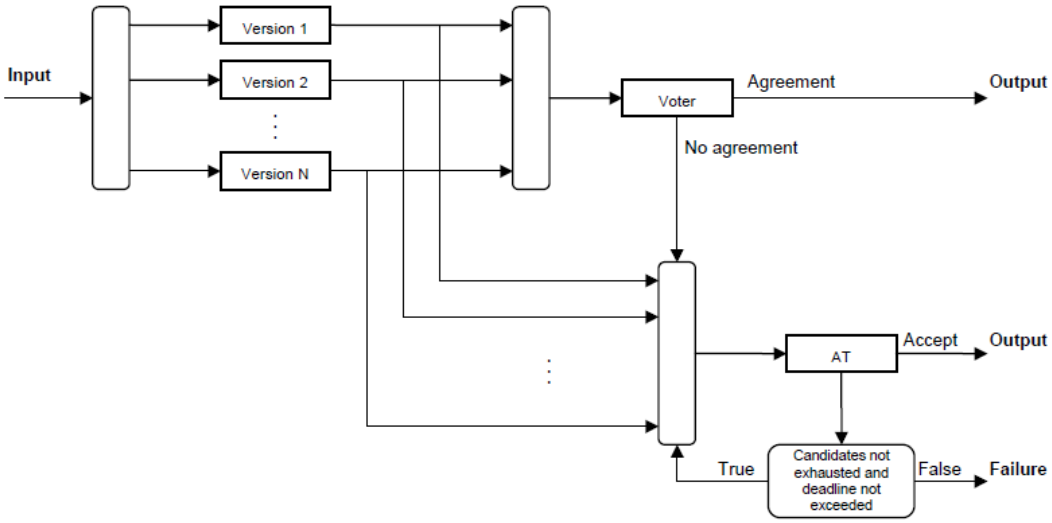


Figura 2.13 - Estrutura do esquema CRB

[1]

A expressão, segundo a classificação de Avizienis, que define este esquema é $N(d)H/NdS/1T$.

2.5.6 Distributed Recovery Block

Para ir ao encontro das necessidades de um RTDS (*Real-Time Distributed System*) as técnicas de redundância tornam-se o foco principal na implementação de um esquema de tolerância a falhas. Tal como a Figura 2.14 mostra, são apresentados dois nós, um primário e um secundário, pressupondo-se à partida um ambiente em que ocorrem falhas distribuídas, ou seja, em partes diferentes do sistema. A redundância do sistema pode também ser feita localmente, a partir da replicação dos *recovery blocks* na mesma máquina.

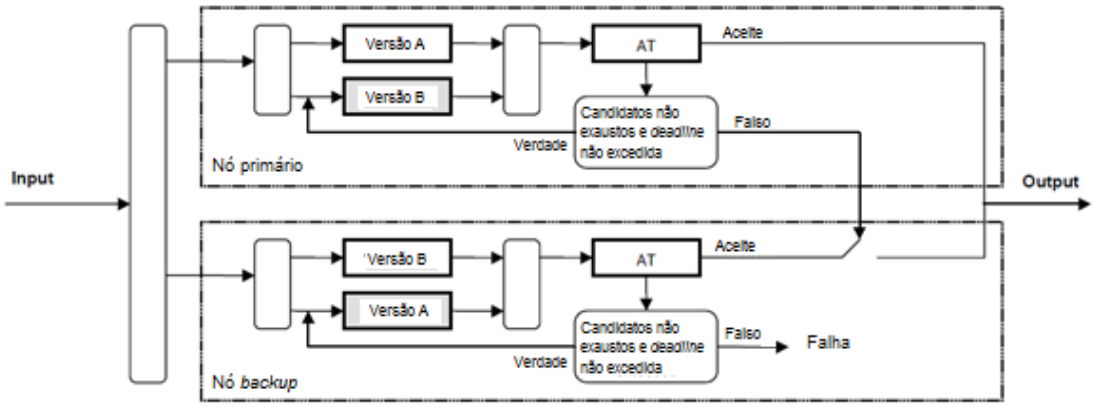


Figura 2.14 - Estrutura do esquema DRB

[1]

Atenda-se que na Figura 2.14 é apresentado um esquema DRB num sistema classificado como $2H/2dS/1T$.

As vantagens deste esquema são o processamento em vários nós, logo efetua-se um balanceamento de carga mais eficiente. Para além disso, o tempo de recuperação é mais curto e a sobrecarga na réplica não é significativa em termos de *overhead*, isto se esta for do tipo físico (réplica colocada noutra máquina física). As desvantagens são, primeiramente, em caso de falha, o nó primário só efetua *rollback/backward recovery* depois de executar o bloco *try* alternativo, para manter os dados consistentes, logo demora mais tempo. Em segundo lugar, este esquema foi desenhado para sistemas onde o *output* de um par de processos era *input* para outros dois processos. Este esquema é classificado como $N(d)H/N(d)S/1T$, de acordo com a notação utilizada.

2.5.7 Extended Distributed Recovery Block

Considerando os benefícios que diversos sistemas tiraram da aplicação do DRB, foi desenvolvido um esquema que pretende definir uma hierarquia nos nós que entram na constituição de um sistema. A Figura 2.15 apresenta o EDRB, procedente do DRB, que consiste num esquema composto por nós supervisores e pares de nós operacionais.

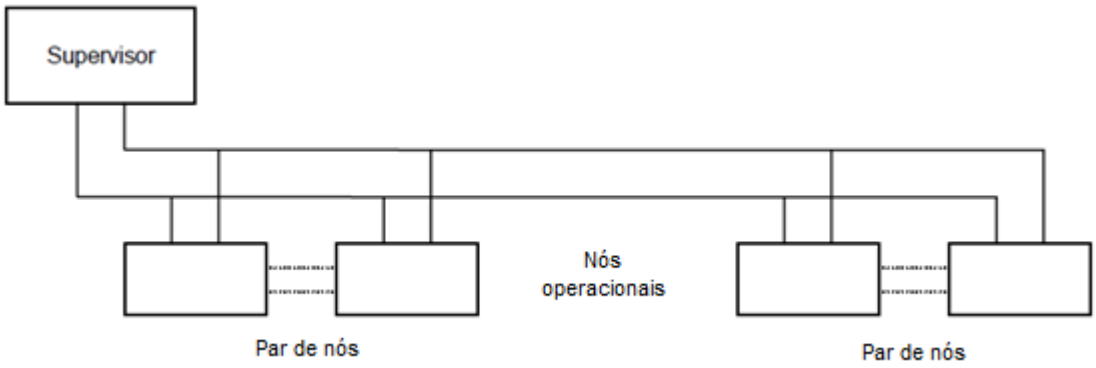


Figura 2.15 - Estrutura da hierarquia do esquema EDRB

[1]

Geralmente, num esquema EDRB, existe um nó supervisor que controla um ou mais pares de nós operacionais. Estes são formados por um nó primário (*primary node*) e um outro secundário (*shadow node*). Em cada nó operacional existe ainda um componente chamado *node executive* que é constituído por um *node manager* e um *node monitor* e é responsável por enviar e receber *hearbeats* para o seu par. As principais vantagens deste esquema são: a redundância, aplicada através dos pares de nós operacionais; o supervisor não é crucial na sincronização dos nós, apesar de importante. As desvantagens são o facto de haver dois SPOFs (*Single Point Of Failure*): para além do nó supervisor, o *node manager* pode falhar, deixando de emitir e de receber *heartbeats*. Caso aconteça, esta situação pode destabilizar o sistema, porque um nó operacional pressupõe erradamente que o seu

par falhou e assume o papel primário. Sendo assim, o *output* é replicado e é criado *overhead* desnecessário na solução. Este esquema tem ainda a desvantagem de ter vários *acceptance tests*, que, como já foi referido, aumentam a complexidade da operação.

A Figura 2.16 ajuda a perceber o funcionamento do EDRB.

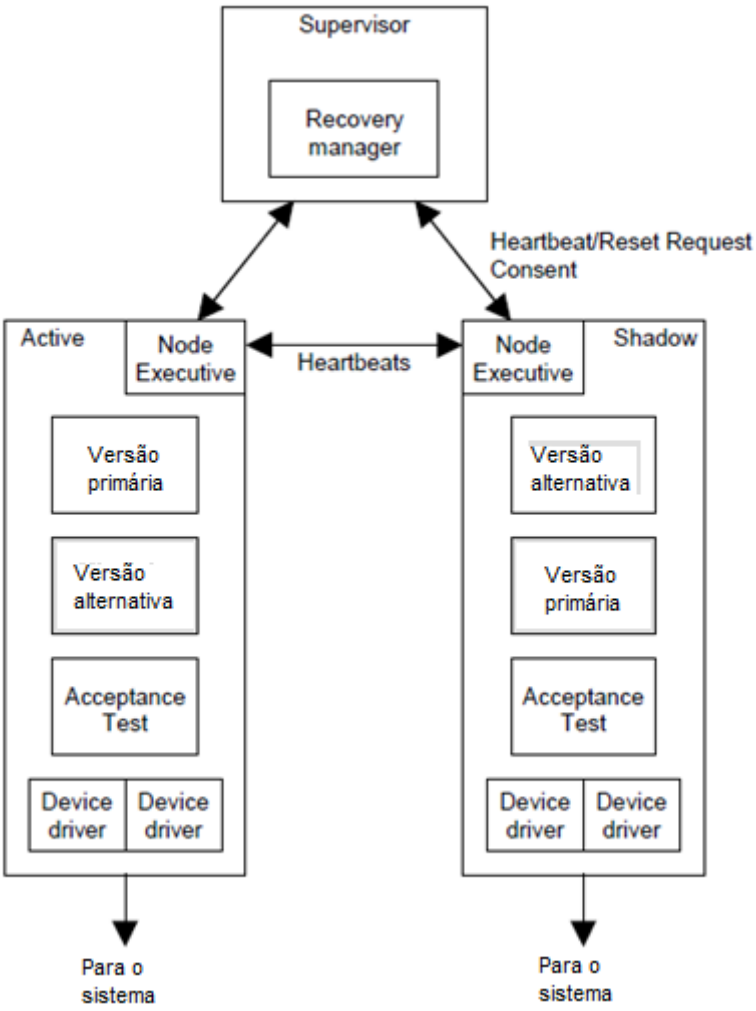


Figura 2.16 - Estrutura do esquema EDRB

[1]

Quando ocorre uma falha na conversação entre os componentes *active* e *shadow* é lançado um pedido ao supervisor para reiniciar o diálogo de troca de *heartbeats*.

2.6 Agentes inteligentes

Existem várias definições para "agente". Os agentes inteligentes (Russel e Norvig, 1995) são processos de *software* independentes capazes de lidarem com tratamento de informação de forma semelhante aos seres humanos pois percebem o contexto em que estão

inseridos, através de sequentes perceções recebidas por sensores. Os agentes podem atuar sobre esse meio através de atuadores. Do ponto de vista de sistemas distribuídos, um agente é um programa com uma identidade única que consegue mover o seu código, dados e estado intactos entre máquinas na rede [8], sempre que dotado com a capacidade de se mobilizar dentro de uma infraestrutura de computadores.

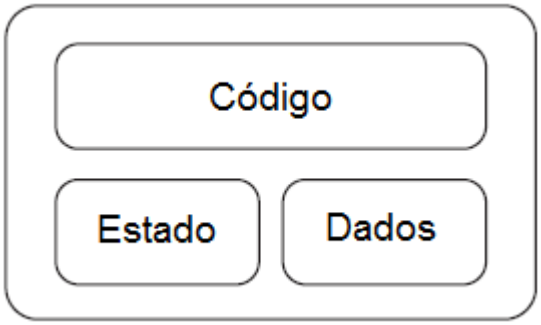


Figura 2.17 - Estrutura básica de um agente

[8]

Os agentes são caracterizados principalmente pela sua autonomia, pro-atividade, sociabilidade e pela capacidade de comunicarem. Sendo autónomos, podem-lhes ser delegadas tarefas complexas, que são executadas independentemente. Ao serem proactivos são automaticamente dotados com a iniciativa de levar a cabo uma determinada tarefa sem a necessidade de receberem um estímulo explícito de um utilizador. Por último, o facto de serem sociáveis e aptos para comunicar de forma independente, permite-os interagir com outra entidade (ser-humano, agente ou outra) para ajudar ou serem ajudados a cumprir determinados objetivos [8].

Para concretizar esses tipos de interações com o ambiente existe a possibilidade de dotar os agentes com comportamentos. Os comportamentos consistem em ações que realizam tarefas específicas, e que podem ser executadas uma única vez, ciclicamente ou condicionalmente. Face ao contexto em que se encontra, um agente pode possuir crenças, desejos e intenções ou planos, funcionando numa arquitetura BDI, *reactive*, entre outras (ver 2.6.2).

Quando são desenvolvidos com o suporte de uma plataforma de mobilidade e de um protocolo de comunicação, os agentes tornam-se capazes de se mover entre as máquinas da rede e transportar consigo todo o seu código e ficheiros serializáveis. Desta forma, após a sua migração um agente pode prosseguir a sua execução a partir do ponto de interrupção (criado antes de se mover). Geralmente, a localização de um agente é transparente para a plataforma de mobilidade.

A modelação de agente móveis baseada em máquinas finitas de estados é bastante eficaz e menos complexa do que outras abordagens. Cada estado contém uma ou mais ações a realizar.

Os agentes podem facilmente trocar mensagens entre si de um modo coordenado e sincronizado. Relativamente à colaboração entre agentes, a execução distribuída de tarefas complexas e inteligentes pode ser facilitada, desde que estes estejam organizados em "sociedades", assim formando um único grupo cooperante, cujo conceito denomina-se *Society of Mind* [9]. Os agentes conseguem resolver problemas ao realizar tarefas complexas com um objetivo comum [10].

Em termos de sobrecarga na rede, existem casos em que os agentes necessitam de migrar para cumprir as suas tarefas. De outra forma, seria impossível transmitir todos os dados relativos a uma tarefa específica, em virtude das limitações da rede de comunicação subjacente [11].

2.6.1 AOP

Agent-Oriented Programming é um paradigma recente de desenvolvimento de *software* que tenta trazer conceitos de teorias da AI (*Artificial Intelligence*) para o mundo dos sistemas distribuídos [8]. Os MAS (*Multi-Agent Systems*) são utilizados num conjunto diversificado de aplicações. Este conjunto estende-se desde aplicações de pequena dimensão, como sistemas de assistência pessoal, até sistemas abertos e complexos, como aplicações industriais, cujas missões são críticas.

Quanto se adota uma abordagem orientada a agentes na resolução de um problema, há uma série de problemas independentes entre si, que devem ser abordados cuidadosamente, *e.g.* o modo como é feita a comunicação entre os agentes.

2.6.2 Tipos de arquiteturas

Existem vários tipos de arquiteturas para modelação de um sistema baseado em agentes, das quais se destacam as seguintes:

- *Logic-based* (simbólica);
- *Reactive*;
- BDI;
- *Layered* (híbrida).

A arquitetura *Logic-based* tem um funcionamento alicerçado em técnicas tradicionais baseadas em conhecimento (*knowledge-based techniques*). É chamada de "simbólica" pois o ambiente é simbolicamente representado e manipulado usando mecanismos de inferência. Isto torna-se uma vantagem visto que o conhecimento humano é também simbólico, logo a informação é mais simples de codificar e mais simples para os humanos entenderem.

A arquitetura *reactive/reflexive* funciona sob a forma de interações estímulo-resposta e na sua génese está a arquitetura de subsunção de Brooks (1991). Esta arquitetura, ao con-

trário das *logic-based*, não lida com informação simbólica e portanto não processa nenhuma inferência complexa sobre os dados. O conceito que está no núdeo do desenvolvimento desta arquitetura enuncia que qualquer comportamento pode ser gerado sem precisar de utilizar técnicas simbólicas de AI ou de representações explícitas. A arquitetura de subsunção define *layers* de FSMs (*Finite State Machines*) que estão conectadas a sensores que transmitem informação em tempo-real (ver Figura 2.18).

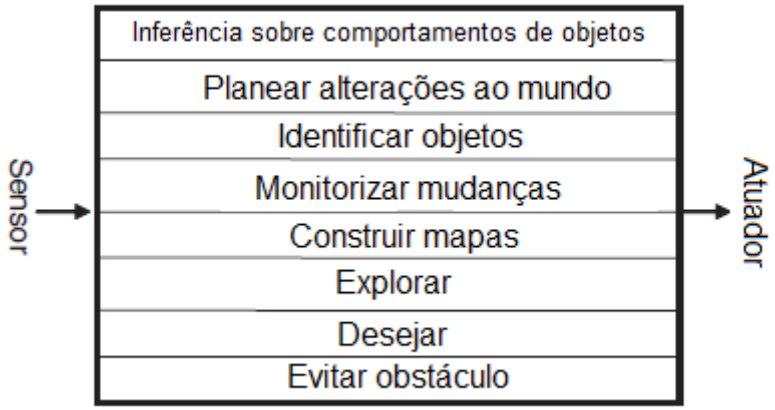


Figura 2.18 - Uma arquitetura de subsunção para um *robot* de navegação

[8]

A arquitetura BDI (Rao e Georgeff, 1995) baseia-se em quatro estruturas de dados centrais: crenças, desejos, intenções e planos, e um interpretador. Um exemplo de uma arquitetura deste tipo bem conhecida é o PRS (*Procedural Reasoning System*), de Georgeff e Lansky, 1987.

A arquitetura *layered* (híbrida) oferece a possibilidade de incorporar comportamentos reativos (*reactive*) e deliberativos (*logic-based* e/ou BDI). Para se tornar flexível, a arquitetura hierarquiza os subsistemas em camadas (*layers*). Assim, é possível a coexistência dos dois tipos de comportamentos no sistema.

2.6.3 Tipos de agentes

À semelhança das arquiteturas retratadas no tópico anterior, os agentes também podem ser modelados de diferentes maneiras, em conformidade com os seus objetivos. Genericamente, os agentes podem ser de três tipos: *reflex/reactive agents*, *deliberative agents* e *collaborative agents*.

Os primeiros referem-se a agentes que não possuem modelos internos do seu mundo e que reagem a estímulos externos (*external stimuli*). Os seus comportamentos resultam de interações simples com agentes individuais e baseiam-se em dados captados por sensores e não em dados representados simbolicamente (técnicas AI).

Quanto aos *deliberative agents*, são caracterizados pela sua capacidade de planeamento face ao estado do "mundo", possuem conhecimento e atuam através de cooperação ativa.

Por último, os *collaborative agents* resolvem problemas ao colaborarem conjuntamente. Os agentes deste tipo têm a capacidade de resolver problemas de grande dimensão, ou seja, de executar tarefas complexas e, para tal, comunicam frequentemente.

2.6.4 Coordenação

A disposição dos agentes na forma de uma organização estruturada é a melhor maneira de explorar a definição de papéis, os caminhos de comunicação e as relações de autoridade entre eles [12].

No âmbito deste trabalho, as melhores técnicas de coordenação entre agentes são a coordenação orientada à negociação, o *contract net protocol* e a coordenação orientada ao planeamento.

2.6.4.1 Orientada à negociação

A técnica de coordenação através de negociação (Bussmann e Muller, 1992) consiste num processo de comunicação de um grupo de agentes que tem como objetivo chegar a um acordo mútuo acerca de um determinado problema. A negociação pode ser de dois tipos: *competitive* ou *cooperative*. *Competitive negotiation* é utilizada em circunstâncias onde os agentes envolvidos têm objetivos distintos, logo independentes entre si. Já em situações onde os agentes têm um objetivo em comum ou uma única tarefa para executar recorre-se à *cooperative negotiation*.

2.6.4.2 Contract net protocol

Esta técnica (Smith e Davis, 1980) é muito utilizada em alocação e distribuição de tarefas e de recursos entre agentes. No *contract net protocol* existe uma estrutura de mercado descentralizada onde os agentes podem desempenhar o papel de *manager* ou de *contractor*. Os problemas encontrados pelos agentes são decompostos em problemas mais pequenos chamados "subproblemas". A resolução de um problema parte da atribuição dos respetivos "subproblemas" aos *contractors*. De seguida, efetuam-se os seguintes passos:

1. *Manager* anuncia o contrato;
2. Os *contractors* enviam *bids* para o *manager*, em resposta ao contrato anunciado;
3. As *bids* são avaliadas pelo *manager* que atribui, em função destas, os subproblemas aos *contractors*.

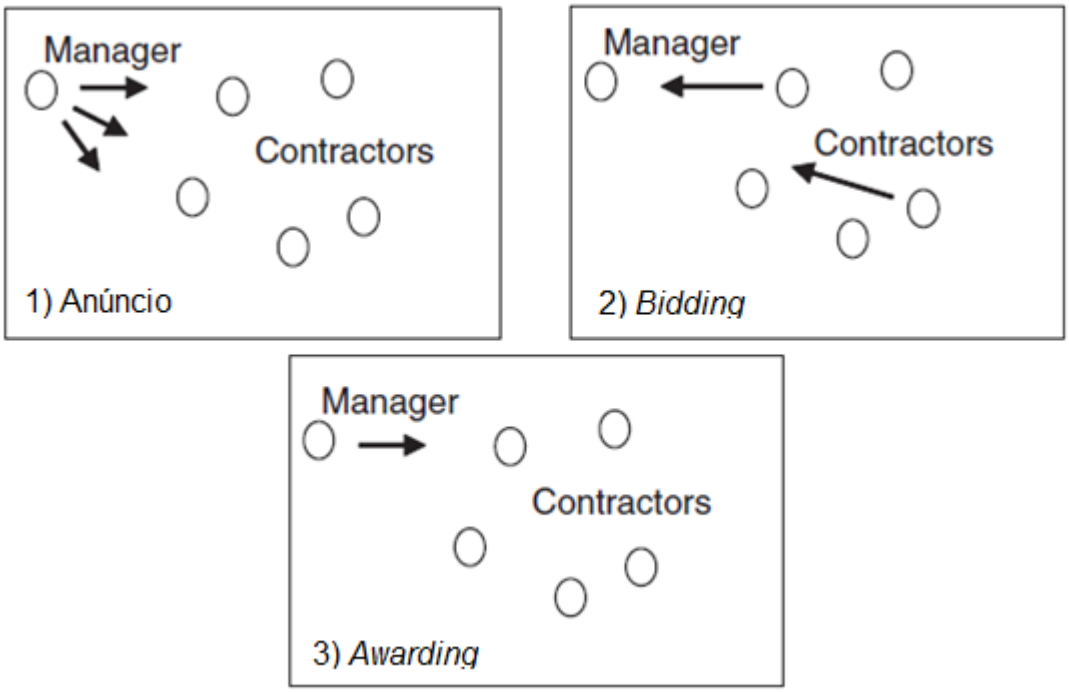


Figura 2.19 – Etapas do *contract net protocol*

[8]

2.6.4.3 Coordenação orientada ao planeamento

Nesta técnica os agentes constroem um plano com uma lista de ações a executar futuramente. O planeamento pode ser de dois tipos: centralizado ou descentralizado. Por um lado, no planeamento centralizado, *centralized multi-agent planning*, existe um agente central que coordena todos os planos individuais dos agentes e constrói um mapa de todos os planeamentos. Após este mapa ser analisado, o agente central tenta identificar e remover todos os conflitos. Por outro lado, o *decentralized multi-agent planning* tem como finalidade fornecer a cada agente um mapa dos planos dos outros agentes. Assim, os agentes constroem e atualizam os seus planos individuais e os mapas dos planeamentos dos outros agentes até que todas as interações conflituosas sejam eliminadas.

2.6.5 Aplicação em tolerância a falhas

Os sistemas multiagente vêm ao encontro das necessidades da tolerância a falhas em sistemas de tempo-real, nomeadamente pela capacidade que têm de formar coligações. Apesar de os agentes atuarem de forma autónoma e "egoísta", quando se juntam e se organizam em coligações, que têm em vista um objetivo comum, podem tornar-se benéficos na partilha e distribuição de tarefas. Para além disso, os agentes inteligentes, dotados de autonomia, aumentam a descentralização dos sistemas e tornam menos provável a existência de SPOFs. Proporcionam à coligação formada um alto dinamismo pois adaptam-se

continuamente "mundo" e, assim, conseguem contornar as falhas que ocorram no ambiente em que se encontram. Em adição, outras circunstâncias que alteram o meio, como a adesão de um novo nó à coligação ou até a remoção de um nó já integrado nesta, também podem ser toleradas.

Em termos gerais, os agentes têm uma perceção elevada acerca do estado do sistema, pois facilmente trocam mensagens uns com os outros.

2.6.6 Vantagens e desvantagens

A utilização do paradigma AOP tem algumas vantagens e desvantagens comparativamente com sistemas *non-mobile*. Relativamente às vantagens destacam-se as seguintes :

- Processamento assíncrono e independente – uma vez migrados para uma nova plataforma os agentes não têm mais de comunicar com o seu criador para executar uma tarefa. Podem, no entanto, ter de enviar resultados. Estas características trazem enormes vantagens em cenários onde estão envolvidos dispositivos móveis com poucos recursos, isto é, com baixa capacidade de processamento e de armazenamento;
- Tolerância a falhas – os agentes podem ser desenvolvidos com inteligência para lidar com situações onde existem problemas de indisponibilidade nos *hosts* da rede (falhas temporária ou permanentes). Adaptam-se facilmente a circunstâncias novas, que eventualmente se tornariam inconvenientes num sistema desenvolvido com outro paradigma. AOP é um paradigma com resiliência e, nesta perspetiva, torna-se apropriado para ambientes instáveis, *i.e.* possivelmente alterados devido ao surgimento de falhas;
- *Sea of data applications* – os agentes móveis são apropriados para aplicações que lidam com grandes quantidades de dados. Os agentes podem mover-se para os dados, o que se torna mais eficiente do que o processo oposto.

Infelizmente os agentes têm algumas desvantagens. As mais relevantes descritas em [13] são as seguintes:

- Escalabilidade e performance – O rigor dos *standards* de interoperabilidade (comunicação e transporte) podem causar a criação de *overheads*, desta forma aumentando a carga de processamento;
- Portabilidade e estandardização – Os agentes não podem interagir se não seguirem *standards* de comunicação, como FIPA ou OMG MASIF (*Mobile Agent System Interoperability Facility*);
- Segurança – se a *framework* utilizada não oferecer mecanismos que tratem da integridade, confidencialidade e autenticidade da informação trocada, e estes não forem devidamente implementados, os agentes podem tornar-se uma ameaça, pois criam vulnerabilidades sempre que atravessam a rede.

2.6.7 Trabalho relacionado

Através do estudo de vários documentos, elaborados na sequência de trabalhos de investigação, foi possível analisar uma quantidade diversificada de propostas no âmbito da tolerância a falhas em RTS baseadas em *software*: planeamento de execução de tarefas distribuídas, resolução de problemas em sistemas distribuídos, formações de coligações e arquiteturas *self-healing*.

Em [14], Dubey *et. al*, 2006, propõem um modelo dotado de *Reflex and Healing* com um componente de diagnóstico e outro de deteção e mitigação de falhas. A arquitetura deste modelo é híbrida, dado que os autores aglomeram a ETA e a TTA. Neste trabalho são demonstradas algumas características como *liveness* – se o sistema não tem *deadlocks*, então a junção de comportamentos autónomos não os cria -, *safety* – o sistema cumpre as *deadlines* e nunca executa em modo de falha ou de forma insegura – e *bounded time responsiveness* – no caso de ocorrer uma falha, a ação de mitigação deve ser executada num determinado limite de tempo. O modelo utiliza uma gestão de falhas hierárquica com um *global manager*, que contém vários *regional managers*. Estes, por sua vez, contêm *local managers*. No entanto, a hierarquia formada não é totalmente descentralizada, visto que o *global manager* é um SPOF e pode eventualmente criar um efeito dominó, propagando-se verticalmente na hierarquia para os seus subdelegados. Este modelo propõe o uso da ferramenta UPPAAL, cuja finalidade consiste em verificar modelos para redes de autómatos temporizáveis.

Khalouzadeh *et. al*, 2010, propõem no seu artigo [15] um algoritmo que encontra uma solução ótima na formação de coligações de agentes homogéneos. Na sua proposta, cada agente tem um vetor de capacidades baseado nos seus recursos, e cada *task* (componente) tem um vetor de requisitos. Neste caso, como os agentes inteligentes consistem em *robots*, a sua relação com as *tasks*, é MR (*Multi-Robot task*), isto é, existem vários agentes a executar uma tarefa com concorrência de acesso. No caso de um ou mais requisitos de uma determinada *task*, não serem preenchidos, isto é, a *task* ser executada dentro da *deadline*, esta é temporariamente descartada e vai para o fim da lista de tarefas. Para a finalidade desta investigação, esta abordagem para não é conveniente, na medida em que existem *tasks* dependentes umas das outras. Sendo dependentes entre si, recebem como *input* o *output* das precedentes, logo não é possível ignorar uma tarefa; esta tem obrigatoriamente de ser executada, numa estratégia NGU. Para além disso, na primeira fase do algoritmo, a coligação requere que haja troca de informação entre os agentes e que, a partir destes, sejam criadas tabelas, com os IDs dos agentes, a posição (x, y) no meio, os seus vetores de capacidades - o que os autores referem como a "distância" de cada agente à *task* em questão. Este processo atrasa a execução dos componentes e pode tornar-se inviável para o cumprimento das *deadlines*.

Em [4], Guerrero *et. al*, 2012, propõem um modelo de formação de coligações multi-*robot*, para autómatos físicos, através da já mencionada abordagem MRTA. Embora seja numa área de tolerância falhas diferente, para *robots* físicos, a conceção do modelo traz

algumas novidades e implementações interessantes em relação a outros trabalhos. A relação de agentes para *tasks* utilizada, segundo a taxonomia *robot-task* anteriormente referida, é do tipo $ST - MR - IA$. Neste modelo, a alocação de uma tarefa numa determinada coligação, só é válida se todos os *robots* pertencem a essa coligação e só executam essa tarefa. A utilidade (*utility*) de uma tarefa é semelhante ao conceito de *reward*, nos esquemas de licitação por tarefas, e varia conforme o tipo de *deadline* (ver 2.3.2). É notório que cria-se um esforço no sentido de tentar prever o tempo de execução das várias tarefas com *deadlines*, através da conjugação do método SVR (*Support Vector Regression*) e DR (*Double Round auction*). No método de atribuição de tarefas, cada agente propõem *bids* pelas *tasks*. Em termos de cálculo de variáveis como carga de trabalho, $taskWorkLoad_j$, não é aplicável em *software*, pois envolve distâncias (m) e remete para ambientes meramente físicos.

Nogueira *et. al*, 2010, em [16], partindo do pressuposto que o sistema é distribuído, aberto e dinâmico, introduzem um modelo de tolerância a falhas através da utilização de um determinado número de réplicas passivas n_{C_i} para cada componente c_i. O número de réplicas calculado é diretamente proporcional ao grau de significância de um componente c_i e inversamente proporcional à soma de todos os graus de significância dos componentes que compõem um serviço S. Esta proposta é interessante na medida em que apresenta um modelo com formação de coligações, coordenado, com adaptação local e eleição de novas réplicas primárias. No entanto, a criação de demasiadas réplicas, ainda que passivas, não traz benefícios à performance do sistema, nomeadamente para sistemas com pouca capacidade de armazenamento. Parte do funcionamento algorítmico e alguns dos conceitos deste modelo, completado em [17] e em [18], são aproveitados e adaptados ao modelo a ser desenvolvido neste documento.

Relativamente a outras ferramentas utilizadas na tolerância a falhas, de acordo com Kumar *et. al*, 2011, as melhores abordagens usadas para explorar o aspeto da redundância são TERCOS e DEBUS [19]. A primeira melhora significativamente o aspeto do escalonamento/redundância até 17% e conjugada com o DEBUS pode resultar em melhoramentos adicionais até 12% sobre a percentagem obtida inicialmente.

Em [20], Gong *et. al*, 2002, apresentam um trabalho de investigação acerca de formação de coligações, utilizando sistemas baseados em agentes, mais especificamente MAS (*Multi-Agent System*). Neste trabalho, os autores propõem um algoritmo para sistemas dinâmicos com restrições ao nível do tempo (*time-bounded*) e dos recursos. Os melhores agentes são selecionados pelas *capability organizations* para executar uma tarefa. Segundo os autores, uma *capability organization* é uma entidade responsável pela coordenação de coligações. O desenvolvimento do algoritmo foca-se na procura de coligações que beneficiem a solução, mas não tenta otimizar gradualmente a qualidade de serviço que é fornecida.

Shehory e Kraus, 1993, debruçam-se profundamente sobre a formação de coligações em [21] e apresentam algumas definições e axiomas. Nas definições introduzem o termo "*su-*

per-additive environment" que se refere a um ambiente em que duas coligações C_1 e C_2 se juntam formando uma nova coligação. O modelo proposto define também que os agentes têm um vetor de recursos, assim como a coligação que estes formam. O vetor de recursos é modificado após a distribuição de quantidades de recursos.

No que diz respeito às abordagens de tolerância a falhas ao nível da infraestrutura de rede, podem ser utilizadas técnicas como a *restoration* - *e.g.* RSVP-TE (*Resource reSerVation Protocol for Traffic Engineering*) e *IP layer dynamic rerouting* -, que consiste em procurar dinamicamente caminhos de *backup* que economizam a capacidade da rede, sob a ocorrência de falhas de rede, ou como a *protection* – *e.g.* MPLS e *Optical restoration* -, que consiste em reservar inicialmente caminhos secundários dedicados, bem como recursos na rede. No trabalho de investigação de Valcarenghi *et. al*, 2008, é proposto um esquema de tolerância a falhas em redes de fibra-ótica baseadas no protocolo *Generalized Multi-Protocol Label Switching* (GMPLS) [22], resiliente em termos de *QoS-awareness*. Nesta proposta, é utilizada uma técnica de restauração dinâmica (*dynamic restoration*), com o auxílio do LSP (*Label Switched Path*), que garante largura de banda. Existe ainda um *resource broker*, localizado num dispositivo específico da rede, que interage com o cliente, que consome o serviço, com o servidor, que fornece o serviço, e com os dispositivos de rede LSR (*Label Swtiched Routers*). O *resource broker* atua de maneira a manter a largura de banda, através de *recovered connections*. Em último lugar, este esquema utiliza heurísticas para colocação de réplicas na rede: MAX-FANOUT, HOP e SUPER-NODE DEGREE. Apesar de este trabalho sugerir esquemas e ideias interessantes, particularmente as três heurísticas para replicação de serviço, desvia-se do âmbito do trabalho, pois direciona-se unicamente para a tolerância a falhas baseada em redes (*network-based fault tolerance*).

Atualmente existem diversos trabalhos de investigação na área da tolerância a falhas que se debruçam sobre formação de coligações para atingir uma configuração benéfica para o serviço, de forma cooperativa. Considerando a natureza distribuída dos sistemas da atualidade, em *cloud computing*, a grande maioria dos modelos propostos visam a distribuição de tarefas pelos *hosts* da rede a fim de aproveitar recursos inutilizados e de tornar eficiente o balanceamento de carga na totalidade da rede de comunicação.

3 Modelo de Tolerância a Falhas

Este capítulo retrata o desenvolvimento do modelo e todas as suas componentes: técnicas de tolerância a falhas, arquitetura do modelo, sistema baseado em agentes, algoritmos de decisão na formação de coligações. São descritas também, relativamente a cada uma das características, as vantagens da aplicação do modelo a sistemas de tempo-real e no final é feita uma síntese conclusiva acerca do modelo.

3.1 Descrição do ambiente

De uma forma introdutória, quando uma máquina com poucos recursos pretende executar uma determinada aplicação em tempo-real, que é caracterizada por ser distribuída e baseada em componentes (CBSE), deve solicitar uma execução cooperativa com outras máquinas da rede LAN onde se encontra. O modelo desenvolvido deve permitir que uma máquina com essas características peça auxílio na execução dessa aplicação. Deve ainda ter a capacidade de tolerar, externamente, falhas que ocorram nesses componentes. Para isso deve recorrer a técnicas de tolerância a falhas que ocultem ou tornem transparentes os erros aplicacionais que possam verificar-se para o resultado. Convém recordar que em ambientes *real-time* uma falha pode resumir-se ao não cumprimento de uma *deadline*.

O modelo proposto tem um conjunto de pressupostos. Estes podem ser entendidos como assunções, que são concebidas no arranque do desenvolvimento deste trabalho e influenciam a forma como este vai evoluir. O modelo a desenvolver deve estar em conformidade com uma estruturação concreta do sistema e com uma descrição de qualidade de serviço específica, que são referidas nos próximos tópicos.

Este modelo atua num estilo estratégico NGU, o que significa que os mecanismos de tolerância a falhas implementados devem induzir constantemente o sistema num estado correto, removendo ou mascarando faltas existentes no sistema. A estratégia NGU desenvolvida para este modelo é composta por comportamentos de tolerância a falhas que permitem a reconfiguração da parte do sistema que executa incorretamente. O esquema EDRB

baseado em agentes, conjugado com a *QoS-awareness*, constitui a maioria dos mecanismos utilizados. Considera-se que as faltas raras (*rare events/faults*) podem ocorrer ou devido a um erro interno da aplicação de tempo-real ou quando uma *deadline* não é cumprida pelo bloco primário nem pelo alternativo.

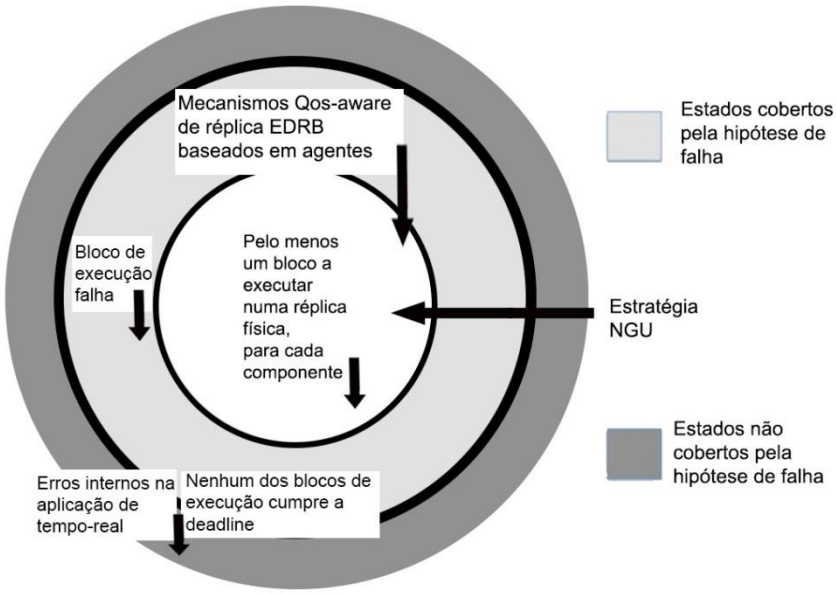


Figura 3.1 – Espaço de estados do modelo

A Figura 3.1 representa o espaço de estados do modelo onde são enquadradas as faltas normais e raras e os mecanismos de tolerância a falhas. Concretamente, a estratégia NGU consiste na constante reativação dos componentes, por parte dos agentes, em caso da ocorrência de uma falta do tipo normal. Este estilo de atuação do modelo vem ao encontro dos requisitos de tempo-real através da inserção do esquema EDRB que contém um teste de aceitação, que determina a validade do resultado produzido. O AT também define valores de tempo exatos para o cumprimento da *deadline*. Para além disso, o esquema EDRB utiliza replicação ativa, como forma de aplicar redundância na execução de um componente do sistema.

3.1.1 Estrutura do sistema

Os sistemas computacionais sob os quais incide este modelo de tolerância a falhas são do tipo distribuído (RTDS). São também embebidos e dinâmicos, *i.e.* com uma alocação de recursos não estática, devido à necessidade de ocorrerem migrações de componentes. Os sistemas devem também ser caracterizados por serem de tipo aberto, *i.e.* configuráveis em tempo de execução, o que permite que sejam facilmente reparametrizados. Posteriormente deve ser permitida a sua integração com outras aplicações, por exemplo de suporte à performance ou outras que assegurem a estabilização da sua *dependability*.

Considerando este tipo de características dos sistemas-alvo, pretende-se desenvolver um modelo que preencha requisitos como a descentralização e a autonomia, sendo dotado de uma alta flexibilidade e dinâmica. Deve conseguir-se adaptar a novas circunstâncias, previstas ou imprevistas, tendo em conta a alta instabilidade e imprevisibilidade na alocação de tarefas do sistema nas máquinas da rede, pois há sempre a possibilidade de surgimento de diversos tipos de erros no sistema. O modelo proposto deve ainda coordenar todas as mudanças efetuadas a cada um dos módulos do sistema, para corresponder às necessidades de configuração dinâmica e deve basear-se numa arquitetura genérica para corresponder à heterogeneidade dos ambientes em que irá ser envolvido.

Foram seguidas diversas assunções acerca de algumas estruturas de informação com que o modelo lida, com o intuito de simplificar o funcionamento do modelo. Procedeu-se à estandardização de alguns tipos de dados, *e.g.* os níveis e dimensões de QoS, e de alguns métodos de comunicação, *e.g.* tipos de mensagens trocadas.

Assunção 1 Comunicação

Assume-se que existem vários métodos de comunicação, de maneira a que os agentes possam negociar e realizar contratos [23].

O protocolo de comunicação entre agentes deve estandardizar os diferentes tipos de interações que estes possam realizar. A uniformidade nos diálogos é um passo importante para o funcionamento salutar da coordenação e sincronização, apesar da diversidade dos agentes.

Assunção 2 Serviço baseado em componentes

Assume-se que o sistema de tempo-real consiste num serviço $S = \{c_1, c_2, \dots, c_n\}$ como um conjunto de componentes de software que são executados cooperativamente por uma coligação formada por nós [17]. Cada componente c_i é definido pela sua funcionalidade e, sendo capaz de enviar e receber mensagens, é alcançável num certo ponto da rede [16]. Assume-se que o serviço S é representado por um grafo acíclico dirigido (DAG) $\mathcal{G}_S = (v_s, \varepsilon_s)$, que contém todas as interdependências de QoS entre componentes $c_i \in S$. Cada vértice $v_i \in v_s$ representa um componente c_i e um arco dirigido $\varepsilon_i \in \varepsilon_s$ de um componente c_j para c_k indica que c_k é funcionalmente dependente de c_j. Ou seja, um componente c_i diz-se funcionalmente dependente de um outro c_j, se o *output* deste último for *input* para c_i [18] [16]. No grafo $\mathcal{G}_S$ chama-se «*cut-vertex*» a um componente $c_i \in v_s$ se este no caso de ser removido, separar $\mathcal{G}_S$ em dois grafos conexos desunidos [18]. Um componente $c_i \in S$ está na expetativa de receber um conjunto de *inputs* $I_{c_i} = \{(c_j, Q_{val}^j), \dots, (c_k, Q_{val}^k)\}$, que descrevem a qualidade de *input* fornecida pelos seus componentes predecessores em $\mathcal{G}_S$, e de enviar um conjunto de *outputs*

$O_{c_i} = \{(c_l, Q^l_{val}), \dots, (c_m, Q^m_{val})\}$, que descrevem a qualidade de *output* fornecida aos seus sucessores em $\mathcal{G}_S$. De notar que Q^i_{val} refere-se à qualidade de serviço produzida por c_i.

É importante referir ainda o conceito de grau de significância de um componente, utilizado na coordenação do sistema através de formulação de propostas (ver 3.4). Dado um DAG $S_{\mathcal{G}} = \{\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_n\}$ como o conjunto de grafos conectados de interdependências entre componentes de um dado sistema, e $O_{\mathcal{G}_j} = (c_i)$ como o grau de saída de um nó $c_i \in \mathcal{G}_j$, o grau de significância σ_i de c_i representa-se através da equação (3.1) [16].

$$\sigma_i = \sum_{k=1}^{n} O_{\mathcal{G}_k}(c_i) \tag{3.1}$$

O grau de significância (σ_i) de um componente c_i representa a influência de c_i para um resultado final percebido pelo utilizador. Quanto maior for o grau de significância de um componente, maior é o somatório das suas ligações de saída e de todas as dos seus sucessivos nós, nos grafos $\mathcal{G}_1, \mathcal{G}_2, \dots, \mathcal{G}_n$, logo maior é o seu peso na execução do serviço. Evidentemente que um componente inicial $c_1 \in S_{\mathcal{G}}$ tem o maior grau de significância $\sigma_1(c_1)$ da totalidade do grafo conexo $S_{\mathcal{G}}$ formado por n componentes, com $n > 1$.

Sistemas baseados em componentes (CBSE) são o resultado de um género de desenvolvimento de *software*, cujo objetivo é separar os diferentes interesses da aplicação em componentes. Neste tipo de sistemas, é suposto fornecer-se um serviço amplo em termos funcionais, integrando normalmente uma arquitetura SOA. Um componente deve ser completamente independente em termos de *design* e é parcial ou totalmente independente em termos de execução. Cada um representa um módulo do serviço que executa uma tarefa específica e tem uma determinada influência no resultado final produzido, mediante o seu grau de significância. As ligações entre os componentes são as entradas e saídas de dados. Este tipo de estrutura permite que o modelo produza uma distribuição equitativa de *workload* pelos nós da rede. O funcionamento do modelo é auxiliado por algoritmos *anytime* que, por sua vez, distribuem os componentes pela rede e tentam estabelecer contratos de execução com qualidades de serviço específicas (SLAs).

3.1.2 Configuração de QoS

A qualidade de serviço (QoS) lida com aspetos fundamentais dos sistemas, tais como *timeliness*, a robustez, a *dependability* e principalmente a performance. É, decisivamente, um fator determinante na eficiência da execução em *real-time embedded systems*. Aplicações que sejam caracterizadas como *QoS-aware* estão cientes dos níveis de QoS fornecidos aos vários componentes da aplicação e desempenham a importante missão de acordar níveis de qualidade específicos, através de SLAs, e, com isto, operam como mecanismos de QoS que oferecem suporte à tolerância a falhas. Adicionalmente, estas aplicações devem re-

servar determinadas quantidades de recursos para fornecer níveis estáveis de qualidade de serviço, monitorizando continuamente a disponibilidade dos dispositivos em rede.

Em termos de parametrização, cada componente pode ser parametrizado por valores QoS que definem a qualidade que deve produzir. A configuração da qualidade de serviço influencia a qualidade do *output* produzido por um determinado componente e, consequentemente, a qualidade final do serviço fornecida ao utilizador.

A característica de *QoS-awareness* é intrínseca à tolerância a falhas. Caracterizado como *QoS-aware*, o sistema reage às alterações no meio, adaptando-se através da aplicação de reconfigurações de QoS mais apropriadas aos componentes. Assim, é mais provável que a execução global do serviço não falhe. Exemplificando, na prática, se um sistema *non-QOS-aware* estiver a fornecer um nível de QoS mais alto do que o que seria correto a um componente, este pode não devolver o seu *output* a tempo ao seu sucessor ou, no caso de ser o último, ao utilizador. Já num sistema *QoS-aware*, efetuando um *downgrade* à qualidade configurada no componente, consegue-se certamente obter um resultado mais atempado, apesar de mais degradado. Em suma, é possível depreender que o cumprimento das *deadlines*, que traduz a ocorrência de faltas num *real-time software*, está dependente, ainda que não totalmente, das configurações de QoS efetuadas, ou seja, da característica de *QoS-awareness*.

Assunção 3 Descrição da qualidade de serviço

Assume-se que cada componente $c_i \in \mathcal{G}_S$ tem um conjunto de parâmetros de QoS que podem ser alterados para adaptar o fornecimento de serviço a um ambiente dinâmico e mudável. A cada subconjunto de parâmetros QoS que se referem a um único aspeto de qualidade de serviço dá-se o nome de «*dimensão de QoS*». Cada uma destas dimensões tem diferentes requisitos de recursos (consumo de CPU e RAM) para cada combinação possível de qualidade de serviço. Assume-se que execuções com níveis qualidade de serviço mais altos consomem maior quantidade de recursos. O utilizador pode fornecer uma especificação única através da definição de um intervalo de níveis de QoS aceitáveis, parametrizando a totalidade de um serviço S. Dados diversos componentes $c_i \in S$, o intervalo definido deve conter valores entre o nível de qualidade mínimo até ao nível de qualidade desejável, respetivamente $L_{minimum}$ e $L_{desired}$, obtendo-se a expressão $L_{minimum} \leq L^i_{current} \leq L_{desired}$. Cada utilizador não necessita de compreender cada c_i [16].

Relativamente ao esquema genérico onde é especificada a qualidade de serviço, assume-se que esta é descrita formalmente com conjuntos de domínio: dimensões, atributos e valores, sendo que o espaço vetorial do domínio QoS é representado pela expressão $Qos = \{Dim, Atrib, Val, DA, AV, Dep\}$. Mais concretamente, existe um conjunto de m atributos de QoS $Atr = \{atr_1, atr_2, \dots, atr_m\}$ de um serviço S, cujos valores são tirados dos domínios $D = \{D_1, D_2, \dots, D_n\}$. Cada atributo $attr_m$ tem um conjunto $Val =$

$\{val_1, val_2, \ldots, val_o\}$ de valores possíveis. Cada valor é representado pelo seu tipo e domínio: $val_i = \{Tipo, Domínio\}$, onde o tipo é descrito por $Tipo = \{integer, float, double, string\}$ e o domínio é representado por $Domínio = \{contínuo, discreto\}$. Relativamente às relações entre domínios, o conjunto DA_r denota uma atribuição de uma dimensão Dim_n a um conjunto de atributos Atr, que reproduz a relação Dimensão-Atributos, e é definido como $DA_r : Dim_i \rightarrow Atr, \forall\, Dim_i \in Dim$. A segunda relação, Atributo-Valores, atribui a cada atributo Atr um valor específico Val e é representada pela expressão $AV_r : Atr_i \rightarrow Val_k, \forall\, Atr_i \in Atr, \exists^1_{Val_k} \in Val$. Quanto ao domínio Dep, especifica um conjunto de dependências existentes entre os valores dos atributos e é representado como $Dep_{ij} = d(Val_{ki}, Val_{kj}), \forall\, Atr_k \in Atr$. Esta função de relação entre dois valores (Val_{ki} e Val_{kj}) de um domínio de um atributo Atr_k define que uma tarefa só oferece um dado nível de QoS desde que um outro nível de QoS específico seja oferecido por outras tarefas ou pelo ambiente [17].

O modelo proposto não aborda a estruturação da informação relativa às dependências entre atributos.

A Tabela 3.1 ilustra um exemplo de especificação QoS para uma aplicação de *streaming* de vídeo e áudio.

Campo	**Valores possíveis**
QoS Dimensions	{Media Container,Video Quality, Audio Quality}
Media Container	{container format}
Video Quality	{color depth, frame size, frame rate}
Audio Quality	{sampling rate, sample bits}
Container format	{3GP, ASF, AVI, QuickTime, RealVideo, WMV}
Color depth (*bits*)	{1, 3, 8, 16, 24}
Frame size (*pixels*)	{240x180, 320x240, 640x480, 720x480, 1024x768, 1280x1024}
Frame rate (*per second*)	{[1,30]}
Sampling rate (kHz)	{8, 11, 32, 44, 88}
Sample bits (*bits*)	{4, 8, 16, 24}

Tabela 3.1 - Exemplo de lista genérica de dimensões QoS, atributos e valores possíveis

[17]

Uma descrição QoS com esta estrutura dá a possibilidade tanto aos utilizadores como aos fornecedores de serviços de, num determinado domínio aplicacional, definirem requisitos e preferências, ou seja, níveis mínimos e níveis desejados. Também permite a formulação de propostas (ver 3.4.1) para contratualizar o fornecimento de níveis de qualidade, acertados entre as entidades envolvidas.

Associados à Tabela 3.1, a Tabela 3.2 mostra, numa perspetiva de descrição formal da qualidade de serviço, o formato genérico das dimensões, atributos e valores possíveis.

Conjunto	**Valores possíveis**
Dim	{Video Quality, Audio Quality}
Attr	{compression index, color depth, frame size, frame rate, sampling rate, sample bits}
Val	{{1,integer,discrete},{3,integer,discrete}, ..., {[1,30],integer,continuous},...}
DA Video Quality	{image quality, color depth, frame size, frame rate}
DA Audio Quality	{sampling rate, sample bits}
AV compression index	{[0,100]}
AV frame size	{SQCIF,QCIF,CIF,4CIF,16CIF}
AV color depth (*bits*)	{1,3,8,16,24,...}
AV frame rate (*per second*)	{[1,30]}
AV sampling rate (kHz)	{8,11,32,44,88}
AV sample bits (*bits*)	{4,8,16,24}

Tabela 3.2 - Exemplo de associação conjunto-valores da descrição de QoS

[17]

O processo de pedido de auxílio na execução de S levará, posteriormente, à formação de uma coligação de agentes instalados nos nós aderentes a um *membership service* (ver 3.4).

Com esta descrição formal de QoS uniforme, disponível para todas as máquinas participantes na formação da coligação, é possível estabelecer um diálogo consensual, unívoco e percetível no seio de um sistema para desenvolver iterativamente a coligação formada, isto é, melhorar a qualidade produzida. Exemplificando, na Tabela 3.3, para um sistema de vigilância remota um utilizador pode facilmente definir a os atributos *compression index* e *frame size* como prioritários relativamente a outros como *frame rate* e *color depth*, para obter uma qualidade de imagem melhor, ainda que com uma velocidade de apresentação mais degradada [17].

	Video Quality			**Audio Quality**	
	compression index	frame rate	color depth	sampling rate	sampling bits
Valores	{[0,20]}	{[5,1]}	{3,1}	{11,8}	{8,4}

Tabela 3.3 - Exemplo de hierarquia de descrição QoS

3.2 Esquema baseado em EDRB

O modelo desenvolvido baseia-se no esquema EDRB, proposto por Hecht *et al.*, em 1991, com algumas modificações ao nível da hierarquia dos nós integrantes, referidas nos tópicos seguintes. Como o nome indica, a ideia fundamental do RB é ter um bloco de código de recuperação num nó para o caso de ocorrer uma falha.

Para munir o modelo com descentralização foi retirado o nó supervisor, restando os nós operacionais. Recorde-se que o supervisor era um SPOF no esquema original, pois era responsável pela reinicialização dos nós faltosos. Neste caso, os nós operacionais serão os próprios coordenadores, funcionando num sistema *peer-to-peer*. O esquema aplicado consiste em conjuntos de pares de nós que podem ser vistos como um só relativamente à sua função, pois têm o objetivo comum de suportarem a execução do serviço, tolerando falhas. Cada par de nós deve coabitar numa máquina com a principal função de colmatar as falhas mutuamente, para cumprir todas as restrições temporais impostas, com resultados atempados, ainda que com uma qualidade de serviço degradada.

Tanto os nós físicos como os lógicos trocam *heartbeats* em intervalos curtos para verificarem se o parceiro de execução se encontra ativo e para terem perceção do seu papel na execução, ou seja, se devem atuar como réplica primária ou secundária. Este método de sincronização acaba por não criar *overhead* significativo, dado que o volume das mensagens é extremamente reduzido[5]. Para além disso, como a comunicação é efetuada num

[5] O conteúdo de uma mensagem FIPA-ACL do tipo INFORM contém uma *string* ASCII de 7 bytes.

estilo ponto-a-ponto os componentes do modelo tornam-se autónomos e menos fulcrais para a execução de um componente.

Cada um dos nós conterá dois blocos de execução - um primário e um alternativo – e um teste de aceitação AT. Este último avaliará o cumprimento da *deadline* através do *hearbeat* definido (arquitetura TT). Se um nó não realizar a troca de *heartbeats* dentro dos limites definidos, então é considerado um membro congestionado e, portanto, faltoso. Assume-se que também não conseguirá devolver o resultado produzido respeitando a *deadline*. Em caso de incumprimento da *deadline*, o nó, independentemente de ser virtual ou físico, não passará o teste de aceitação e assumirá o papel secundário na execução do componente. Conceptualmente, ambos conterão ainda uma componente denominada de *active node* no esquema, cuja função é o envio e a receção de *heartbeats*. O *active node* será responsável por averiguar se o parceiro se encontra "vivo" e de o informar acerca do estado do nó. O *active node* deve reagir, se necessário, a este diálogo despoletando mudanças de papel no seu nó.

Em cada par, um dos nós assume o papel primário, enquanto o seu parceiro assume o papel secundário. Cada um dos nós é constituído por duas versões do bloco de execução, uma primária e outra alternativa, do respetivo componente. A única diferença entre a réplica primária, também chamada *active*, e a secundária, ou *shadow*, é a forma como os seus blocos de execução são parametrizados.

A configuração de QoS nas réplicas de um *host* é assimétrica. Genericamente, uma réplica primária r_1^i de um componente c_i executa a sua versão primária do bloco de execução com um nível de qualidade Q_{val}^i, que é fornecido em função da *availability* da máquina em que se encontra. A sua versão alternativa consiste numa imagem idêntica do bloco primário mas com um nível de QoS inferior Q_{val+1}^i. Isto permite prevenir uma falha causada pela parametrização errada do componente, pois este pode eventualmente falhar a sua execução devido a uma configuração com um nível de QoS excessivo. Com isto a qualidade de *input*, se ainda estiver no intervalo que o utilizador definiu ($L_{minimum} \leq L_{current}^i \leq L_{desired}$), esta será degradada para o nível imediatamente inferior.

Por outro lado, numa segunda réplica secundária r_2^i, que executa o mesmo componente, a versão primária é executada com um nível de QoS inferior, praticando um nível de QoS equivalente ao configurado no bloco alternativo da réplica primária (Q_{val+1}^i). Já o seu bloco alternativo possui a mesma configuração do bloco primário da réplica primária (Q_{val}^i). O processo descrito é representado na Figura 3.2. Recorde-se: é assumido que os níveis de QoS são especificados por ordem decrescente. Daí a réplica secundária ser configurada com Q_{val+1}^i.

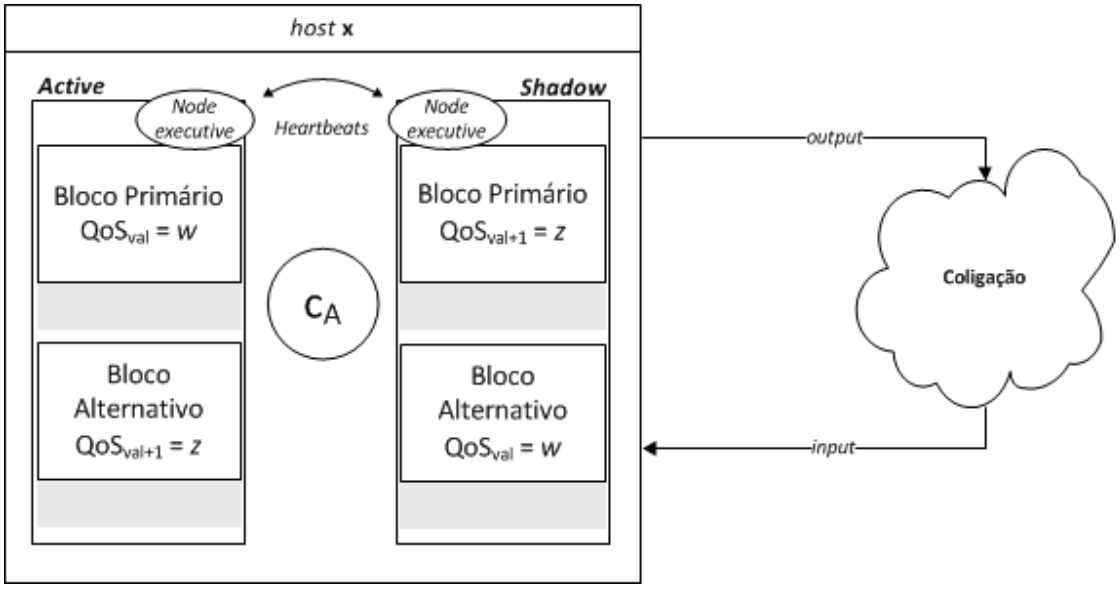


Figura 3.2 – Composição genérica EDRB num *host* da coligação

Nesta figura, w e z são os valores de QoS executados por cada bloco de execução dos nós virtuais, *active* e *shadow*. Note-se que a configuração entre estes é invertida. A ligação do *input* ou do *output* à coligação é opcional, visto que em alguns casos pode não existir, conforme haja dependência funcional do componente c_A executado pelo *host* x para outros componentes.

O nó *shadow* substituirá o nó *active,* em caso de falha, e assumirá o papel de nó primário, enquanto o nó *active* passará a desempenhar o papel de nó secundário. A aplicação do esquema ao nível do *software* repercute-se ao nível do *hardware*. Assim sendo, este esquema permite tolerar falhas tanto ao nível do *hardware* como do *software*, pois a replicação de um mesmo componente pode ser feita na mesma máquina, isto é, virtualmente, mas também em outra, isto é, fisicamente.

Utilizando tolerância a falhas com dois âmbitos distintos, através de replicação ativa, consegue-se aumentar a área da *fault hypothesis*. Em caso de paragem de processamento num nó físico ou virtual, a réplica secundária (*hardware* ou *software*) é promovida a primária, ao mesmo tempo que a réplica primária reinicia e assume o papel de réplica secundária.

3.2.1 Replicação ativa

A criação de réplicas físicas tem como principal objetivo proporcionar resultados aceitáveis, em termos qualitativos e temporais, tolerando qualquer tipo de falhas de *hardware*, *i.e.* lida com falhas de rede, físicas, de media e de processadores (ver 2.2.1). Em ambientes *cloud computing* ou similares as falhas físicas são mais prováveis de acontecer, *e.g. link down* ou *resource failure*. Esse tipo de vulnerabilidades é colmatado com a replicação físi-

ca enquadrada no esquema EDRB descrito. A replicação virtual tem os mesmos objetivos do que a física. No entanto, a sua aplicação é direcionada para a tolerância de falhas ao nível do *software*, *i.e.* permite tolerar falhas de processos e de expiração de serviço. Cada uma das réplicas físicas contém duas réplicas virtuais, que consistem em dois agentes móveis que executam concorrentemente um determinado módulo do serviço.

Considerando o momento em que a coligação converge, existem duas réplicas físicas (*hardware*) em cada dispositivo aderente à coligação. Os nós físicos trocam *heartbeats* entre si tal como os nós virtuais.

A escolha de uma réplica física que seja superior à inicial, em termos de capacidade de processamento, é gradual, pois os agentes de reconhecimento (ver Definição 1) continuam a inspecionar outras máquinas. Eventualmente, estas podem ser capazes de propor melhores ofertas e, caso isso se confirme, são selecionadas para se juntarem à coligação, visto que contribuem para uma melhor qualidade de serviço a nível geral.

No que diz respeito às réplicas físicas, existem diversos aspetos delicados. O número de réplicas físicas secundárias existentes numa coligação depende estritamente da *availability* das máquinas aderentes à coligação. Mesmo com poucas máquinas, se estas tiverem uma alta disponibilidade é dispensável a adesão de outras. Até estar completa, durante a formação da coligação a alocação dos componentes nas máquinas é imediata e prioritária, partindo dos componentes com maior grau de significância. Isto significa que a seleção e a atribuição de réplicas físicas primárias aos componentes é uma tarefa prioritária. Só depois se procede à seleção de réplicas físicas secundárias.

Uma determinada máquina pode ser uma réplica física primária para um componente e para um outro ser a réplica secundária, caso possua a *availability* necessária. Daí se uma máquina que é uma réplica física primária para um componente c_A for interrogada para ser réplica física secundária de outro componente c_B mas esse não é *feasible*, então o nível de QoS fornecido a c_A é degradado para um nível aceitável pelo utilizador. Contudo, caso o nível de c_A seja o mínimo então a execução de c_B nessa máquina é descartada.

Se uma réplica física secundária de um componente c_A falhar, então procede-se à busca de uma réplica física secundária. Em função da *availability* dos nós da rede, seleciona-se a melhor máquina da coligação.

A Figura 3.3 mostra um exemplo de uma rede com um conjunto de máquinas, em que se encontram um dispositivo emissor, *host* x, e um recetor, *host* y. As sete máquinas da rede mostraram-se disponíveis e aptas para participar na formação de uma coligação, auxiliando um *host* inicial *x* a executar um serviço, entregue a um *host* final *y*.

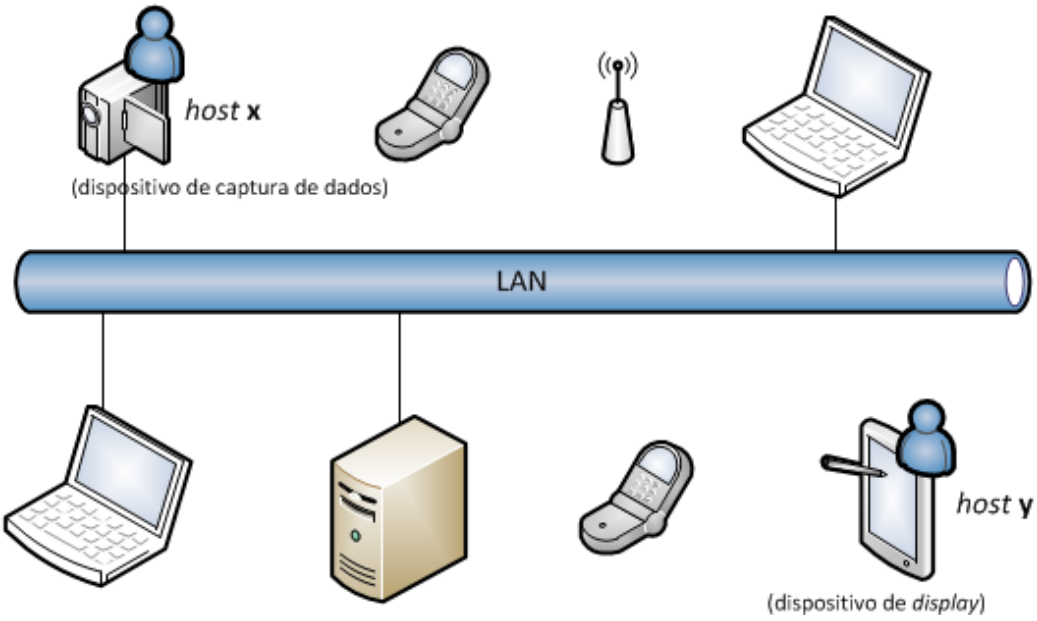


Figura 3.3 - Exemplo de uma rede apta para execução cooperativa

Para efeitos de produção de dados de saída, no caso de os respetivos componentes originarem *output*, as réplicas primárias enviam-no para as réplicas primárias que executam os seus componentes funcionalmente dependentes, ou seja, os seus sucessores. Ao mesmo tempo, as réplicas secundárias enviam o *output* para as réplicas secundárias com componentes sucessores. Desta forma, criam-se dois ciclos de *software* completamente independentes: um primário e um secundário.

Por vezes, devido à falta de máquinas aderentes à coligação, nem todos os componentes conseguem ser executados em duas réplicas físicas. Claramente, o ciclo natural de *software* é destabilizado, no que toca às entradas e saídas de dados. Dada uma réplica física primária n_{B^1} que executa um componente c_B, replicado também num nó físico de *backup* n_{B^2}, e um componente c_A predecessor de c_B, que não é executado numa réplica física de *backup*, o *output* de c_A, enviado por uma réplica física primária n_{A^1} para n_{B^1}, é depois reenviado por n_{B^1} para a sua parceira de *backup* n_{B^2}. Por outras palavras, qualquer réplica primária de um componente cujo antecessor não tem réplica secundária, retransmite o *input* recebido para a sua réplica secundária, sejam estas do tipo físico ou virtual.

Para exemplificar o processo descrito no parágrafo anterior, na Figura 3.4 é ilustrado um esquema com uma coligação, com base na rede apresentada na Figura 3.3. O grafo descreve um serviço formado por cinco componentes, de A a E, que são, juntamente com as máquinas, representados pelos nós do grafo. Os ramos consistem em túplos com três valores que contêm a qualidade de *output* produzida do componente a executar numa determinada máquina. Como se pode ver, apenas os componentes c_B e c_D têm réplicas físicas secundárias, respetivamente n_{B^2} e n_{D^2}. Como c_A não executa numa réplica física de *backup*, quando o seu *output* é enviado para c_B, n_{B^1} replica-o e reenvia-o para n_{B^2}, com

um QoS mais degrado, Q^{A}_{val+1}, que seria o produzido por uma réplica secundária n_{A^2} caso existisse. De notar que o *output* de c_C também é reenviado por n_{D^1} para n_{D^2}.

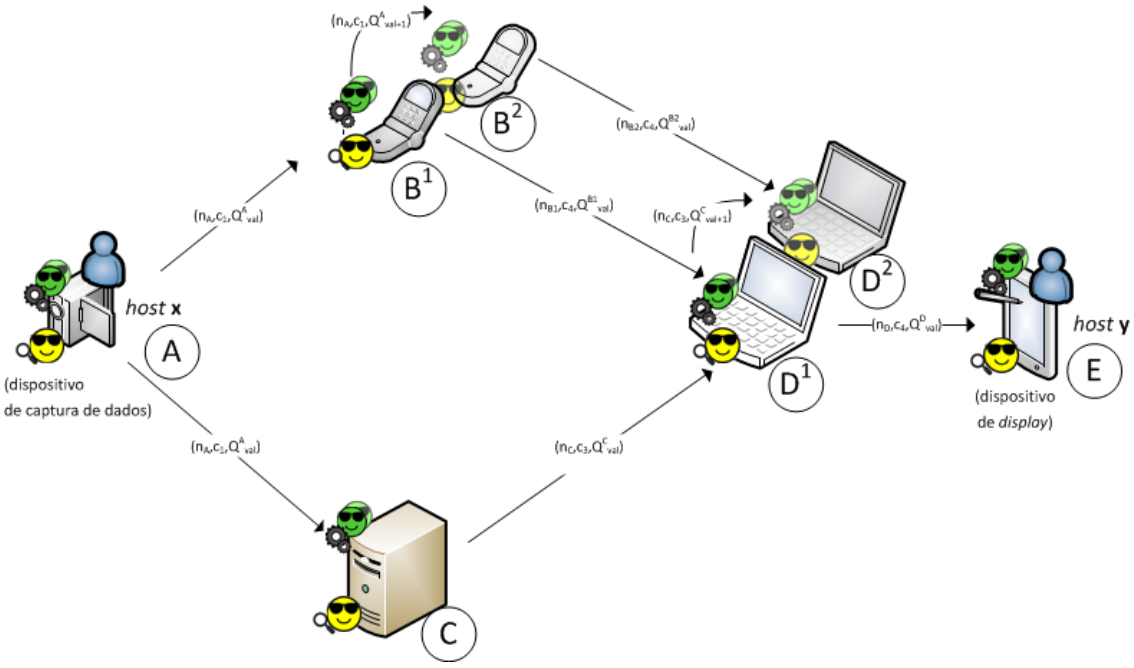


Figura 3.4 – Exemplo do funcionamento de um serviço numa coligação

Ainda relativamente à Figura 3.4, numa situação hipotética em que c_A executaria, para além de em n_{A^1}, também em n_{A^2}, então n_{A^2} seria responsável por enviar o *output* O_{c_A} para n_{B^2}, com um nível Q^{A}_{val+1}. Nesse caso, n_{B^1} não teria de repercutir o *output* para n_{B^2}. De referir ainda que cada máquina tem ainda um agente de reconhecimento e dois agentes de execução (ver Definição 2), um primário e um done, que correspondem às réplicas virtuais. O funcionamento do modelo integrado num MAS é abordado detalhadamente no tópico seguinte (ver 3.3). Nesta figura, os componentes B e C são paralelos (ver Definição 5) e D é um *cut-vertex*.

A chegada de novas máquinas possibilita o aumento da qualidade do serviço fornecido, isto é, oferecem mais possibilidade de escolha e mais capacidade para fornecer ao serviço níveis de QoS mais próximos dos desejados pelo utilizador. Isso traduz-se numa maior tolerância a falhas, pois torna-se maior o alcance de um número de máquina suficientes para criar *backups* para cada componente. Assim, é mais simples delegar funções suplementares para cada uma das máquinas que já se encontram encarregues de execuções com papéis primários. A probabilidade do serviço ser interrompido é muito menor e, ainda que esta se verificasse, seria reduzida para proporções insignificantes.

No entanto, existem várias medidas que são necessárias de adotar para sincronizar os estados entre réplicas:

- Sempre que há uma alteração do *input* recebido pela réplica primária física, a réplica física secundária, como não receberá diretamente os dados do nó onde executa o componente antecessor, necessita de uma sincronização entre máquinas, despoletada pela réplica primária física. O conjunto de execuções distribuídas forma um ciclo de *software* θ. Esta sincronização é absolutamente necessária pois se o ciclo de *software* secundário θ_{sec} (executado pelas réplicas secundárias, físicas e virtuais) não estiver sincronizado com o ciclo da execução primária θ_{pri}, ao nível da parametrização de qualidade de serviço, a qualidade do *output* produzido pode lidar a uma execução secundária errónea. Não é expectável que esta sincronização provoque *overhead* significativo na rede;
- O mesmo acontece quando ocorre uma alteração do *input* das réplicas virtuais, embora as consequências sejam menos danosas, visto este tipo de réplicas se encontrarem no mesmo nó físico;
- Quando se efetua a escolha de um *backup* físico, dá-se a necessidade de o agente original se clonar e do clone migrar para essa máquina. Apesar de o algoritmo de escolha de réplica física tentar minimizar o impacto na performance da rede, o agente terá de transportar todo o código consigo. Para além disso, se o agente tiver de transportar os dados já processados na máquina primária, no caso de o componente a executar requerer um processamento significativo e de o espaço para armazenamento de dados ser considerável, irá haver uma sobrecarga maior na rede.

As situações desvantajosas descritas contribuem para o aumento do *overhead* na transmissão da informação do nó físico primário para o seu *backup*, sempre que há uma alteração do *input* dos nós antecessores a esse componente no grafo. No entanto, pode não se traduzir num problema de *overhead* se considerarmos que os agentes residentes nestas máquinas, sincronizam a parametrização de QoS de forma coordenada. Essa mesma parametrização consiste em dados simples, como constantes de inteiros ou variáveis booleanas, que não criam qualquer tipo de efeitos prejudiciais no funcionamento do sistema. O mesmo já não acontece quando se trata de ficheiros serializáveis. Portanto, mesmo que haja uma sincronização periódica frequente, esta não causará distúrbios no *workload* da rede ou das máquinas envolvidas na atividade da coligação. A sincronização da parametrização de uma maneira coordenada consiste no envio da atualização do novo nível de QoS para todas as máquinas-alvo, tanto primárias como secundárias.

Para além disso, este esquema não exige a criação de mais de duas réplicas para cada componente, logo não sobrecarrega a rede com uma quantidade significativa de blocos de processamento (do EDRB), que, ao invés de auxiliarem no cumprimento das *deadlines*, afetariam negativamente a performance do sistema.

3.3 Integração de MAS

O desenvolvimento de um modelo que utilize agentes independentes traz vantagens em termos de autonomia. Podem também surgir benefícios mútuos devido à partilha de re-

cursos e redistribuição dinâmica de tarefas. A integração de um ambiente multiagente no modelo permite a criação de novos agentes face à ocorrência de determinados eventos e os agentes atuam independentemente no meio. Ações como a coordenação e a sincronização de dados são realizadas através da troca direta de mensagens entre agentes. Cada agente divulga o identificador do seu componente através da exposição de um serviço consumível para outros agentes de execução.

No modelo proposto existem dois tipos de agentes: os agentes encarregues do escrutínio e da inspeção das máquinas de rede que se disponibilizam para integrar a coligação e os agentes responsáveis pela execução dos vários componentes que constituem o serviço aplicacional, posteriormente dispersos pelas várias máquinas da rede. As seguintes definições introduzem os conceitos de «Agente de reconhecimento» e de «Agente de execução».

Definição 1 Agente de reconhecimento

É denominado de agente de reconhecimento (Ar_p) a entidade responsável por encontrar um *host* da LAN onde se encontra, que se mostre disponível para entrar na coligação e para colaborar na execução de um serviço S. Este tipo de agente deve fazer a análise da disponibilidade (*availability*) dessa mesma máquina. Para isso, ao ser detetado um novo nó na coligação, inicia-se um agente de reconhecimento na máquina que pretende executar o serviço cooperativamente. De seguida, o agente migra para a nova máquina, ainda não inspecionada. Uma vez concluída a migração, o agente de reconhecimento não volta a mover-se no seu cido de vida e permanece hospedado na máquina inspecionada até que esta seja removida da coligação.

O agente calcula a *availability* com uma margem de erro tanto menor quanto maior for a quantidade de dados estatísticos relativos a falhas ocorridas nessa máquina.

Exemplo:

As equações (3.2) e (3.3) mostram o cálculo do FIT e do MTTF de 2 discos, dispondo de informação de falhas relativas a 500.000 horas de MTTF por disco, de 1 controlador de discos com 100.000 horas de MTTF e de 1 fonte de alimentação com $400.000h$ de MTTF.

$$FailureRate = \left(\frac{2}{500.000}\right) + \left(\frac{1}{100.000}\right) + \left(\frac{1}{400.000}\right) \tag{3.2}$$

$$= 16.500\ FIT$$

$$MTTF = \frac{1.000.000.000}{16.500} \tag{3.3}$$

$$\approx 60606{,}06h$$

Para além da disponibilidade calculada do *host* local, um agente de reconhecimento tem conhecimento acerca do estado da coligação, dos componentes e dos níveis de QoS atribuídos. Os pontos seguintes especificam detalhadamente os dados que um AR é responsável por saber:

- Sabe quais os agentes de execução atribuídos a cada componente na máquina local;
- Sabe quais os componentes que se encontram a ser executados na máquina local juntamente com o nível de QoS atual;
- Sabe calcular e armazenar as subcoligações dos seus componentes – ver Definição 7;
- Conhece o identificador dos agentes de execução com que já negociou, que estão registados na sua *queried list*[6] e é responsável por manter esta lista;
- Tem a informação relativa ao nível de QoS com que os agentes de execução da máquina local inicialmente estão a executar – mapa de níveis de QoS de execução.

Para além de todos os aspetos mencionados, ao nível de gestão de QoS o agente de reconhecimento tem ainda as seguintes responsabilidades:

- Calcular o peso dos *upgrades/downgrades* de QoS e verificar se são viáveis na máquina (*feasibility*);
- Está encarregue de informar os agentes de execução do novo nível de QoS - melhoria ou degradação da qualidade - a que devem executar o seu componente;
- Coordenar as atualizações dos parâmetros de execução, isto é, os *upgrades* e os *downgrades*;
- Informar os outros agentes de reconhecimento, em *broadcast*, acerca de uma diminuição no nível de QoS da execução c_i de um agente Ae_q. Este poderá usufruir de um nível de qualidade superior, proposto por um outro agente Ar_x, em função da disponibilidade da máquina em que este se encontra.

Este último passo consiste na reactualização da *queried list* de cada AR. Portanto, sempre que baixa um nível de QoS de pelo menos um dos seus componentes, na obtenção de um novo SLA com QoS mais degradado, um AR envia a lista de AEs cujo nível de QoS foi degradado, em *broadcast* para todos os outros ARs da coligação. Desta forma, os ARs removem da *queried list* os agentes contidos nessa lista de reactualização, à exceção do último AE que tenha saído, caso a *availability* não se tenha alterado desde então. Esta ação é essencial pois todos os ARs trocam informação entre si de modo a manter a coligação convergida. Desta forma, um AE só volta a ser interrogado por um AR cuja *availability* sofreu efetivamente alterações, ou seja, se o AR consegue eventualmente oferecer um nível de QoS mais elevado do que o nível que outrora foi rejeitado pelo AE.

[6] Esta lista consiste num registo que contém todos os agentes de execução que já foram interrogados por um AR durante sua a atividade.

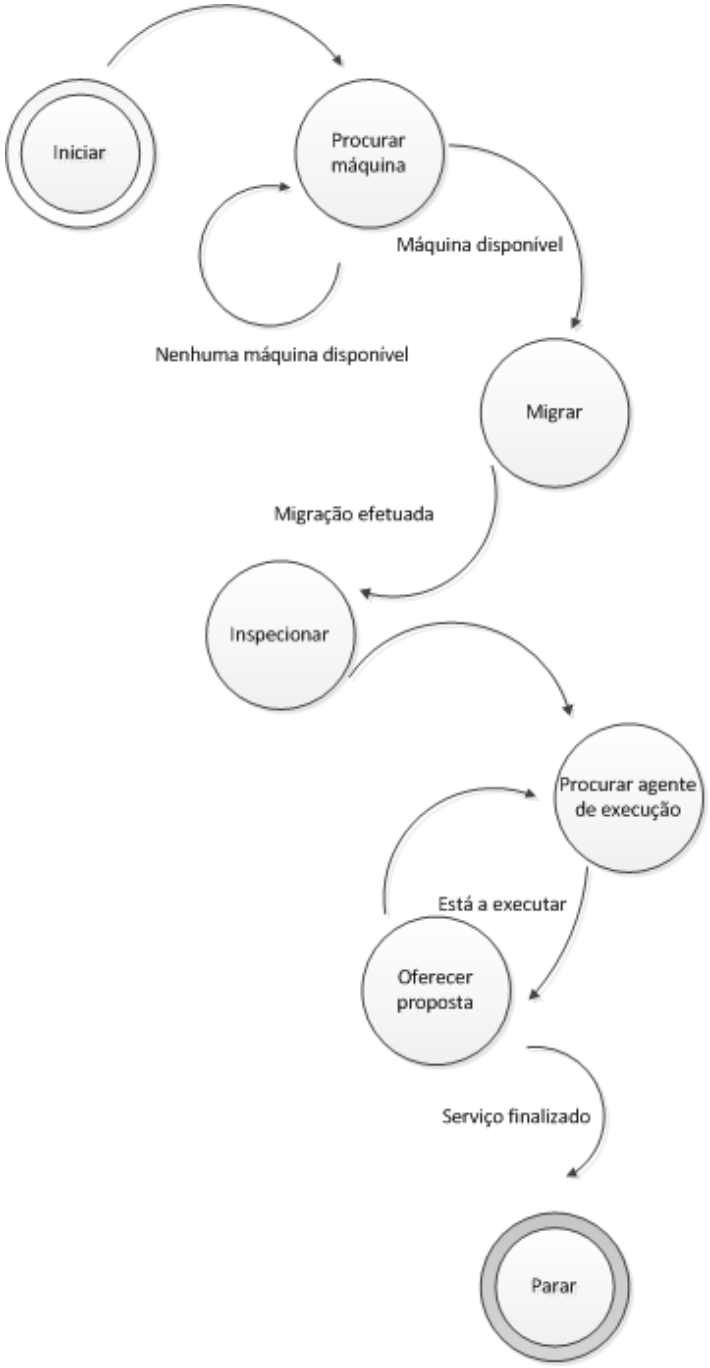


Figura 3.5 – Máquina de estados finitos de um AR

Definição 2 Agente de execução

Um agente de execução Ae_q consiste numa entidade do modelo que é responsável pela execução de um componente c_i de um serviço S numa determinada máquina aderente à coligação. Ae_q dialoga com o agente controlador, embutido na plataforma e responsável por iniciar os agentes, e fica encarregue de um componente ainda não atribuído a outro

agente de execução. Este processo decorre simultaneamente à inicialização e migração dos agentes de reconhecimento, dado que cada agente executa numa *thread* independente.

Sendo C_{A_e} o conjunto de agentes de execução inicializados pelo sistema, um agente de execução $Ae_q \in C_{A_e}$ é responsável pelo manuseamento e manutenção da seguinte informação:

- O componente que executa, bem como o nível de QoS a que este é executado;
- O seu papel de réplica (*active/shadow*);
- Os *n* parâmetros de *input* configurados no componente c_i, previamente à sua execução;
- Os componentes sucessores de c_i no grafo $\mathcal{G}_S$ do serviço.

Numa fase posterior, após o diálogo com outros agentes, deve saber:

- Os identificadores dos agentes responsáveis pelos componentes sucessores de c_i no grafo $\mathcal{G}_S$;
- O identificador do seu clone virtual, após a replicação virtual;
- O identificador do seu clone físico, após a replicação física;
- A proposta atual recebida - sendo esta composta pela *global reward* e pela *local reward* e a *reward* proveniente da maximização dos processamentos em nós paralelos (ver 3.4.2).

Este tipo de agente tem um conjunto de capacidades específicas: efetuar a inicialização de todos os dados necessários para a execução correta de c_i, criar um clone, *i.e.* uma réplica da sua instância idêntica à original, e de migrar para outro dispositivo membro da coligação juntamente com c_i, para além de dominar os protocolos de comunicação e de ser capaz de realizar conversações com outros agentes.

Em termos de iniciativa de diálogo com os seus colaboradores, um agente Ae_q deve:

- Publicar informação acerca do seu componente, como o nome, numa arquitetura *publish/subscribe*;
- Informar o agente de reconhecimento Ar_p que é responsável pela monitorização da *availability* da máquina onde Ae_q se encontra hospedado;
- Informar o seu clone virtual de quando se vai mover e para onde;
- Enviar *heartbeats* aos seus clones, virtual e físico, periodicamente, através do envio de mensagens simples, com o seu identificador, como sinal de que se encontra ativo;
- Informar os seus parceiros, virtual e físico, da ocorrência de uma falta de um *heartbeat*, ou seja, notificá-lo para proceder a uma auto-reiniciação;
- Em caso de não ser um nó final em $\mathcal{G}_S$, deve enviar o seu *output* gerado $O_{c_i} = \{(c_l, Q^l_{val})\}$ para os nós sucessores de c_i no grafo.

Todas as mensagens relativas ao processo de negociação entre agentes de reconhecimento e de execução estão descritas no tópico 3.4.1.

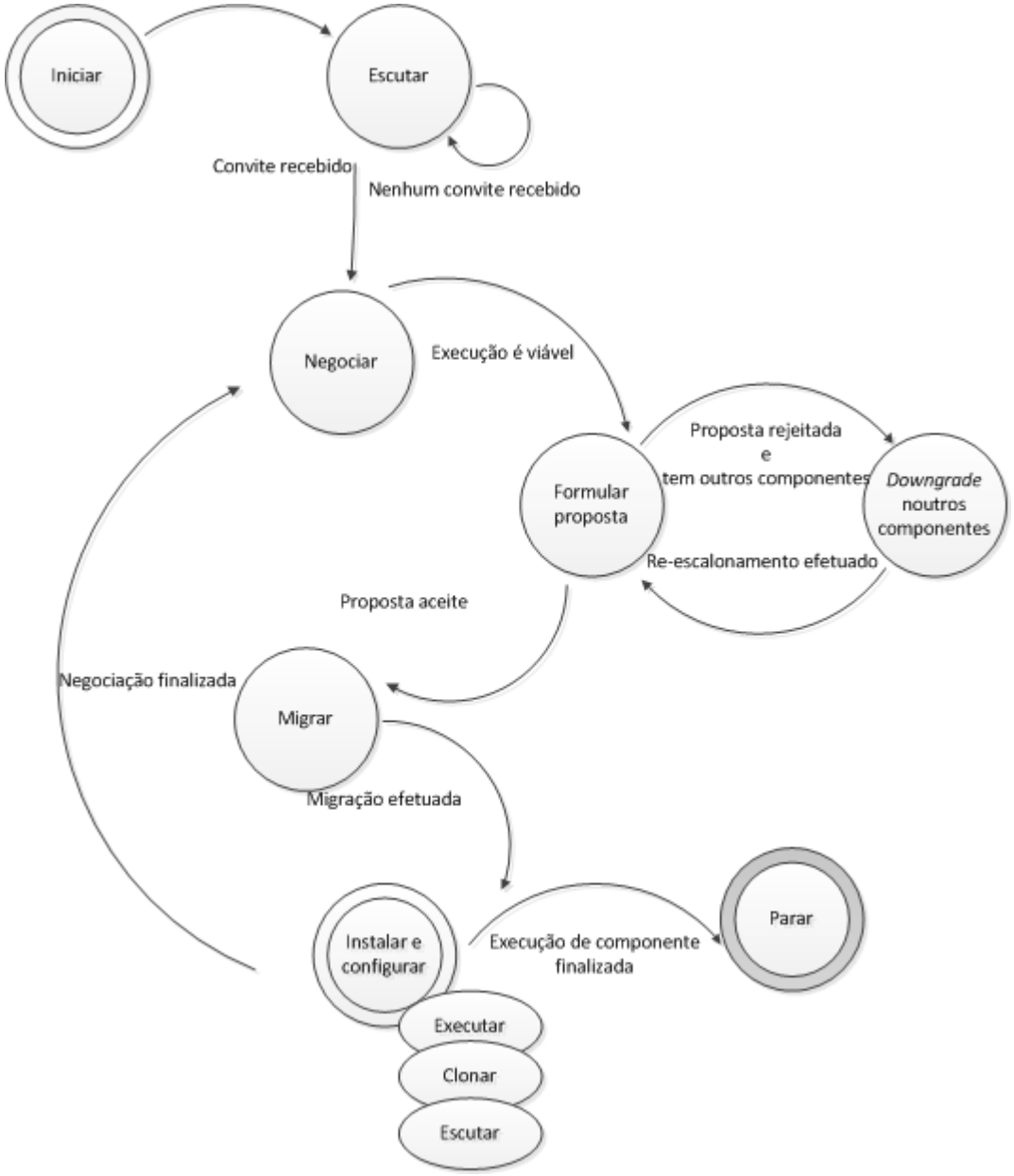


Figura 3.6 - Máquina de estados finitos de um AE

Na Figura 3.6, "Instalar e configurar" é um multiestado composto pelos subestados "Executar", "Clonar" e "Escutar", pois o AE está concorrentemente a executar o seu componente ao mesmo tempo que procede à sua clonagem. Simultaneamente, está à escuta de novas negociações com ARs e de eventuais modificações de nível de QoS.

Considera-se que a formação da coligação está totalmente completa quando já estão criadas as réplicas virtuais dos agentes e para cada componente já foram criadas réplicas físicas, primária e secundária. Contudo, isso não significa que a coligação tenha convergido, ou seja, que a sua qualidade produzida não possa ser melhorada a nível geral.

Quando a formação está completa, é atingido o número máximo de agentes ag_{max} dado pela equação (3.4):

$$ag_{max} = 4n_{c_i} \tag{3.4}$$

Na expressão acima, n_{c_i} é o número de componentes do serviço a executar. Evidentemente, o número de agentes é diretamente proporcional ao número de componentes, com uma razão igual a 4, e isso beneficia o modelo em termos de escalabilidade que é uma das principais desvantagens dos MAS (ver tópico 2.6.6).

3.4 Formação de coligações

A formação de coligações é um método importante na cooperação em ambientes multiagente. O processo de formação de coligações deve permitir a escolha de nós, baseada nos seus recursos e *availability*. Estes nós formam um grupo denominado coligação, cuja qualidade é melhorada ao longo da operação do modelo.

Para efeitos da formação da coligação considera-se o ponto inicial um nó n_r que pede ajuda na execução de S e despoleta a sequência de ações que levam à formação da coligação.

3.4.1 Coordenação auction-bidding

Os esquemas *auction-bidding* dão a possibilidade às máquinas intervenientes na formação da coligação de formularem propostas quantificáveis pelos componentes que constituem o serviço. Desta forma os agentes de reconhecimento conseguem expressar a disponibilidade e capacidade, traduzidas pela *availability*, do *host* onde se encontram hospedados. É este tipo de coordenação em formato de leilão que permite, numa primeira fase, a convergência da coligação e, numa posterior, o desenvolvimento da sua qualidade.

Após a migração de um agente de reconhecimento para uma nova máquina inicia-se o processo de negociação entre este e os agentes de execução, criados na máquina coadjuvada. Posteriormente, os agentes de reconhecimento hospedados nas máquinas da rede, cuja *availability* já foi analisada, concretizam propostas e enviam-nas para os agentes de execução que não se encontram na sua *queried list*. Se um agente de execução Ae_q^r, que se encontre hospedado num dispositivo r, não se encontra listado na *queried list* de um agente de reconhecimento Ar^s, hospedado em s, isto pode corresponder a uma de duas situações:

- Ou Ae_q^r nunca foi interrogado por Ar^s;
- Ou então o agente Ar^r efetuou um *downgrade* ou um *upgrade* ao nível de QoS de qualquer um dos componentes hospedados no seu *host* r, inclusive a Ae_q^r.

Na segunda hipótese a *availability* de r efetivamente alterou-se e Ae_q^r é novamente interrogado de novo por todos os outros agentes de reconhecimento, pois pode existir um outro agente Ar^s que possua uma melhor *availability* e possa assim oferecer um nível de QoS ${Q_{val}^i}'' > {Q_{val}^i}'$.

Em termos algébricos isto corresponde às seguintes expressões:

$$\exists\, Ae_x^r \subset CD^r : \forall\, Ae_x^r \in C_{Ae}^r$$

$$CD^r = \left\{\left\{Ae_1^r, c_i, {Q_{val}^i}'\right\}, \ldots, \left\{Ae_q^r, c_j, {Q_{val}^j}'\right\}\right\} : {Q_{val}^i}' < Q_{val}^i, {Q_{val}^j}' < Q_{val}^j \qquad (3.5)$$

Onde:

- CD^r é o conjunto de *downgrades* a realizar numa máquina r;
- C_{Ae}^r é o conjunto de agentes de execução com componentes na máquina r;
- ${Q_{val}^i}'$ e ${Q_{val}^j}'$ são os novos níveis de QoS;
- Q_{val}^i e Q_{val}^j são os níveis de QoS a atualizar.

Como é possível concluir, os agentes de reconhecimento atuam de uma forma "egoísta", visto que pretendem contratar todos os agentes de execução possíveis. Assim, os agentes de execução vão certamente tender a aumentar o nível de QoS configurado no componente que estão a executar. A chegada de novos dispositivos à coligação irá contribuir para uma maior distribuição de componentes pelas máquinas, achando-se assim uma configuração global mais equitativa na execução de um serviço S.

Inicialmente a negociação decorre entre dois agentes hospedados no mesmo nó, que é a máquina que efetua o pedido de auxílio de execução do serviço. Nesse caso o agente de execução não necessita de migrar, pois é inicializado nesse nó inicial. Caso seja um agente hospedado numa outra máquina a requisitar a migração de um agente de execução, quando este aceita a sua proposta migra para o nó do AR proponente. Os dois tipos de agentes, AR e AE, só podem processar uma negociação de cada vez, isto é, não há negociações concorrentes entre os mesmos agentes, para não sobrelevar a complexidade do modelo e para salvaguardar a sua consistência. O processamento de negociações em paralelo e/ou concorrentes requereria outro tipo de abordagem de coordenação no desenvolvimento do modelo.

A negociação que decorre entre o agente de reconhecimento e o agente de execução passa por duas fases fundamentais. A primeira consiste numa troca de informação entre AE e AR, para o último saber o qual o componente a executar e os respetivos níveis relevantes. Esta primeira fase resume-se nos seguintes passos ordenados:

1. O agente de reconhecimento escuta continuamente a entrada de novos agentes de execução na plataforma, até detetar um que ainda não esteja estabelecido;
2. Quando tal acontece, o AR oferece a máquina ao agente de execução;

3. O AE interpreta esta mensagem como um convite e responde com o nome do seu componente, a executar;
4. Como o AR tem toda a informação acerca da planta do serviço $\mathcal{G}_S$, verifica qual das seguintes decisões irá tomar, conforme as circunstâncias negociais:
    a. Se o componente for *feasible*, isto é, a execução do componente é viável, então AR aceita o componente;
    b. Se não for *feasible*, *i.e.* a máquina não tem recursos suficientes para executar o componente com o nível mínimo, então o AR tenta fazer *downgrade* aos componentes dos AEs que já se encontrem hospedados na máquina local:
        i. Se o *downgrade* efetuado a esses componentes tornar a execução *feasible*, ou seja, se o novo SLA de configuração de QoS degradado for aumentar suficientemente a *availability* da máquina, então AR aceita o componente;
        ii. Senão o AR rejeita o componente.

Como se pode verificar, se o componente não for *feasible*, o AR tenta executar *downgrades* aos componentes que já se encontram a executar na sua máquina, e negoceia com os AEs tendo em vista à obtenção de um novo SLA (ver Definição 1). Este processo de negociação do *downgrade* permite que o modelo reaja numa linha NGU e traz benefícios no esforço realizado para a execução de S.

Se, na primeira fase, o AR chegar a um acordo com o AE acerca do seu componente, procede-se a um segunda fase de negociação, na qual o AR formula uma proposta baseada na disponibilidade da sua máquina para o AE. Este processo é constituído por vários passos, na seguinte ordem:

1. O AR calcula a proposta (vetor de valores numéricos decimais) - *local reward, global reward* e *reward* proveniente da maximização dos processamentos em nós paralelos (PCPM *reward*) - e envia-a para o AR. Estes valores são calculados a partir da *availability* da sua máquina;
2. O AE aceita esta proposta se uma ou mais das seguintes condições forem obedecidas, por ordem:
    a. A *local reward* proposta do componente for maior do que a *local reward* atual;
    b. No caso da *local reward* proposta ser igual à atual, o AE vai aceitar o componente se a global *reward* for maior do que a *global reward* atual;
    c. Se a *local reward* e a *global reward* propostas forem iguais à *local reward* atual e à *global reward* atual, respetivamente, então o AE aceitará a proposta se:
        i. A PCPM *reward* proposta for maior do que a PCPM *reward* atual (ver Definição 6).

Se estas condições não forem verificadas o AR adiciona o identificador do AE com que negociou à *queried list*. De notar que, se o AE chegar a acordo com o AR proponente antes de migrar para o novo *host*, informa o AR da máquina local que se vai mover. Esta notifi-

cação, apesar de ser de cariz *event-triggered*, é mais benéfica do que uma notificação *time-triggered*, pois o evento de notificação da migração não é crucial para o funcionamento da coligação. Para além disso, este tipo de notificação evita que o AR de uma máquina esteja constantemente à escuta de AEs recém-chegados.

Entretanto os agentes de reconhecimento continuam a notificar tanto a máquina local como as restantes máquinas da coligação acerca da adesão de novas máquinas à execução cooperativa.

3.4.2 Algoritmos *anytime*

O funcionamento em formato de leilão (*auction-bidding*) dá a possibilidade aos agentes de reconhecimento de concorrerem pelos nós através de licitações. Contudo, só é possível formular propostas de serviço com valores aceitáveis e entendidos comumente se se puder expressar a qualidade de serviço pretendida através de uma descrição formal da configuração QoS. O tipo de descrição de QoS utilizado permite que esta seja integrada nos algoritmos *anytime*, que têm como objetivo a criação de licitações mensuráveis utilizadas nos diálogos entre os agentes.

Para a formação de coligações quase ótimas e determinísticas são utilizados dois algoritmos *anytime* propostos em [17], cujas propriedades se encontram provadas e demonstradas nesse documento. Foram efetuadas algumas modificações no funcionamento dos algoritmos e estes foram adaptados para um ambiente MAS. Antes de introduzir os dois conceitos algorítmicos e detalhar o seu funcionamento e os seus objetivos é importante referir que um algoritmo *anytime* consiste num conjunto de instruções que permitem o retorno de uma solução válida para o problema mesmo que a sua execução seja interrompida em qualquer momento antes da sua finalização [24].

Considera-se que uma proposta só é aceitável se conseguir satisfazer todas as dimensões QoS para os níveis de qualidade especificados pelo utilizador. Assume-se também que as restrições de QoS do utilizador são especificadas numa ordem de preferência descendente. Cada proposta admissível P_s é calculada, para cada dimensão, através da soma pesada de todas as diferenças entre os valores preferidos pelo utilizador ($L_{preferred}$) e os valores propostos em P_{st}, usando as seguintes equações [17]:

$$distance(P_s) = \sum_{k=1}^{n} \omega_k \times dif(Q_k) \tag{3.6}$$

$$\omega_k = \frac{n-k+1}{n} \tag{3.7}$$

Onde:

- n é o número de dimensões QoS;

- $0 \leq \omega_k \leq 1$ é a importância relativa da dimensão QoS k, Q_k.

Na equação (3.6, $distance(P_{st})$ determina a distância dos valores propostos relativamente aos valores preferidos pelo utilizador, que resulta num só valor decimal.

A importância relativa (ω), expressa através da equação (3.7, é calculada em função da prioridade que o utilizador definiu para os vários atributos e dimensões QoS que compõem a configuração de QoS. Quanto maior a importância relativa de um atributo ou de uma dimensão de QoS, maior é a prioridade que o utilizador lhe imputou.

A função das equações (3.8) e (3.9) consiste em calcular ao grau de aceitabilidade de um valor proposto para cada atributo, considerando os domínios contínuo e discreto.

$$dif(Q_k) = \sum_{i=1}^{attr_k} \omega_i \times |da(Prop_{st} \times Pref_{st})| \tag{3.8}$$

Onde:

- $attr_k$ é o número de atributos da dimensão k;
- $Prop_{st}$ é o valor proposto para um atributo i;
- $Pref_{st}$ é o valor preferido para um atributo i;
- $length(Q_k)$ é o número de atributos da dimensão Q_k.

$$da = \begin{cases} \dfrac{Prop_{st} - Pref_{st}}{\max(Q_k) - \min(Q_k)}, se\ Q_{ki}\ for\ de\ domínio\ contínuo \\ \dfrac{pos(Prop_{st}) - pos(Pref_{st})}{length(Q_k) - 1}, se\ Q_{ki}\ for\ de\ domínio\ discreto \end{cases} \tag{3.9}$$

O valor do grau de aceitabilidade traduz a conformidade dos valores de qualidade propostos relativamente aos valores de qualidade preferidos. Relembra-se que na especificação dos níveis QoS tanto as dimensões como os atributos devem ser indicados por ordem decrescente, ou seja, dos preferidos aos mínimos. Caso contrário os resultados reproduzidos pelas equações não corresponderão aos valores corretos.

As seguintes fórmulas, da (3.10) à (3.13), são utilizadas na determinação dos valores das *rewards*. Estas consistem em indicadores devidamente calculados segundo determinados critérios, com o objetivo específico de determinar valores heurísticos. Para este trabalho, as *rewards* são apresentadas sob a forma de números decimais e são calculadas tendo em conta critérios específicos do problema: em função da *availability* de uma determinada máquina e em função de um componente a negociar. Neste caso, o principal objetivo é possibilitar o cálculo de um valor, com um tipo de dados primitivo e computacionalmente simples (decimal), e com este licitar pelo componente. Para além disso, as *rewards* permitem quantificar o nível de qualidade de serviço que um AR irá fornecer de forma unifor-

me. Isto torna-se extremamente vantajoso em cenários onde os componentes têm parametrizações de QoS distintas.

Definição 3 ***Local reward***

A *local reward* consiste numa medida heurística, utilizada no Algoritmo 1, cuja fórmula baseia-se inteira e exclusivamente no SLA de qualidade de serviço acordado para um conjunto de componentes C^s que se encontra a executar numa máquina s. Quanto maior for a degradação de serviço em C^s, para que seja possível acomodar um novo componente c_x, pior é a *local reward* proposta ao agente de execução Ae_x responsável por c_x. Independentemente da ocorrência de degradação de serviço, a *local reward* é calculada baseando-se na qualidade de serviço fornecida ao conjunto C^s.

De notar que se não houver qualquer componente a executar numa máquina proponente s, *i.e.* $C^s = \{\emptyset\}$, então a *local reward* não será utilizada no Algoritmo 1. Isto deve-se ao facto de a máquina s não estar sobrecarregada com outros componentes; o que irá entrar em conta na formulação da proposta será o valor da *global reward* (ver Figura 3.9), que representa a qualidade efetiva que s oferece ao componente negociado.

A expressão apresentada na equação (3.10) representa a equação da *local reward*, oferecida por s a t, que corresponde ao inverso do somatório de todas as diferenças entre os níveis de serviço propostos e os níveis de serviço desejados, *i.e.* o inverso das distâncias, num conjunto de n componentes que se encontram a executar numa máquina proponente s.

$$LR_{st} = \frac{\left(\frac{1}{\sum_{i=1}^{n}\left(distance\left(P_s\left(c_i^s\right)\right)\right) + distance\left(P_{st}\left(c_x^t\right)\right)}\right)}{n+1} \tag{3.10}$$

Nesta equação:

- LR_{st} é a *local reward* oferecida pela máquina s a uma outra t;
- n é o número total de componentes a executar numa máquina s;
- $distance(P_s(c_i^s))$ é a distância entre o nível de QoS preferido para cada componente c_i^s, a executar numa máquina s, e a proposta de QoS realmente oferecida a esse componente;
- $distance(P_{st}(c_x^t))$ é a distância entre o nível de serviço preferido de um componente c_x^t, a executar numa máquina t, e a proposta de serviço que é efetivamente oferecida a esse componente por s;
- c_i^s é um componente i que se encontra a executar numa máquina s;
- c_x^t é o componente, cuja execução decorre numa máquina t, pelo qual Ar^s negoceia.

À semelhança da equação (3.10), a *local reward* oferecida por t ao seu conjunto de componentes C^t é dada pela equação (3.11).

$$LR_t = \frac{\left(\frac{1}{\sum_{i=1}^{n}\left(distance\left(P_t\left(c_i^t\right)\right)\right)}\right)}{n} \tag{3.11}$$

Da mesma forma que no cálculo da *global reward*, nas equações (3.10) e (3.11), quanto menor for o valor de $distance\left(P_y(\dots)\right)$, maior é a *local reward*. Relativamente a n, quanto maior o seu valor, menor será a *local reward*, visto que maior é o número de componentes e, possivelmente, a sobrecarga na máquina s. Na equação (3.10) soma-se mais uma unidade a n, assumindo um cenário em que s sai vencedor da licitação por c_x^t. Assim, a *local reward* oferecida é o valor efetivo proposto a c_x^t no caso de este vir a executar em s.

Nesta fórmula não se assume nada relativamente à carga que cada componente exige das máquinas. No entanto faz-se assunção razoável de que quanto maior for o nível de QoS configurado num componente, maior é o consumo de recursos na máquina onde este é executado. Torna-se assim o funcionamento do modelo mais neutro, e não orientado a cenários com maior especificidade. As hipóteses de haver cenários onde há um balanceamento de carga pobre são eliminadas, pois Ar^s para licitar por c_x^t efetua o menor decréscimo possível no QoS dos seus componentes. Assim, a *local reward* mais elevada será a proposta efetuada pelo AR que efetua o *downgrade* mais reduzido, inclusive nenhum. Dado que os algoritmos *anytime*, Algoritmo 1 e Algoritmo 2, primam por assegurar a execução de todos os componentes e, pelo menos, os níveis de qualidade de serviço mínimos, fornecidos a cada componente, a *local reward* é utlizada em primeira instância. Só depois há um esforço no sentido de aperfeiçoar gradualmente os níveis de QoS em todos os componentes, recorrendo-se à *global reward*.

Definição 4 *Global reward*

A *global reward* é uma medida heurística, utilizada no Algoritmo 1, que representa numericamente a qualidade de serviço oferecida por um determinado agente segundo uma descrição de uma configuração QoS. Quanto mais próximos dos valores preferidos forem os valores propostos, para cada atributo, maior será a *global reward*. A equação (3.12) mostra o cálculo da *global reward*, incluída numa proposta P_{st} formulada pela máquina s para t, no sentido de obter a execução de um determinado componente. A fórmula que representa esta *reward* é a seguinte:

$$GR_{st} = \frac{1}{distance(P_{st})} \tag{3.12}$$

Onde:

- GR_{st} é a *global reward* oferecida pela máquina s a uma outra t, em função da proposta P_{st}.

Ainda relativamente à equação (3.12), $distance(P_{st})$ é inversamente proporcional ao valor da *global reward*. Portanto, quanto menor for o valor de $distance(P_{st})$, maior será o valor da *reward*.

Se o componente é servido em todas as dimensões ao nível máximo, então o valor de $distance(P_{st}) = 0$, logo $GR_{st} = \infty$.

Definição 5 Componentes paralelos

Considerando um conjunto de componentes $c_i \in \mathcal{G}_S$, c_i diz-se paralelo de um outro c_j , se e só se o *output* de c_i não é utilizado direta ou indiretamente como input c_j, e vice-versa, *i.e.*, algebricamente, se $(c_i, Q^i_{val}) \notin O_{c_j} \wedge (c_j, Q^j_{val}) \notin O_{c_i}$. A relação de paralelismo entre c_i e c_j representa-se pela expressão $c_i \,//\, c_j$.

Na Figura 3.7 é apresentado um exemplo de um serviço formado por 6 componentes, de A a F. Neste exemplo, o conjunto de componentes paralelos é formado por $B//C \wedge B//E \wedge D//C \wedge D//E$.

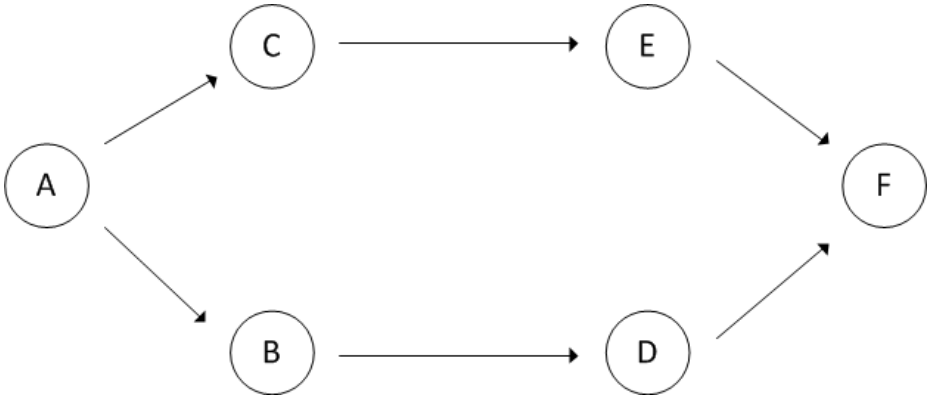


Figura 3.7 - Exemplo de serviço com componentes paralelos

Definição 6 PCPM *reward*

O terceiro indicador relevante para a atribuição mais benéfica de uma máquina s a um componente c_i designa-se por *Parallel Components Processing Maximization reward*. Esta *reward* é uma medida heurística utilizada para, tal como o seu nome indica, maximizar o processamento de componentes paralelos em máquinas diferentes.

A PCPM *reward* é calculada através da equação (3.13), onde uma máquina s que executa um componente c_i^s, com um grau de significância σ_i^s, se propõe para executar um componente c_x^t, localizado na máquina t.

$$PCPMR_{st} = \frac{1}{\sum_{i=1}^{n}(\sigma_i^s + 1)}\ ,\forall\ c_i^s // c_x^t, n > 0 \tag{3.13}$$

Nesta equação, os vários componentes c_i são paralelos a c_x, *i.e.* $\forall\ c_i^s // c_x^t$, e existe pelo menos um c_i em s paralelo a c_x, com $n > 0$. Se não existir pelo menos um componente c_i paralelo a c_x ($n = 0$), então $PCPMR_{st} = \infty$. O número de iterações, n, do somatório corresponde ao número de componentes paralelos a c_x. Quanto menor for o somatório do grau de significância dos componentes paralelos a c_x que se encontram a executar num *host* s, maior será o valor da $PCPMR_{st}$. Esta *reward* privelegia a execução de componentes com maior grau de significância, ou seja, os componentes mais importantes para o sistema num todo.

O cálculo da PCPM *reward* resulta da obtenção dos componentes paralelos num grafo $S_{\mathcal{G}}$ (Definição 5). Esta vai integrar o Algoritmo 1 que visa também potencializar as execuções dos componentes que são paralelos na planta do sistema em nós diferentes. É um processo que se torna muito menos complexo se for aplicado a várias partes do grafo, em vez de à sua totalidade, ou seja, aplicado às subcoligações (ver Definição 7).

Este indicador é apenas utilizado se ambas as *rewards local* e *global* atuais, que são as maiores que uma máquina recebeu desde o início da formação da coligação, forem iguais às *local* e *global rewards* da proposta. Por outras palavras, esta *reward* só é utilizada quando as *local* e *global rewards* oferecidas por um AR são iguais às, até ao momento, melhores *local* e *global rewards* que foram recebidas por um AE.

Em suma, no caso de a PCPM *reward* ser avaliada, se um componente c_i reunir as condições para ser processado em paralelo com um outro c_j, ambos serão colocados pelos seus agentes em máquinas distintas, pelo facto de serem componentes funcionalmente independentes entre si.

Recorrendo ao grafo ilustrado na Figura 3.7, é possível demostrar as vantagens da utilização desta *reward*. A Figura 3.8 apresenta um exemplo de uma coligação formada por 4 *hosts*: x, s, t, y. Os componentes paralelos entre si são colocados em máquinas distintas. Por exemplo, através da Figura 3.7, como $B\ //\ C$ ambos são colocados em *hosts* separados, pois o *output* produzido por um não é enviado para o outro, isto é, B não é funcionalmente dependente de C, e vice-versa. Já C não é paralelo a E, *i.e.* $\neg\ (C // E)$, logo podem ser colocados na mesma máquina. É importante frisar que a PCPM *reward* serve como "critério de desempate" entre duas *global rewards* equivalentes. Este indicador irá determinar se é preferível migrar o AE para a máquina do AR proponente, ou se, pelo contrário, é preferível o AE permanecer na máquina onde se encontra.

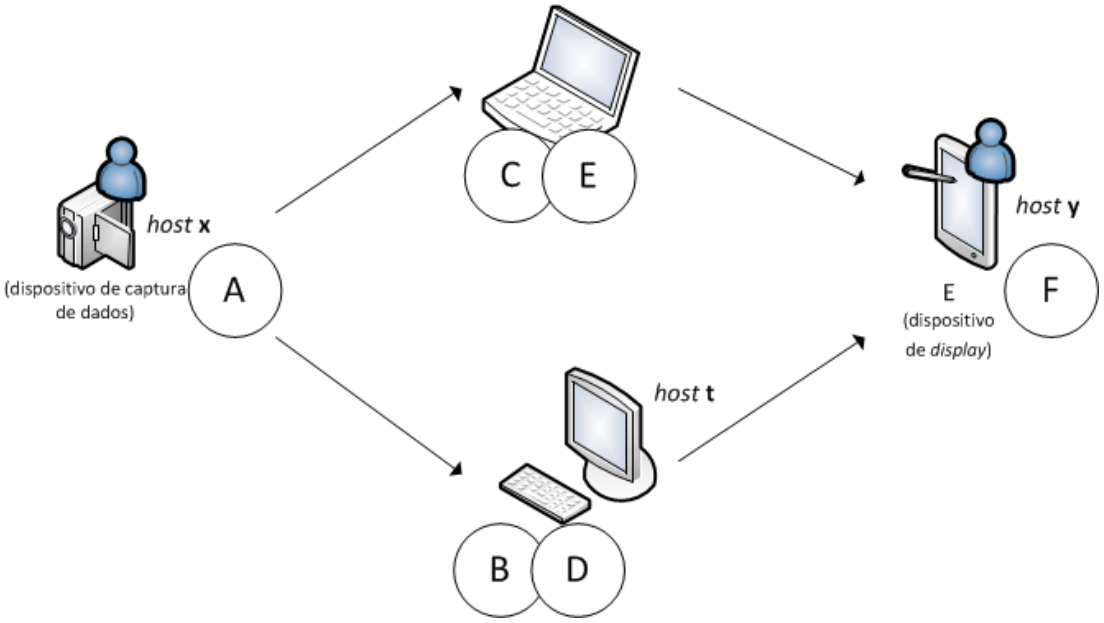


Figura 3.8 - Exemplo de coligação com componentes paralelos

No exemplo apresentado na Figura 3.8 é possível apurar que, ainda que o AR do *host s* formule uma proposta ao AE com o componente B, no *host t*, oferecendo *global* e *local rewards* idênticas às atuais deste AE, a PCPM *reward* não é satisfatória. A sua incapacidade de conseguir B advém do facto de o *host s* já se encontrar a executar os componentes C e E. Na realidade, o valor da PCPM *reward* oferecida será $PCPMR_{st} = \frac{1}{3}$, pois $\sigma_C^S = 2$ e $\sigma_E^S = 1$, enquanto a *reward* previamente oferecida pelo *host t* seria ∞, visto que esta máquina não se encontra a executar qualquer componente paralelo a B. Para além de maximizar as execuções dos componentes paralelos de um serviço, esta reward, quando utilizada, tende, naturalmente, a colocar componentes funcionalmente dependentes entre si nas mesmas máquinas, *e.g.* $\neg\,(C\ //\ E)$. Assim, o *output* de C, que é enviado para E, não terá de ser transportado pela rede ao ir para outra máquina e é passado internamente, isto é, no mesmo dispositivo, para o componente destinatário, que recebe os dados como *input*. Desta forma, é minimizado o *overhead*, logo o impacto dos dados produzidos pela coligação no desempenho da rede é menor. Adicionalmente, o escalonamento para a execução de E encontra-se em espera visto que este não executa antes de C lhe enviar o seu *output*. Assim sendo, a execução de C é priorizada pois não é efetuada concorrentemente com E.

Definição 7 Subcoligação

Dado um DAG $\mathcal{G}_S$ de um serviço S, designa-se por subcoligação $\mathcal{G}_k$ de $\mathcal{G}_S$ uma coligação de menor dimensão, cujos n vértices são representados por $V_n^{\mathcal{G}_k} \subset \mathcal{G}_S$. O primeiro vértice $v_1^{\mathcal{G}_k}$ deste conjunto é: ou o nó inicial de $\mathcal{G}_S$, *i.e.* $v_1^{\mathcal{G}_k} = v_1^{\mathcal{G}_S}$, ou um *cut-vertex* de $\mathcal{G}_S$. O último vértice $v_n^{\mathcal{G}_k}$ é: ou o *cut-vertex* de $\mathcal{G}_S$ imediatamente a seguir a $v_1^{\mathcal{G}_k}$, ou o nó final de

$\mathcal{G}_S$, *i.e.* $v_n^{\mathcal{G}_k} = v_n^{\mathcal{G}_S}$. Por outras palavras, as subcoligações definem subconjuntos da coligação que são encarregues de executar parte da planta do grafo, e são delimitados por um dos seguintes pares de vértices:

- Vértice inicial e o vértice final;
- Ou vértice inicial e um *cut-vertex*;
- Ou dois *cut-vertexes;*
- Ou entre um *cut-vertex* e o vértice final.

Por conseguinte, quando efetua os cálculos relativos ao grafo do serviço, um agente de reconhecimento não necessita tratar toda a informação respeitante ao grafo. Para os casos convenientes, considera-se suficiente que divida a toda a informação relativa às subcoligações onde os seus componentes estão contidos. As subcoligações são apropriadas para casos como o cálculo da PCPM *reward*.

É importante referir que um *cut-vertex* no grafo pertence a ambas as subcoligações, logo possui informações acerca das duas, como o número total de componentes da coligação, quais os componentes e as suas ligações, entre outras. Assim sendo, quando um AE está a executar um componente que é um *cut-vertex* no grafo, o AR responsável pela máquina que hospeda esse AE deve armazenar a informação relativa aos grafos das duas subcoligações que o *cut-vertex* une. Caso ainda não possua essa informação pode pedir ao agente de reconhecimento da máquina inicial para lhe enviar a informação em falta relativa à subcoligação.

Algoritmo 1 *Anytime global* QoS *optimisation*

Este algoritmo tem em vista fundamentalmente a otimização do nível de QoS do componente a negociar. A partir deste algoritmo resulta a formação iterativa de uma coligação, dado que é expectável que a qualidade produzida melhore á medida que o algoritmo vai processando a informação relativa à coligação. Este algoritmo utiliza as três *rewards* para selecionar o melhor nó N_s para a execução de um componente $c_i \in S$, enquanto o tempo de execução t não ultrapassa a *deadline* definida.

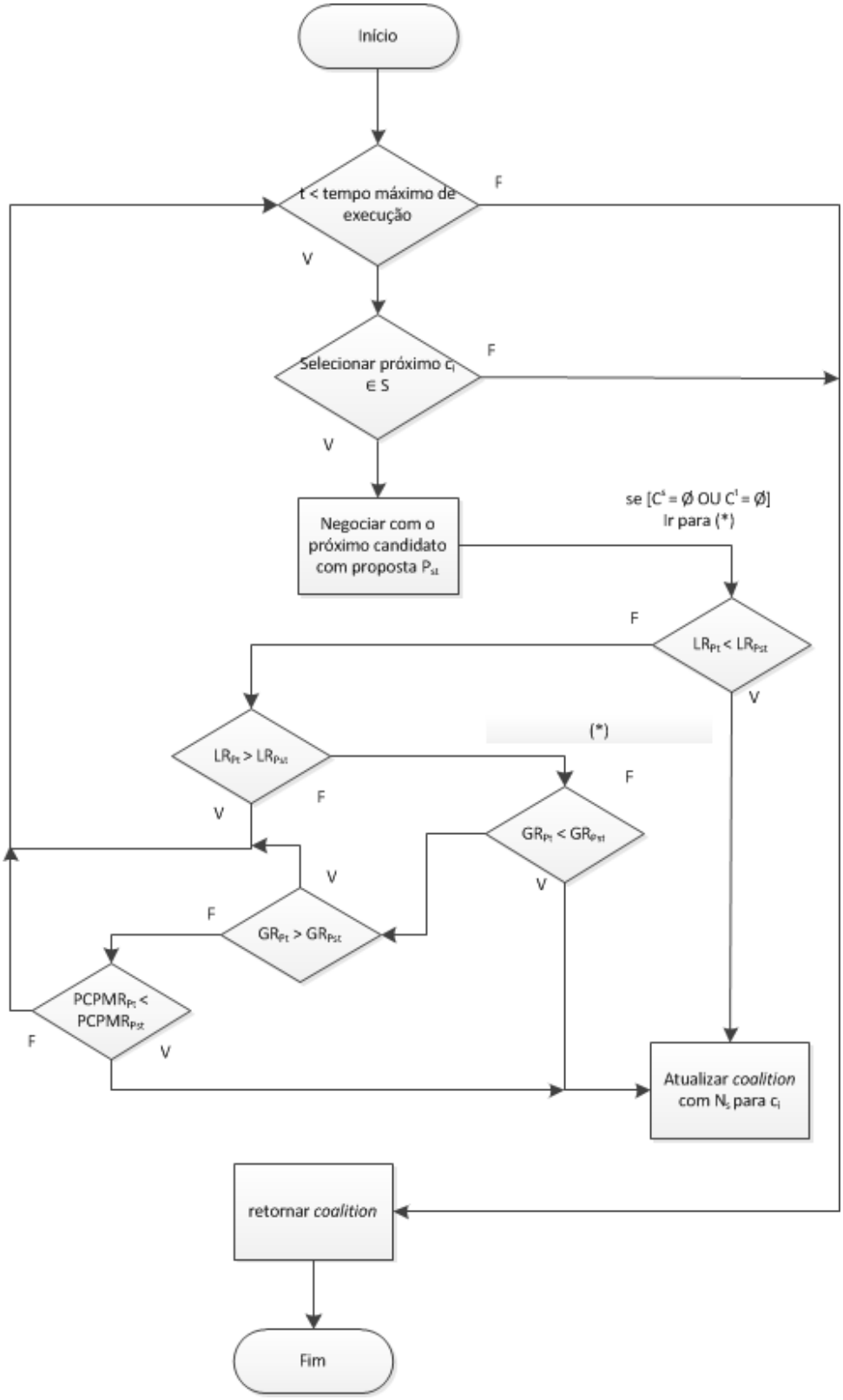


Figura 3.9 - Algoritmo *anytime global* QoS *optimisation*

A Figura 3.9, acima apresentada, descreve o algoritmo *global* QoS *optimisation*. Cada agente de reconhecimento Ar^s instalado num nó N_s que esteja apto para hospedar um

componente do serviço, formula uma proposta de serviço de acordo com o algoritmo *local* QoS *optimisation* (ver Algoritmo 2). De seguida, envia a sua licitação para a um nó N_t com a sua proposta de fornecimento de serviço P_{st} e a sua *local reward* LR_{st}.

Neste algoritmo dá-se prioridade à seleção de nós candidatos que forneçam um nível de QoS que minimize a degradação de qualidade do serviço dos componentes previamente já aceites e hospedados em cada membro. Eventualmente, pode nem haver degradação do seu nível. O algoritmo pode ser interrompido em qualquer instante, ainda assim devolvendo uma solução, que no pior dos casos é vazia, *i.e.* $coalition = \{\emptyset\}$. As propostas formuladas pelos AR são imediatamente avaliadas pelos AE e no caso de serem melhores, segundo os critérios especificados na segunda fase da coordenação *auction-bidding* (ver 3.4.1), são aceites. Posteriormente, dá-se o processo de migração do AE para o *host* do AR proponente.

Algoritmo 2 *Anytime local* QoS *optimisation*

Este algoritmo fundamenta-se no algoritmo proposto em [17] e tem duas vertentes na sua atividade. Uma delas consiste em realizar *upgrades* de QoS num determinado componente, mantendo os níveis de QoS previamente assegurados nos componentes coabitantes na mesma máquina. Por outro lado, a outra vertente tem como objetivo calcular e devolver um novo SLA de qualidade de serviço na tentativa acomodar um novo componente c_j. No cálculo do novo SLA o algoritmo deve encontrar uma configuração onde haja uma degradação mínima de serviço, ou seja, deve realizar o *downgrade* menos pejorativo para a qualidade do serviço.

A vertente de *upgrade* de serviço do algoritmo *anytime local* QoS *optimisation* passa por melhorar o nível de QoS de um conjunto de componentes C^s que estão a ser executados numa máquina s: um de cada vez, mantendo o nível de serviço previamente garantido em todos os outros. Para isso começa-se por tentar melhorar os níveis QoS nos componentes, partindo do atributo mais importante da dimensão QoS mais importante, definidos pelo utilizador. Com a parte do aperfeiçoamento de QoS no algoritmo pretende-se priorizar o melhoramento dos parâmetros mais relevantes, numa ótica de usufruição de qualidade do serviço por parte do utilizador. Só na fase final de um *upgrade* é que os atributos menos preferidos são melhorados, caso a *availability* da respetiva máquina seja suficiente.

A Figura 3.10 apresenta a vertente de melhoramento de serviço do algoritmo *local* QoS *optimisation*.

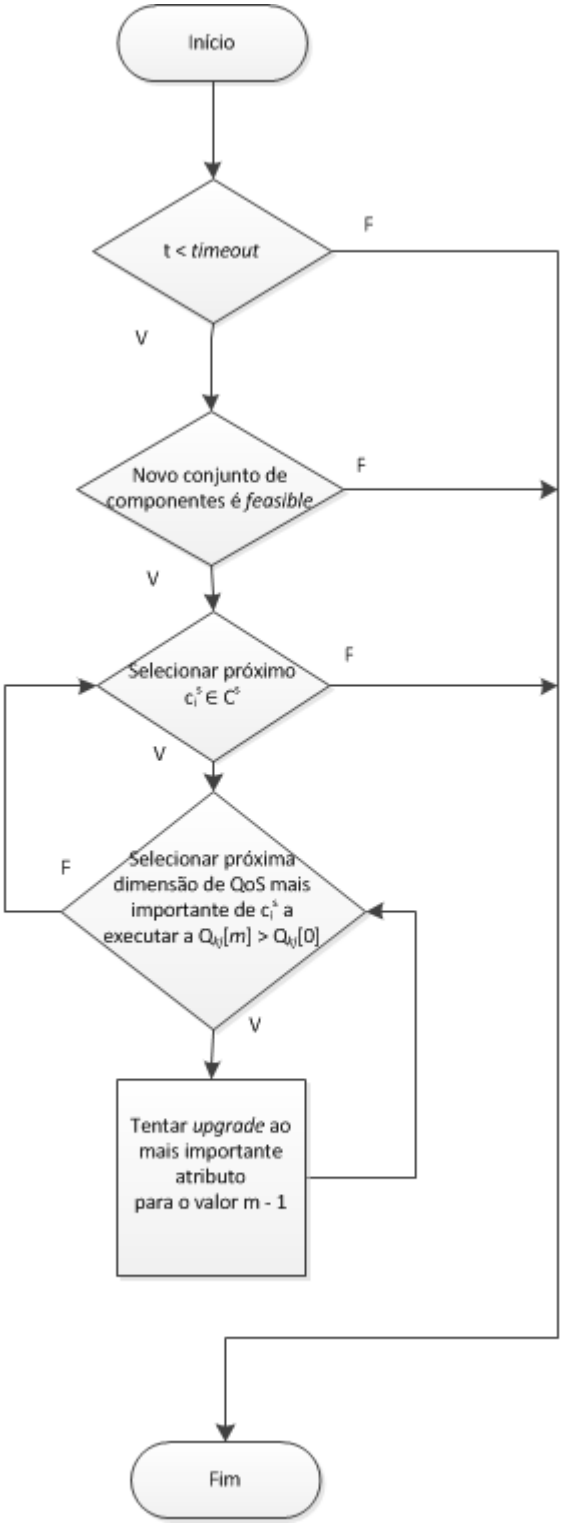


Figura 3.10 - *Upgrade* no algoritmo *anytime local* QoS *optimisation*

No processo de *downgrade* da qualidade de serviço o algoritmo tenta encontrar uma degradação mínima para acomodar um novo componente c_a. Começa por baixar a qualidade nos atributos menos significativos para o utilizador das dimensões QoS menos importantes.

A Figura 3.11 apresenta a parte do algoritmo relativa à degradação de serviço.

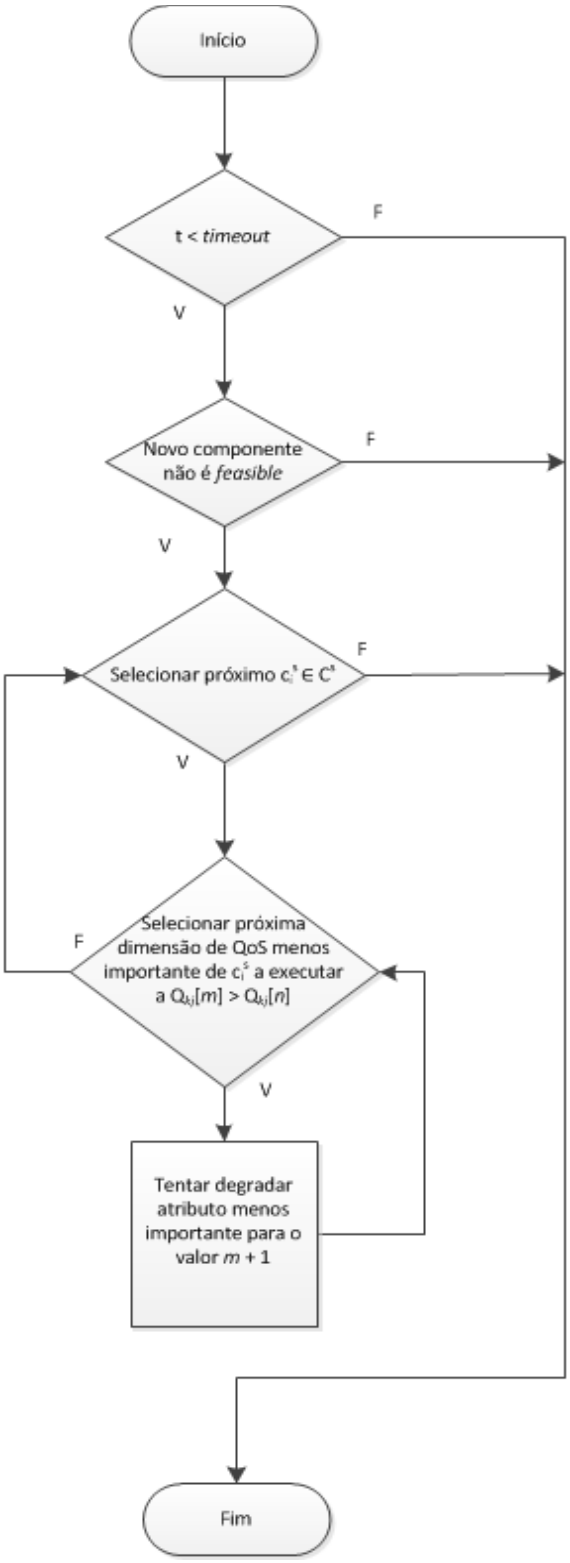


Figura 3.11 - *Downgrade* no algoritmo *anytime local* QoS *optimisation*

A utilização do algoritmo *local* QoS *optimisation* possibilita o melhoramento da qualidade de serviço dos componentes, pois o *upgrade* nos seus níveis QoS é tentado incessantemente. Tentam-se fazer *upgrades* aos atributos mais significativos (atr_1, …) das dimensões mais significativas (dim_1, …) e, de seguida, tenta-se melhorar o próximo atributo mais significativo (atr_2, …). Quando não existem mais atributos nessa dimensão efetua-se o mesmo às seguintes dimensões mais significativas (dim_2, …). A degradação, pelo contrário, só é realizada em última instância, para acolher um novo componente. Esta só se realiza se, dada uma determinada *availability*, o componente a ser negociado não for *feasible*. Para além disso, a negociação só tem sucesso se o *downgrade* efetuado tiver menor impacto nos níveis de QoS fornecidos pela máquina proponente ao seu conjunto de componentes do que nos níveis de QoS fornecidos pela máquina anterior ao seu conjunto local de componentes. Isto é, no cômputo geral, a quantidade de degradação proposta pela máquina proponente, para o seu conjunto de componentes, tem de ser menor do que a quantidade de degradação efetivamente existente na máquina atual do componente licitado. A degradação inicia-se no atributo menos significativo (atr_n) da dimensão menos significativa dim_m e, de seguida, parte para o próximo atributo menos significativo (atr_{n-1}). Quando não existem mais atributos nessa dimensão efetua-se o mesmo às seguintes menos significativas (dim_{m-1}, …).

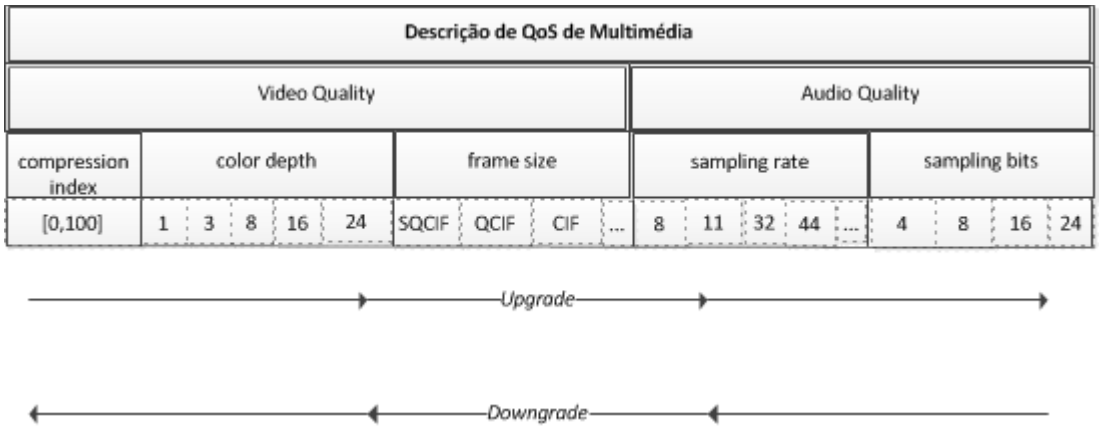


Figura 3.12 – (a) *upgrade* de QoS; (b) *downgrade* de QoS

Para a esquematização apresentada na Figura 3.12 recorreu-se ao exemplo apresentado na Tabela 3.2 para elaborar um exemplo prático de *upgrade*/*downgrade* de QoS. Sempre que é efetuado um *upgrade* (a), este é feito a partir da dimensão mais significativa para menos significativa. O mesmo sucede com os respetivos atributos e valores de cada dimensão. Isto significa que inicialmente, no processo de *upgrade*, o atributo «*compression index*», tendo um valor de 30, é alterado para 31 (domínio contínuo). O *upgrade* continua até o nível de QoS não ser praticável (*feasibility*) nesse componente, devido à *availability* da máquina onde se encontra hospedado. Quando «*compression index*» atinge o seu valor máximo (100), o processo de *upgrade* prossegue da mesma forma no atributo seguinte, «*color depth*». Assim que todos os atributos de uma dimensão estiverem parametrizados com o nível máximo, o preferido pelo utilizador, então passa-se para a seguinte dimensão

mais significativa, que neste caso é «*Audio Quality*». Relativamente ao processo de *downgrade* (b), este decorreria de forma totalmente inversa ao *upgrade*.

Definição 8 Qualidade da coligação

A qualidade geral efetivamente produzida por uma coligação é uma variável importante na mensuração da capacidade e performance da coligação gerada pelo modelo. Para efetuar o seu cálculo recorre-se aos valores das *local rewards* oferecidas pelo conjunto de máquinas aderentes à coligação. A equação (3.14) mostra a fórmula para o cálculo da qualidade geral de uma coligação.

$$Q_{coalition} = \sum_{s=1}^{m} \frac{\left(\frac{1}{\sum_{i=1}^{n} \left(distance\left(P_s\left(c_i^s\right) \right) + 1 \right)} \right)}{n}, k \in coalition, n > 0 \quad (3.14)$$

Nesta equação:

- $Q_{coalition}$ é a qualidade geral produzida pela coligação $coalition$;
- m é o número total de máquinas da coligação;
- n é o número de componentes hospedados numa máquina s, pertencente à coligação;
- $P_s(c_i^s)$ é a proposta real da máquina s oferecida ao componente c_i^s, hospedado em s.

Dado que todas as *local rewards* das m máquinas correspondem a valores baseados unicamente na qualidade de serviço fornecida aos componentes das máquinas, constata-se que estas medidas heurísticas são variáveis precisas no cálculo da qualidade produzida por uma coligação. No entanto, no caso de o nível de QoS proposto ser igual ao nível desejado, *i.e.* $L_{proposed} = L_{desired}$, a parte da fórmula $\frac{1}{distance\left(P_s\left(c_i^s\right)\right)}$ torna-se não quantificável, pois o seu valor é ∞. Por isso é somada uma unidade ao denominador desta parte da fórmula, em muito semelhante à da *local reward*. Este quociente resulta num valor confiável, baseado nos níveis de QoS proporcionados aos demais componentes da coligação. A qualidade da coligação, tal como a *local reward*, devolve um valor que leva em consideração o balanceamento de carga entre as máquinas. Quanto menor o número de componentes hospedados numa determinada máquina, maior a qualidade resultante nessa máquina. O valor de n tem que ser maior do que zero de forma a manter o valor devolvido mensurável para a avaliação da coligação.

A qualidade produzida pela coligação é calculada baseando-se principalmente no balanceamento de carga do serviço e na degradação mínima de QoS nos componentes, de forma análoga ao cálculo da *local reward*. Só depois é que entra em consideração, em termos de cálculo, a qualidade produzida individualmente por cada uma dos componentes.

3.5 Conclusões

Em síntese, o modelo proposto é aplicável em sistemas embebidos, nomeadamente do tipo distribuídos, *soft real-time* onde a máquina executante é pobre em termos de recursos físicos. O sistema de formação de coligações é de carácter distribuído, heterogéneo e aberto. Para ir ao encontro destas características optou-se por integrar no modelo um sistema multiagente. Numa perspetiva individual, ambos os tipos de agentes, AE e AR, podem ser classificados como *multi-task*, isto é, cada agente executa várias tarefas. No entanto, no que toca á contribuição para a execução do serviço, um AE é responsável por apenas uma *task*/componente. Os agentes tratam de aspetos como a decisão algorítmica e são proactivos na procura de novas máquinas. Trabalham em conjunto, com objetivos puramente individuais - o melhoramento do nível de QoS concedido a cada componente -, porém contribuindo para um objetivo comum – o desenvolvimento iterativo da configuração global da coligação, que por sua vez resulta na melhoria da qualidade geral do serviço.

A aplicação de agentes móveis no modelo, por sua vez, traz vantagens em termos de interindependência nodal. As vantagens da utilização deste paradigma são as seguintes:

- Os agentes podem clonar-se e mover-se fácil e rapidamente na rede, o que confere flexibilidade e dinamismo ao modelo;
- Os agentes dotados de algoritmos inteligentes e de comportamentos reativos, com estímulos internos, podem atuar de forma autónoma, dotando o modelo de uma maior descentralização e inteligência;
- O serviço é completamente transparente para o modelo e este não tem de saber detalhes acerca do seu funcionamento;
- Qualquer nó que esteja apto a hospedar um agente e a fornecer um serviço de qualidade mínimo ($\geq L_{minimum}$) definido pelo utilizador, é considerado apto para integrar a coligação;
- Não há necessidade de instalar um protocolo de diálogo entre os nós aderentes à coligação, pois os agentes, que se movem na rede, levam consigo o próprio meio de diálogo, por intermédio da plataforma de agentes[7];
- Nunca ocorrem *nested conversations*, que provocam o "efeito dominó", pois cada conversa entre agentes é única e terminável.

Relativamente à utilização de conhecimento heurístico, a eficácia dos algoritmos *anytime* utilizados juntamente com as três *rewards*, promove primeiramente a qualidade geral da coligação, através da utilização da *local reward* no método de licitação; tenta-se minimizar ao máximo a degradação do serviço, escolhendo as máquinas que maximizam a qualidade do serviço no acolhimento de novos componentes. Em segundo plano é utilizada a *global reward*, que indica o melhoramento de serviço de um só componente. Esta *reward* é utilizada para melhorar individualmente os níveis de QoS de cada componente. Em último lugar recorre-se à utilização da PCPM *reward* para não só potencializar as execuções de

[7] Atualmente, a grande maioria das plataformas de desenvolvimento de agentes móveis disponíveis proporcionam transparência relativamente à comunicação, como o protocolo e linguagem utilizados.

componentes funcionalmente independentes em máquinas distintas, como também para minimizar a introdução de dados na rede ao colocar componentes dependentes entre si nas mesmas máquinas. Como se tratam de algoritmos *anytime*, estes promovem também a resiliência do modelo, em virtude de devolverem sempre uma solução válida, tanto para a coligação (*global optimisation*) como para os SLAs de QoS estabelecidos nas máquinas da coligação (*local optimisation*), mesmo nos casos em que o algoritmo é interrompido.

O modelo visa potencializar as capacidades de todas as máquinas para que estas possam contribuir ao máximo no melhoramento do desempenho da coligação. A capacidade de cada máquina dita o impacto que esta terá na coligação. No modelo, a formulação de propostas recorre ao uso de *rewards* cujo cálculo se fundamenta na *availability* do *host*. Logo, as *rewards* representam com exatidão a capacidade das máquinas.

O facto de o modelo assentar num esquema baseado em EDRB que utiliza replicação ativa faz com que o tempo de recuperação dos nós, após a ocorrência de uma falha, seja mais curto. O processamento nos vários nós da coligação permite que se alcance um melhor balanceamento de carga. O modelo utiliza técnicas de tolerância a falhas como a replicação, de nós, e a migração, de agentes móveis. Procedeu-se à implementação de redundância virtual ativa, através da criação de réplicas virtuais, que podem assumir um papel primário ou secundário na execução do componentes que lhes é atribuído. As réplicas secundárias acompanharão e substituirão, caso necessário, as primárias. Para tolerar falhas de foro físico aplica-se redundância física ativa, replicando agentes e migrando-os para outras máquinas da rede que se encontrem disponíveis. Estas assumem um papel secundário e tornam-se substitutas das máquinas primárias, já estabelecidas. No modelo considera-se implícita a utilização da técnica de *checkpointing*, pois as réplicas tornam as falhas invisíveis para o utilizador, e os agentes de execução mantêm um *snapshot* do estado de processamento, visto que armazenam toda a informação necessária para a execução do seu componente. Para além disso, mesmo que os agentes sejam interrompidos conseguem prosseguir a sua atividade, ao retomar o estado anteriormente gravado.

Este modelo funciona numa arquitetura híbrida: resulta de uma fusão da ETA (*Event-Triggered Architecture*) com a TTA (*Time-Triggered Architecture*) devido ao facto de as ações serem despoletadas tanto com base em eventos espontâneos como periodicamente, isto é, em função do tempo. No entanto, para fazer face aos eventos que bloqueiam a execução de um componente, *e.g.* a falha de um nó, recorre-se a técnicas que se enquadram na TTA, como a troca de *heartbeats*.

Neste modelo não existe aquilo a que se pode chamar de "diagnóstico" de falhas. Porém, existe efetivamente a sua "deteção". Para serem toleradas falhas causadas internamente, ou seja, pelo *software* baseado em componentes, é necessário implementar interligações entre o modelo e o respetivo *software* de tempo-real. Assim, o modelo pode ser notificado de falhas internas à aplicação, sobre as quais, à partida, não se teria nenhum tipo de controlo. Podem ser toleradas falhas externas à aplicação: são toleradas falhas de *software* como erros em componentes do serviço, não cumprimentos de *deadlines*; e falhas ao

nível de *hardware* como corrupção de *links*, *crashes* de máquinas. A deteção de falhas é simples: caso ocorra uma falha num nó virtual ou físico, o respetivo nó parceiro terá esta perceção através da falta de uma *heartbeat* e trocará de papéis com o nó faltoso.

4 Implementação

Este capítulo descreve a implementação do modelo teórico descrito no capítulo anterior. Desenvolveu-se um sistema multiagente com o suporte de algumas bibliotecas. Optou-se por provar o conceito por simulação através da programação de raiz do modelo devido à natureza prática do problema. Para isso, recorreu-se à *framework* JADE que permite a criação e manipulação dinâmica de agentes inteligentes. No final do capítulo, é feita uma síntese acerca de aspetos relevantes desta plataforma, a implementação do modelo é detalhada e são identificadas as principais vantagens da sua utilização.

4.1 JADE

JADE (*Java Agent DEvelopment*) é uma plataforma *open-source* para desenvolvimento e execução de MAS, iniciada em 1998, que foi desenvolvida em Java e suporta programação de sistemas de agentes assíncronos, comunicação entre agentes, na mesma ou em diferentes plataformas, e utilitários para mobilidade e segurança, entre outros.

Ao nível de serviços básicos e infraestrutura para aplicações distribuídas multiagente nesta ferramenta encontram-se implementadas funcionalidades como *life-cycle* de agentes, serviço de *white & yellow pages*, para publicação de serviços, comunicação *multi-party*, transporte e *parsing* de mensagens ponto-a-ponto, escalonamento de tarefas multiagente e ferramentas auxiliares, com GUI, para *logging*, *debugging*, monitorização, gestão e controlo (*e.g. Sniffer Agent, Introspector Agent*, *Log Manager Agent*).

A *framework* JADE permite interoperabilidade através da sua alta compatibilidade com a FIPA. Possui dois níveis de modelação de concorrência: *inter-agent* (preemptiva: *threads* Java) e *intra-agent* (cooperativa/não preemptiva: classes *Behaviour*). Para além disso, a JADE suporta a clonagem de agentes.

Ao nível da comunicação, quando um agente recebe uma mensagem, esta é-lhe transmitida de diferentes maneiras, em função da localização do agente emissor na plataforma. Pode verificar-se o seguinte conjunto de situações e o respetivo tratamento:

- Quando o recetor se encontra no mesmo *container* que o emissor, a passagem é feita como evento;
- Quando ambos se encontram na mesma plataforma, mas em diferentes *containers*, a comunicação é feita por RMI;
- Quando ambos habitam em plataformas diferentes, é utilizado o protocolo IIOP para trocas de mensagens.

Recorreu-se à JADE para o desenvolvimento da simulação dado que esta plataforma facilita a criação de agentes e a execução de aplicações que são desenvolvidas por esta plataforma é independente da máquina (executa na *Java Virtual Machine*). Para além disso, o tamanho dos agentes, mesmo juntamente com o código, situa-se na ordem das dezenas de Kbs (depende também da com a quantidade de dados que o agente transporta consigo). Os problemas de escalabilidade tipicamente conhecidos em AOP não têm tanto impacto em arquiteturas *reactive*; caso os agentes sejam dotados de comportamentos determinísticos, isto é, ausentes de crenças ou desejos (comportamentos de caráter probabilístico).

4.1.1 FIPA

FIPA, iniciada em 1996, é uma associação internacional, sem fins lucrativos, pertencente ao IEEE (*Institute of Electrical and Electronics Engineers*), formada por empresas e organizações que desenvolve *standards* para tecnologias de *software* baseadas em agentes. Esta associação segue vários princípios, tais como:

1. Tecnologias baseadas em agentes constituem um novo paradigma para resolver problemas mais antigos e recentes;
2. Algumas tecnologias baseadas em agentes atingiram um certo nível de maturidade;
3. Os agentes requerem estandardização para serem utilizadas;
4. A estandardização de mecanismos internos de agentes não é um problema prioritário, mas sim a infraestrutura e linguagens, que são obrigatórias para a interoperabilidade aberta.

A JADE incorpora na sua infraestrutura diversas especificações propostas pela FIPA, cujo intuito consiste em estandardizar os métodos de coordenação e comunicação e disponibilizar serviços fiáveis e consistentes às entidades da plataforma. As seguintes definições são detalhadas num prisma FIPA. As relações entre alguns destes conceitos encontram-se representadas na Figura 4.4.

4.1.1.1 Agent

No âmbito da plataforma JADE, um agente é um processo computacional que vive numa AP (*Agent Platform*). Geralmente, oferece um ou mais serviços que podem ser publicados como uma descrição de serviço, através do serviço DF (*Directory Facilitator*), e implementam funcionalidades de comunicação autónomas numa aplicação. Os agentes são identificados por um *id* FIPA (AID), que é único numa plataforma, e comunicam entre si através de mensagens designadas ACL *messages*. Um agente pode ser registado em vários endereços de transporte, através dos quais é contactado.

4.1.1.2 *Agent Platform* (AP)

Uma AP consiste numa infraestrutura de máquinas e SOs onde são desenvolvidos os agentes. É um componente principal da *framework* que delimita a fronteira do sistema mais próximo. Uma única AP pode estar disseminada por vários *hosts* e assim, os seus agentes não necessitam de coabitar na mesma máquina. Por conseguinte, a localização de agentes que coabitam na mesma AP é transparente para a JADE e, consequentemente, para os seus agentes.

4.1.1.3 *Agent Management System* (AMS)

Este componente é um agente singular e obrigatório e encontra-se numa única AP. Por sua vez, uma AP possui apenas um AMS. O AMS é responsável por supervisionar o acesso e utilização da AP e por gerir, criar e eliminar agentes da plataforma. Mantém os AIDs de todos os agentes registados na sua AP e disponibiliza-os a outros agentes, através do serviço de *white pages*.

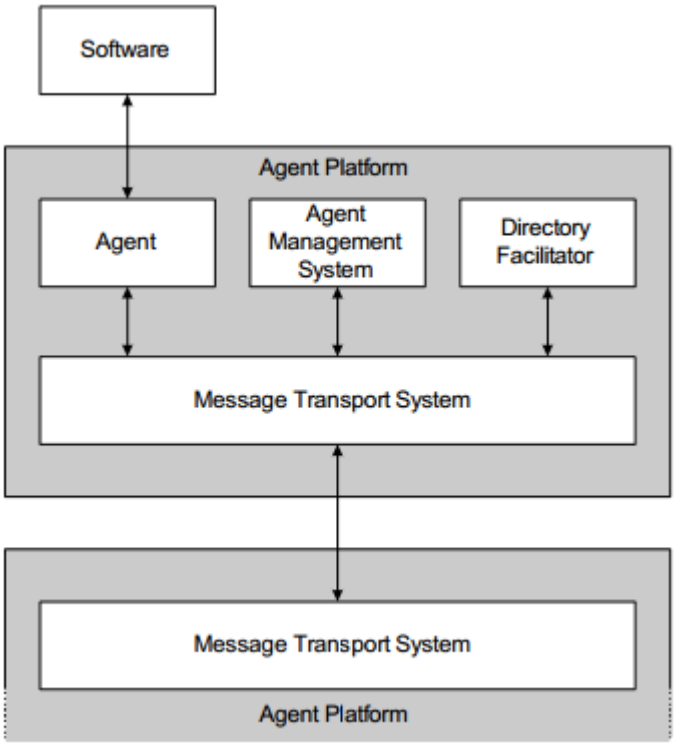


Figura 4.1 - Modelo referência FIPA de um AMS

[25]

4.1.1.4 *Agent Communication Channel* (ACC)

ACC é um agente que faculta caminhos de comunicação entre agentes, dentro ou fora da AP. É um serviço de comunicação fiável que proporciona integridade na ordem de chegada de mensagens.

4.1.1.5 FIPA-ACL *message*

As FIPA-ACL *messages* são as mensagens utilizadas pelos agentes para estabelecer diálogos entre si. As mensagens são transportadas num envelope que contém detalhes acerca da mensagem, *e.g.* conteúdo, data de envio, AID remetente, entre outros. A Figura 4.2 apresenta a estrutura genérica de uma mensagem FIPA-ACL.

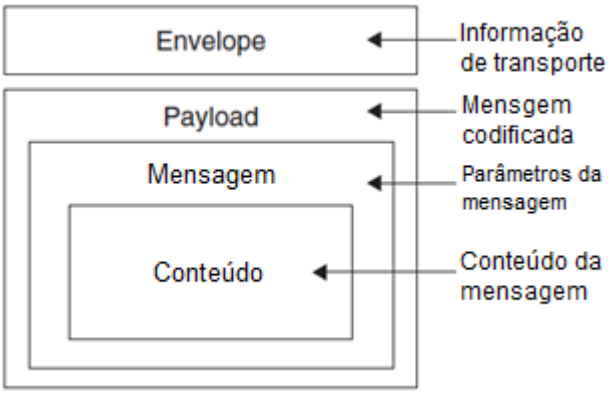


Figura 4.2 - Estrutura de mensagem FIPA-ACL

[8]

4.1.1.6 *Message Transport Service* (MTS)

MTS é um serviço fornecido pela AP, mais concretamente pelo seu ACC, para o transporte eficiente de mensagens FIPA-ACL entre agentes a nível intra-plataforma (dentro da mesma AP) e inter-plataforma (entre APs diferentes).

4.1.1.7 *Message Transport Protocol* (MTP)

Este protocolo é fornecido pelo MTS e é utilizado para a comunicação inter-plataforma. A JADE fornece MTPs adequados para comunicação através de HTTP e IIOP (protocolo de transporte definido pela CORBA). Exemplo de endereço MTP: http://oscar-laptop:7778/acc.

4.1.1.8 *Internal Message Transport Protocol* (IMTP)

Este protocolo é utilizado exclusivamente para a comunicação entre agentes que executam em diferentes *containers*, na mesma plataforma.

4.1.1.9 *Directory Facilitator* (DF)

DF é um componente opcional numa AP, mas, ao mesmo tempo, bastante útil. Este componente fornece serviços de *yellow pages*, *i.e.* publicação de serviços consumíveis para outros agentes e mantém as listas de serviços completas e atualizadas em tempo útil. Uma AP suporta inúmeros DFs, que se podem registar hierarquicamente para formar federações de serviços (*federated services system*). O DF permite que qualquer aplicação baseada em agentes funcione numa arquitetura *publish-subscribe*. Em suma, ao interagir com o agente DF, os agentes da plataforma conseguem expor serviços a outros agentes, mesmo de outras APs, como também pesquisar por serviços.

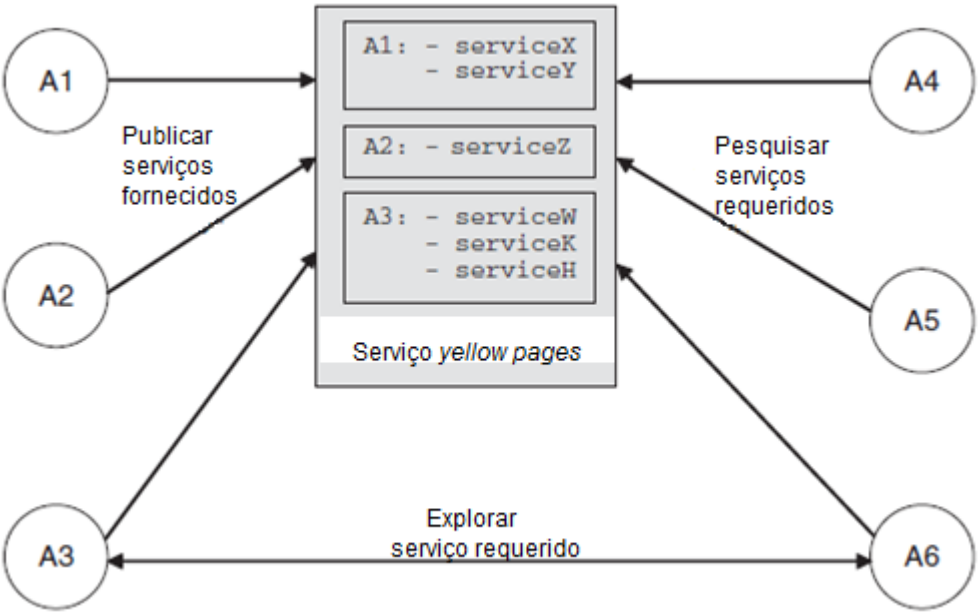


Figura 4.3 - Serviço *yellow pages*

[8]

4.1.1.10 *Remote Monitoring Agent* (RMA)

Este agente é o responsável por apresentar, na GUI, a consola gráfica para controlo da plataforma, que permite ao administrador da plataforma que manipule e monitorize a informação interna à AP de uma forma simples. Este componente permite a visualização da informação relativa ao estado de agentes e dos *containers*. Numa plataforma podem existir vários RMAs, ao passo que num *container* apenas pode existir um.

De notar que quando a plataforma JADE é lançada, os agentes AMS e DF são criados automaticamente. É também criado um terceiro agente, RMA. A Figura 4.4 apresenta a ontologia relativa ao funcionamento dos componentes descritos de 4.1.1.1 a 4.1.1.10.

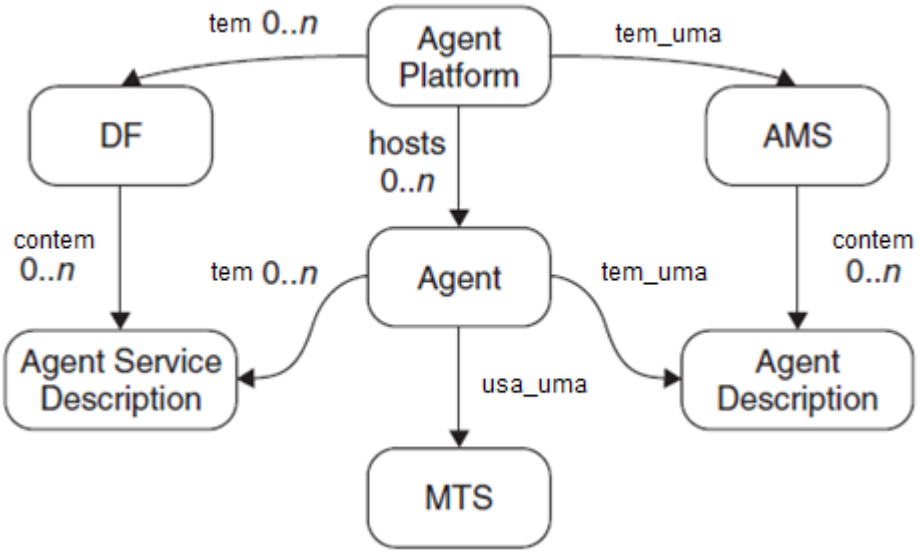


Figura 4.4 - Ontologia da gestão de agentes na JADE

[8]

4.1.2 Arquitetura

O modelo arquitetural da *framework* JADE é bastante simples, sendo constituído apenas por quatro entidades: agentes, *containers* e AP. A arquitetura da JADE define uma comunicação intrinsecamente ponto-a-ponto, pois a comunicação entre agentes é bidirecional, isto é, cada agente pode iniciar uma comunicação e pode também ser recetor, a qualquer altura.

Uma aplicação desenvolvida em JADE é constituída por coleções de componentes ativos chamados agentes. O AID é o identificador único de cada agente. Estes vivem num *container* que pertence a uma plataforma. Existe um *container* que atua como principal, denominado *main-container*, e é nele que se encontram alojados serviços como AMS e DF. O *main-container* reside na máquina onde executa o servidor RMI. Um conjunto de *containers* pertence a uma AP.

Num cenário exemplificativo, uma plataforma pode estar distribuída através de múltiplas máquinas e em cada uma destas está um *container*. No tópico 4.2 são pormenorizados diversos aspetos do funcionamento da *framework*, nomeadamente situações em que uma máquina adere a uma coligação.

A Figura 4.5 apresenta as relações entre os principais elementos da arquitetura.

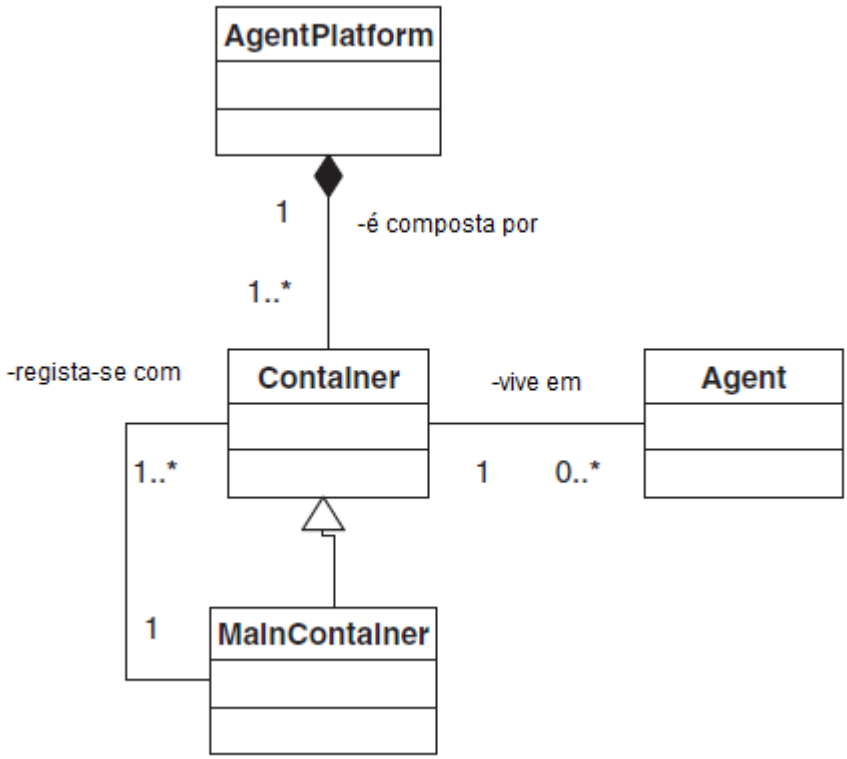


Figura 4.5 - Relações entre elementos da arquitetura

[8]

4.1.3 Mobilidade de agentes

A mobilidade de agentes é um assunto abordado no tópico 2.6 de forma concisa, sob um ponto de vista teórico. Neste tópico são mencionados alguns aspetos merecedores de destaque relativamente à integração de serviços de mobilidade na JADE.

A mobilidade de agentes (White, 1996) é um paradigma que abrange a IA e os sistemas distribuídos que define o conceito de mobilidade de código (Picco, 2000). O suporte à mobilidade de agentes que a JADE proporciona permite que estes se movam entre sistemas numa rede, de forma segura e a *framework* também se preocupa com aspetos como custo de migração e tenta minimizar ao máximo o impacto deste tipo de ações na rede.

Em termos de migração, os agentes podem mover-se de duas formas: *strong migration* ou *weak migration*. Na primeira, o estado do agente é congelado e, de seguida, o agente é movido para o *host* de destino. O processamento do código do agente é re-continuado após a migração, saltando para a próxima instrução. Esta técnica é a mais vantajosa e a utilizada pela JADE, embora requeira armazenamento e proteção de estado. Já no *weak migration*, os agentes são praticamente reiniciados após a migração, isto é, não mantêm estado.

Para a execução de uma determinada tarefa, os agentes podem necessitar de passar por diversos *hosts*. Esses caminhos percorridos pelos agentes designam-se por itinerários. Existem dois tipos de itinerários:

- *Static itineraries* – são determinados no momento da criação dos agentes, podendo ser modificados aquando da sua execução;
- *Dynamic itineraries* – são determinados durante a execução dos agentes de acordo com as suas necessidades e desejos.

Existem dois tipos de mobilidade: entre *containers* na mesma plataforma (*intra-platform*) ou entre diferentes plataformas (*inter-platform*). Relativamente à mobilidade intra-plataforma, a JADE fornece o serviço AMS que é o responsável por este tipo de operações. Quando inicia o processo de migração entre duas máquinas, o agente muda o seu estado de ACTIVE para TRANSIT. Quanto à mobilidade inter-plataforma, a JADE, por si só, não suporta essa funcionalidade. O *add-on* IPMS (*Inter-Platform Mobility Service*) consiste num mecanismo incorporado no *kernel* da JADE, que se encarrega de mover os agentes entre plataformas distintas e, para isso, utiliza mensagens FIPA-ACLs. A ontologia FIPA define duas ações para este tipo de migração: move (migração do agente) e power-up (reativação do agente, após a ação move estar completa). Durante a migração, no momento da saída de um *host*, um agente é serializado. De seguida, na sua chegada ao *host* de destino, o agente é de-serializado.

A Figura 4.6 apresenta um agente A a pedir ordem a um outro A', através do protocolo FIPA Request, para iniciar o processo de migração e prossecução da atividade.

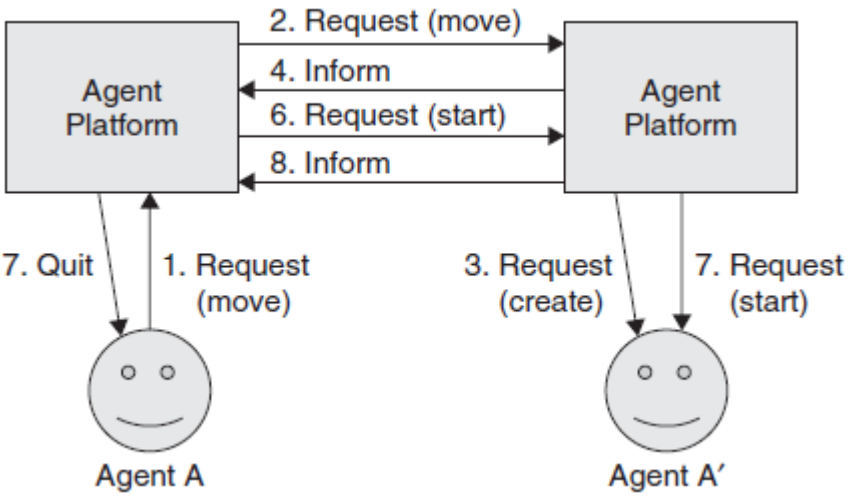


Figura 4.6 - Interação FIPA Request entre dois agentes

[8]

4.1.4 LEAP

O LEAP é um *add-on* da JADE que permite a execução de agentes JADE em dispositivos *lightweight*, como telemóveis que executam a plataforma Java ou a *framework* Microsoft .Net. O LEAP, combinado com a JADE, substitui algumas partes do *kernel* da JADE e forma um *runtime* modificado para dispositivos móveis *lightweight* (*tablets*, *smart phones* ,PDAs), denominado JADE-LEAP. Este *add-on* contém ainda os seguintes módulos[8] supletivos à *framework* JADE:

- *pjava* – para executar a JADE-LEAP em dispositivos móveis que suportem J2ME CDC;
- *midp* – para executar a JADE-LEAP em dispositivos móveis que suportem MIDP1.0 (ou versões superiores);
- *android* - para executar a JADE-LEAP em dispositivos móveis que suportem Android 2.1 (ou versões superiores);
- *dotnet* – para executar a JADE-LEAP em dispositivos móveis que suportem a *framework* Microsoft .Net 1.1 (ou versões superiores).

Figura 4.7 - Modos de execução da JADE-LEAP

[8] Existem algumas limitações do LEAP nos módulos PJAVA, MIDP e Microsoft .Net.

[26]

O JADE-LEAP pode ser executado em modo *stand-alone*, com um só dispositivo representado por um *container* que interage com o *main-container*, ou em modo *split*, com vários dispositivos a interagir com o mesmo, como a Figura 4.7 apresenta.

4.2 Modelação

No desenvolvimento da simulação foram tomadas algumas decisões a nível conceptual na tentativa de beneficiar o funcionamento deste modelo, adaptando-o ao ambiente da *framework* JADE. Todo o modelo baseia-se numa arquitetura *reactive* pois as reações dos agentes são despoletadas por estímulos externos e internos. Neste caso, os agentes efetuam o *listening* de outros componentes, como *containers*, e consultam outros agentes para compreender o estado do ambiente. Não se optou por desenvolver um modelo baseado em agentes com crenças nem desejos (BDI), pois isso tornaria os resultados produzidos de cariz probabilístico, e dadas as exigências temporais, é necessário obter resultados determinísticos.

Comparativamente com o modelo integralmente proposto no capítulo 3, na simulação apenas não foi implementada a replicação física, ou seja, todas as interações entre agentes advindas dessa funcionalidade, como também a parte das subcoligações.

4.2.1 Classes

A implementação em Java foi divida nos seguintes *packages*, para além da classe Main.java:

- agent – Este componente agrupa todas as classes relativas a agentes e *behaviours*: FTAgent.java, ControllerAgent.java, ExecutionAgent.java, ReckonAgent.java;
- qos - Este componente contem todas as estruturas de dados que possibilitam a descrição formal de qualidade de serviço: Level.java, Dimension.java, Attribute.java, Value.java, Proposal.java, QoSHandler.java, IQoSIndex.java, IUpDowngradable.java;
- service – componente que contem todas as classes que descrevem o grafo de um serviço: Service.java, Component.java, Connection.java, ServiceHandler.java;
- util – componente com classes utilitárias, criadas para facilitar o desenvolvimento do projeto: Util.java, Entry.java, NotDowngradableException.java, NotUpgradableException.java, RuntimeInfo.java.

É importante referir que Component, a classe que representa os nós do serviço Service, contem configurações de três níveis: o nível mínimo de execução ($L_{minimum}$), o nível preferido ($L_{desired}$) e o nível atual de execução ($L_{current}$), onde $L_{minimim} \leq L_{current} \leq$

$L_{desired}$. Os ramos do grafo são representados pela classe Connection que contém o valor da *deadline* (em milissegundos).

Para organizar a estrutura da implementação, na parte relativa à *package* qos foram criadas duas interfaces: IUpDowngradable e IQoSIndex. Estas são interfaces comuns a diversas classes do componente de QoS. A primeira indica se a classe é indexável do ponto de vista de qualidade de serviço, característica denominada *Quality Index* [27] e contém métodos para efetuar *upgrades* ou *downgrades* como outros para anular os efeitos das alterações de qualidade, através dos métodos rollbackUpgrade() e rollbackDowngrade(). A segunda indica se um objeto da classe é degradável ou passível de melhoramento de QoS.

As classes que implementam estas interfaces são Level e Dimension. A classe Attribute apesar de conter os métodos upgrade() e downgrade(), não implementa estas interfaces pois foi desenvolvida de maneira diferente das inicialmente mencionadas.

A Figura 4.8 apresenta o diagrama de classes do projeto implementado. O diagrama não contém todas as classes do projeto, apenas as mais importantes no âmbito do modelo. Nenhuma classe interna da JADE se encontra ilustrada nesta figura.

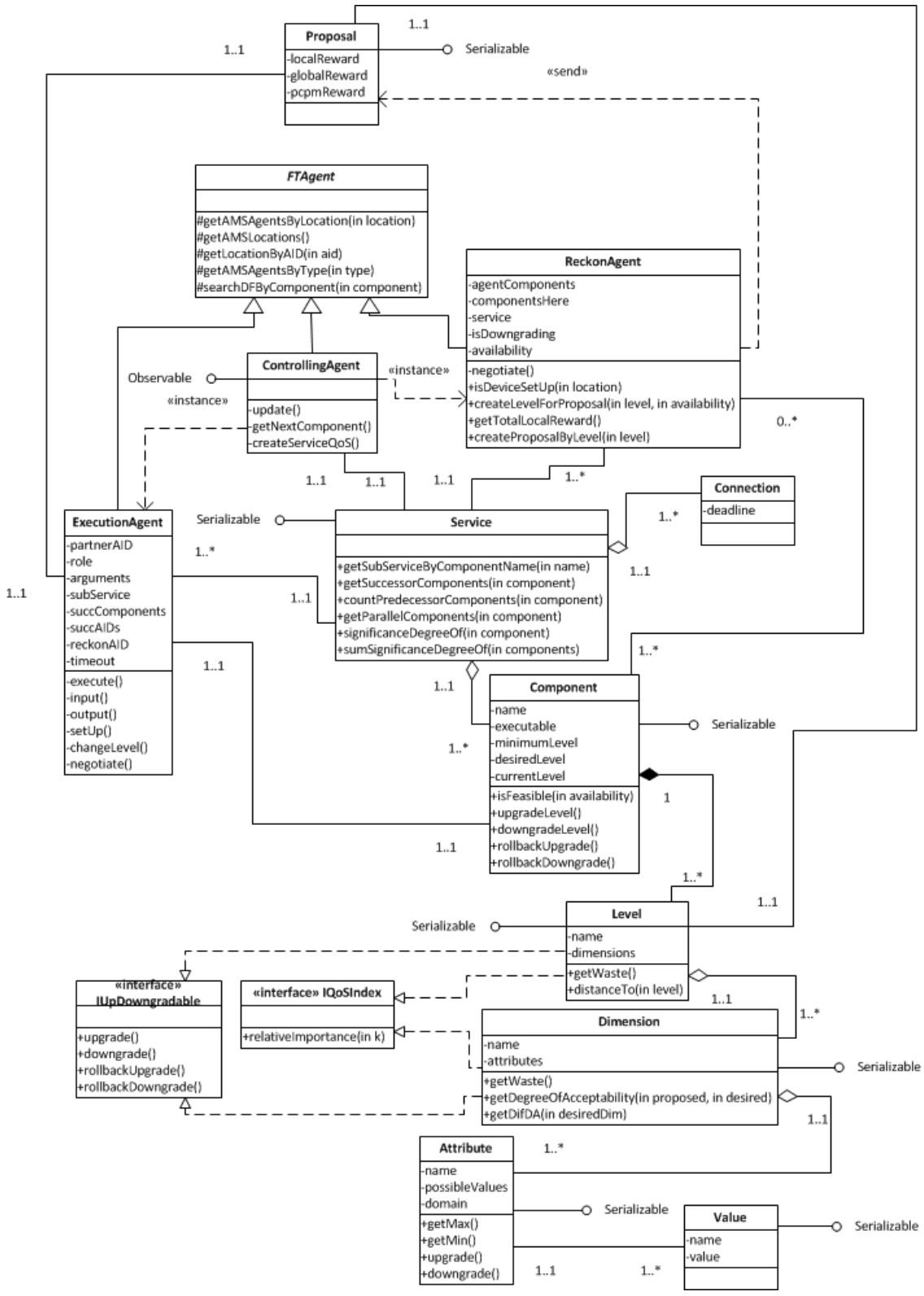


Figura 4.8 - Diagrama de classes

Na Figura 4.8, a classe FTAgent é abstrata e é superclasse das classes ControllingAgent, ReckonAgent e ExecutionAgent. ControllingAgent é responsável pela inicialização do grafo do

serviço (jgrapht-jdk1.6.jar) e das descrições de QoS - dados que são carregados a partir dos ficheiros de configuração. As classes ReckonAgent e ExecutionAgent correspondem à implementação dos agentes de reconhecimento e dos agentes de execução, respetivamente.

Para possibilitar a migração de agentes em conjunto com o seu código e dados, foi necessário que algumas classes implementassem a interface java.util.leap.Serializable. Esta interface é uma extensão da interface Serializable original do Java (em java.util), que foi desenvolvida de modo a permitir execução em dispositivos móveis, suportada pelo *add-on* LEAP (ver 4.1.4).

Todas as classes de *behaviours* foram implementadas dentro dos respetivos agentes. Para o processo de negociação entre ARs e AEs recorreu-se à aplicação do protocolo de interação FIPA Propose, para realizar a comunicação entre agentes: ProposeInitiator.java (jade.proto), nos agentes ReckonAgent, e ProposeResponder.java, nos agentes ExecutionAgent. O funcionamento deste protocolo *standard* é ilustrado na Figura 4.9.

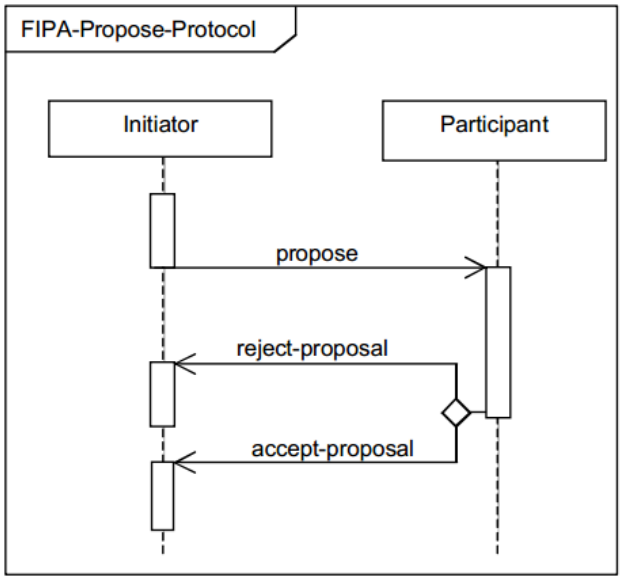


Figura 4.9 - Protocolo de interação FIPA Propose

[25]

O comportamento que trata do envio e receção de *heartbeats* entre agentes, definido na classe ExecutionAgent, é configurável, sendo assim possível parametrizar-se os valores de *timeout* e de frequência de transmissão de *heartbeats*.

O projeto conta também com a utilização das bibliotecas jade.jar, commons-codec-1..3.jar, commons-io-1.3.2.jar, commons-lang3-3.1.jar, jgrapht-jdk1.6.jar, json-lib-2.4-jdk15.jar, json-org.jar, json-simple-1.1.1.jar, migration.jar, sendo que algumas destas foram estendidas. A sua integração na implementação facilita o trabalho com grafos, linguagens de codificação e des-

codificação (*e.g.* base64), descrição de estruturas em linguagens *standard* e serviços de mobilidade de agentes.

Na implementação foram criados dois *handlers*: do serviço, ServiceHandler, e da descrição QoS, QoSHandler. Estas classes são utilizadas no arranque da simulação e servem para inicializar o serviço e as qualidades de serviço pretendidas, através de ficheiros de configuração XML e JSON. A classe RuntimeInfo devolve alguma informação relativa ao estado da máquina em termos de memória RAM, CPU, nº de núcleos e foi utilizada apenas para realização de testes.

4.2.2 Interações entre agentes

A negociação de componentes entre ARs e AEs é composta por quatro fases: o anúncio da máquina, feito por um AR, a notificação dos componentes, feita pelos AEs, a proposta das *rewards* juntamente com os níveis de QoS oferecidos pelo AR e, finalmente, as resposta dos AEs a essa proposta. O processo de negociação é formado por duas interações FIPA Propose. As duas primeiras fases correspondem a uma interação e as duas últimas fase a outra.

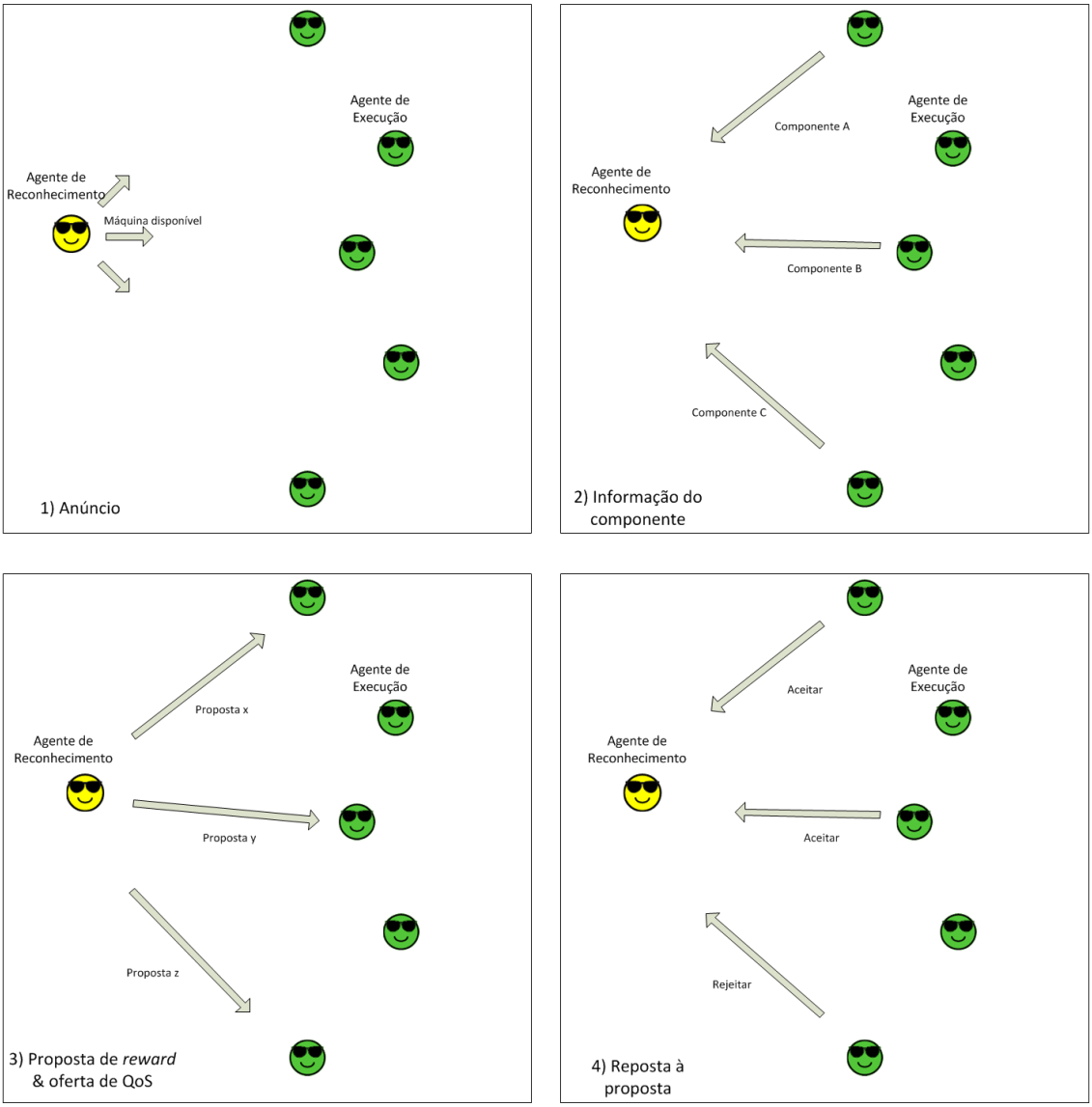


Figura 4.10 - Etapas de negociação de componentes

A Figura 4.10 mostra o processo de negociação de componentes. Nela são ilustradas as várias interações que constituem a coordenação *auction-bidding*. É importante referir que quando o AR recebe o nome do componente pelo qual negoceia, como já conhece o seu nível mínimo de QoS, verifica se essa execução é *feasible* na sua máquina, mediante a sua *availability*.

Sempre que uma negociação decorre e é finalizada com sucesso o AE migra para a máquina proponente juntamente com o seu componente para prosseguir a sua execução. Uma migração segue-se sempre de uma subida da qualidade da coligação, ou então de um passo intermédio que posteriormente permite efetuar a melhoria do QoS. Nos instantes após a migração, a nova máquina onde o AE se hospeda sofre um decréscimo na sua *availability*, pelo facto de se encontrar com mais um componente, enquanto na máquina anterior a *availability* aumenta. Tal como a *availability*, o valor que representa o consumo de re-

cursos (*waste*) deste novo componente é simulado. As seguintes equações, da (4.1) à (4.3), calculam, em conjunto, o gasto simulado que um componente *c* provoca após o início da sua execução num *host t*:

$$wasteLevel(L_{current}) = \frac{\sum_{i=1}^{d}\left(\frac{\left(\frac{wasteDim(dim_i)}{\omega_i}\right)}{d}\right)}{d} \quad (4.1)$$

$$wasteDim(dim_i) = \frac{\sum_{j=1}^{a}\left(\frac{\left(\frac{valuePercent\left(attr_j(value)\right)}{\omega_j}\right)}{a}\right)}{a} \quad (4.2)$$

$$valuePercent\big(attr_j(value)\big) = \frac{100}{a} \times (a - pos(value)) \quad (4.3)$$

Onde:

- $L_{current}$ é o nível atual de execução;
- d é o número de dimensões do nível $L_{current}$;
- ω_k é a importância relativa de uma elemento de QoS k (dimensão ou atributo);
- $wasteDim(dim_i)$ é o gasto simulado de uma dimensão dim_i;
- $valuePercent(attr_j(value))$ é a percentagem por valor $value$ para um atributo $attr_j$;
- a é o número de atributos de uma dimensão dim_i;
- $pos(value)$ é a posição de um valor $value$ no vetor valores possíveis de $attr_j$.

A percentagem por valor $valuePercent(\dots)$, de um determinado atributo, é calculada em função da posição que esse valor ocupa no vetor de possíveis valores que esse atributo pode assumir. Considerando uma especificação de preferência de QoS em ordem decrescente, dado um atributo ao qual se possa atribuir 5 valores possíveis, o 1º valor possível, o preferido, tem um valor de percentagem de 100%. Já o 2º valor possível, o segundo mais preferido, tem um valor de 80%.

Para facilitar a compreensão das interações mais relevantes na formação e manutenção da coligação são ilustradas algumas interações em diagramas de sequência. Os conjuntos de interações formam diálogos que têm objetivos independentes: a negociação de componentes, a coordenação de *upgrades* e *downgrades* (QoS), a execução e coordenação de

inputs e *outputs* dos AEs. As interações seguintes são representadas sob um ponto de vista conceptual, para simplificar os processos de comunicação entre agentes. A Figura 4.11 apresenta um digrama de sequência que ilustra a inicialização e execução dos componentes do serviço.

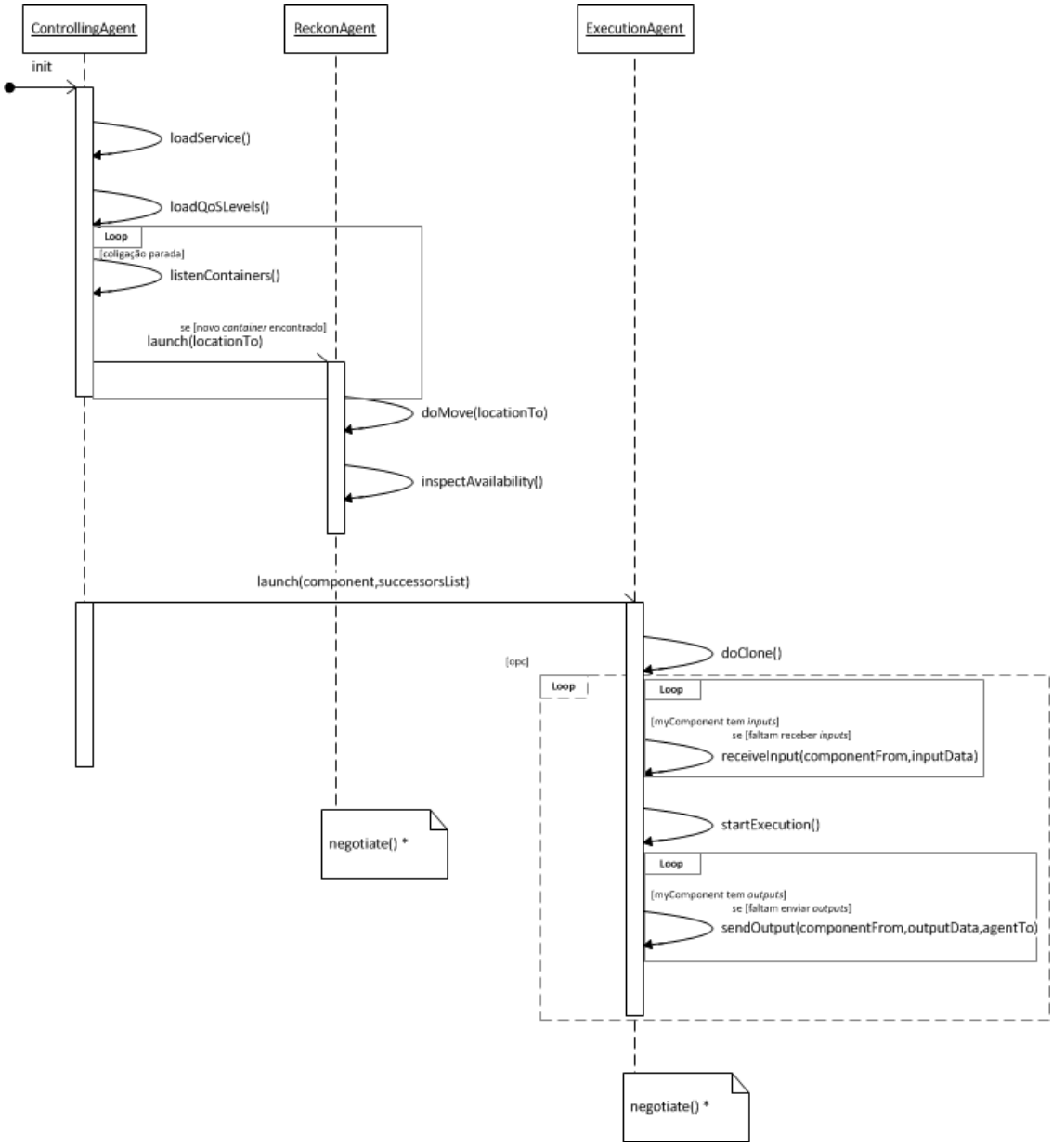


Figura 4.11 - Diagrama de sequência da inicialização e execução

Os agentes do tipo ControllingAgent, ReckonAgent e ExecutionAgent efetuam uma série de interações entre eles, cada uma com diferentes objetivos. Após o ControllingAgent proceder ao carregamento do serviço e da descrição de QoS para memória, instancia todos os agentes de execução. À medida que vai recebendo notificações de adesão de máquinas à AP, cria os ARs para estes se instalarem numa das máquinas aderentes, ainda sem AR, e

para calcularem a sua *availability*. Para se simplificar a obtenção de um valor para a *availability*, esta é gerada aleatoriamente com valores entre 60 e 99%.

Na linha temporal do ExecutionAgent, o *loop* de execução do componente é opcional, pois este pode ser executado apenas uma vez ou então ciclicamente. Quando os ExecutionAgent são inicializados, já com um componente atribuído, o ControllingAgent passa-lhes um parâmetro que indica se o seu componente recebe *input*, e se sim, quantos parâmetros. Desta forma o AE sabe se antes de executar tem de receber *input* de um outro componente. Relativamente ao *output*, os agentes de execução possuem uma lista com os sucessores no grafo. Os agentes de execução são responsáveis por publicar o nome dos seus componentes, como foi referido nas suas responsabilidades na Definição 2. Portanto, através do serviço DF conseguem consultar quais os respetivos *outputs* para o seu eventual conjunto de componentes de saída.

O ExecutionAgent clona-se de imediato e começa a sua execução. De seguida, procede juntamente com o ReckonAgent à parte da negociação, apresentada na Figura 4.12.

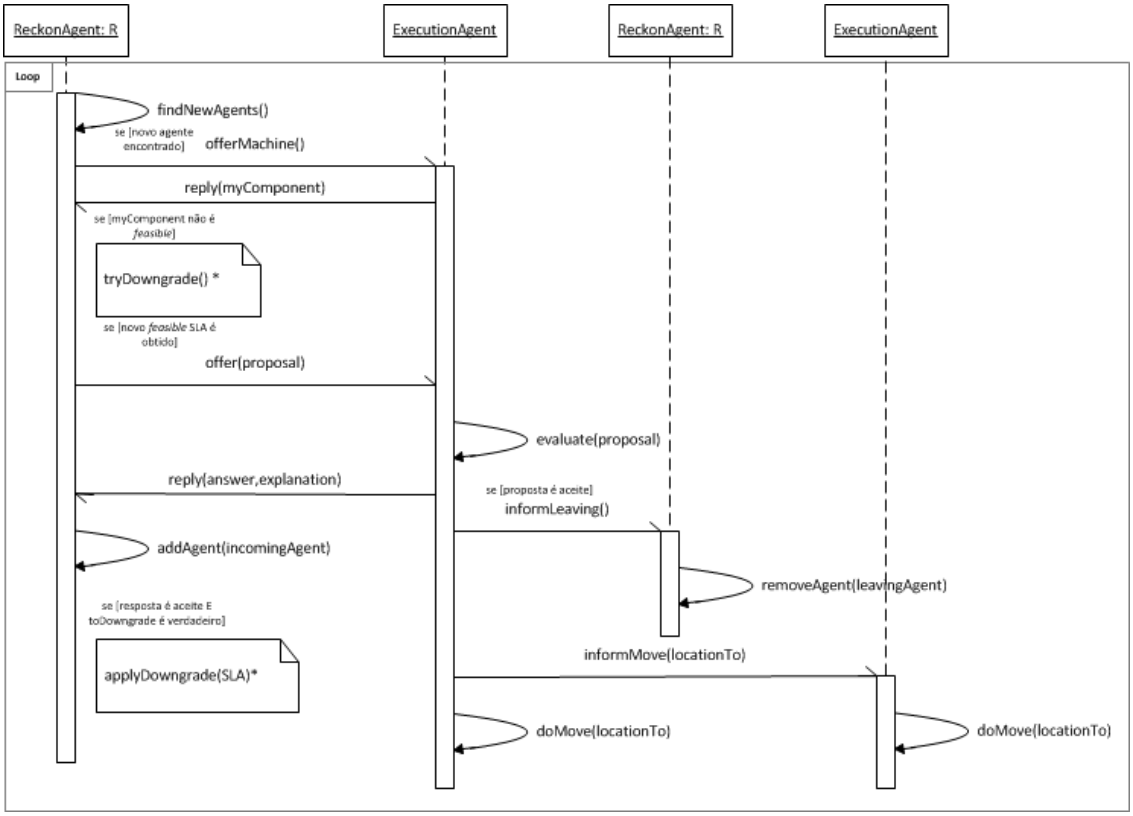


Figura 4.12 - Diagrama de sequência da negociação

O processo ilustrado na Figura 4.12 é executado ciclicamente e inicia-se com a procura de agentes ExecutionAgent por parte de um ReckonAgent R_B. Quando encontra um AE primário E_{A_1} que não se encontra na sua *queried list*, R_B inicia a negociação com este. Se a *feasibility* da sua máquina não permitir a execução do componente de E_{A_1}, R_B tenta calcular um

novo SLA para os seus outros componentes de forma a receber E_{A_1} e a sua réplica[9] E_{A_2} (tryDowngrade()*[10] - Figura 4.13). Se a proposta for aceite, E_{A_1} informa R_A que vai sair do *container* da sua máquina, e R_A remove-o da sua *queried list*. Já R_B adiciona E_{A_1} à sua *queried list* e, no caso da ingressão de E_{A_1} na máquina de R_B exigir um determinado *downgrade*, R_B aplica-o imediatamente (applyDowngrade(SLA)* - Figura 4.13). Antes de migrar, E_{A_1} informa a sua réplica E_{A_2} para se mover, indicando o *container* de destino, através do parâmetro locationTo.

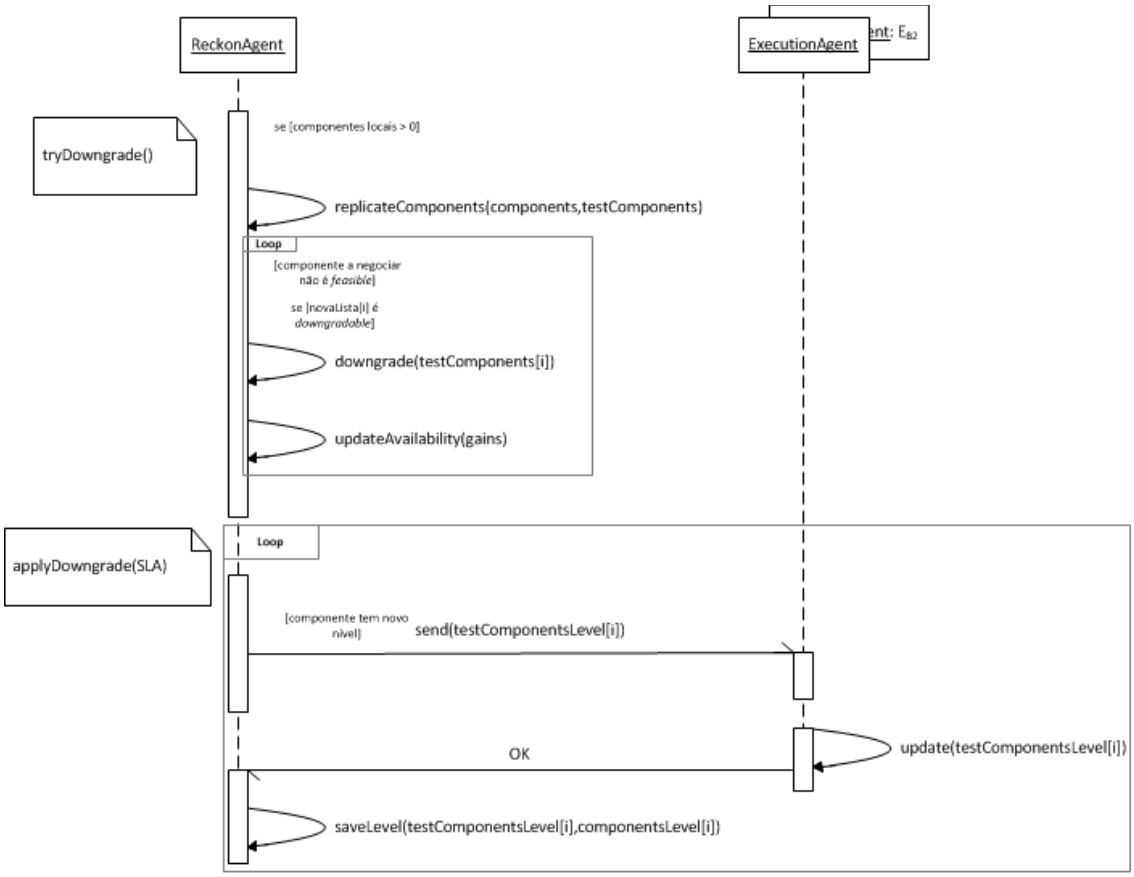


Figura 4.13 - Diagrama de sequência de *downgrades*

A Figura 4.13 apresenta o diálogo entre agentes ReckonAgent e vários ExecutionAgent ($E_{A_1}, E_{B_2}, \ldots$). Esta conversação tem o objetivo de estabelecer um novo SLA de QoS, que seja consensual entre os agentes da coligação. Os ReckonAgent enviam uma mensagem para os ExecutionAgent com um objeto Level, que contem o novo nível calculado para cada componente.

[9] Numa perspetiva da *framework* JADE é indiferente se E_{A_1} é o original ou o clone; o mesmo se aplica a E_{A_2}. A notação refere-se apenas ao papel assumido pelo agente (*1*: primário; *2*: secundário).

[10] tryDowngrade() e applyDowngrade(SLA) não correspondem a métodos reais na implementação. Designam apenas conjuntos de ações com nomes sugestivos e conceptuais. Daí não serem referidos no diagrama de classes (Figura 4.8).

Na primeira parte do diagrama de sequência (tryDowngrade()), o ReckonAgent replica os componentes que se encontram no *container* da sua máquina criando uma imagem virtual deste conjunto juntamente com os seus níveis de QoS. De seguida, tenta efetuar *downgrades* ao nível de QoS de cada componente até que a *feasibility* seja suficiente para aceitar o componente negociado. Os componentes são replicados em memória para que os testes de *downgrade* sejam experimentais e não sejam realizados no mapa de componentes-níveis de QoS real que o ReckonAgent possui. Se efetivamente se conseguiu obter um novo SLA para o grupo dos componentes do ReckonAgent, na segunda parte (applyDowngrade(SLA)) então aplicam-se essas alterações aos níveis de QoS dos agentes E_{A_1}, E_{B_2}, e aos restantes, que se encontrem no *container* desse AR. As alterações são enviadas aos agentes aos quais foi aplicada a degradação da qualidade, e são repercutidas no mapa componente-nível de QoS do ReckonAgent para este manter a informação relativa aos seus AEs atualizada.

Recorde-se que o processo de degradação de serviço só é desencadeado em última instância, isto é, apenas se a execução de um componente não for *feasible* na máquina do AR proponente.

À semelhança dos *downgrades*, os *upgrades* são processados segundo a mesma lógica. No entanto, ao contrário da degradação de QoS, a melhoria é tentada persistentemente pelos ReckonAgent. A Figura 4.14 apresenta o processo de *upgrade* entre um ReckonAgent e um conjunto de agentes ExecutionAgent ($E_{A_1}, E_{B_2}, \ldots$).

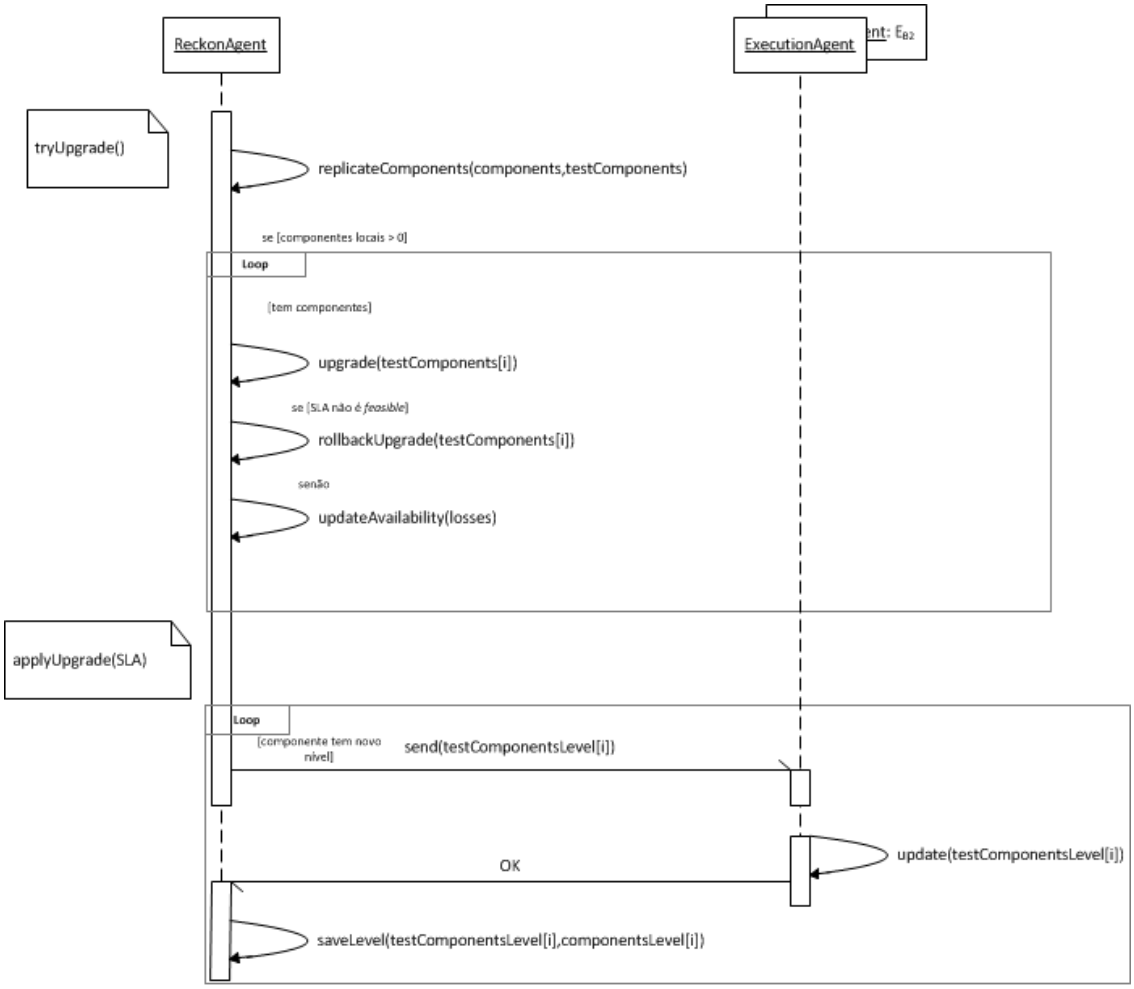


Figura 4.14 – Diagrama de sequência de *upgrades*

Da mesma maneira, na parte tryUpgrade() o ReckonAgent tenta calcular um novo SLA para o conjunto dos ExecutionAgent, experimentando as alterações numa réplica fictícia deste conjunto. Na segunda parte (applyUpgrade(SLA)), o SLA passa a ter efeito para os respetivos AEs reais.

De referir que se um SLA não viabilizar o acordo numa negociação para a migração do AE, então nunca entra em vigor, visto que não beneficia a qualidade produzida pela coligação.

4.3 Testes

A fase de testes tem o intuito de avaliar a qualidade do modelo desenvolvido através dos resultados produzidos. Neste tópico os resultados são analisados, quer em termos de QoS fornecido aos componentes como em termos temporais, isto é, ao nível do cumprimento de *deadlines*.

Foram realizados diversos tipos de testes, todos eles incidindo em diferentes aspetos-chave relevantes para a análise do modelo proposto. O que realmente é crítico e objetivo principal no funcionamento do modelo, para além da ativação de todos os componentes o mais rapidamente possível, é o cumprimento de todas as *deadlines*, especificadas no ficheiro que descreve a planta do serviço (service.xml), ainda que todos os componentes estejam a executar no nível mínimo de QoS. O *schema* XSD associado ao ficheiro service.xml é apresentado na Figura 4.15.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="service">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="connections" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="connection" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="origin" type="xs:string"/>
                  <xs:element name="destination" type="xs:string"/>
                  <xs:element name="deadline" type="xs:unsignedLong"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Figura 4.15 – *Schema* XSD do ficheiro service.xml

Assim sendo, o cumprimento das *deadlines* é um indicador de performance que deve ser tido em consideração e que merece especial atenção na avaliação dos testes. Adicionalmente, é importante que as dimensões e os atributos de QoS prediletos para o utilizador sejam privilegiados no fornecimento de qualidade de serviço.
Para simplificar o processo de testes foram adicionados ao ficheiro de propriedades algumas variáveis que podem ser parametrizadas tais como *timeouts* para a negociação nos ExecutionAgent e ReckonAgent, *timeouts* para a receção, frequência de envio de *heartbeats* entre parceiros de execução, *delays* para pesquisa de novos ExecutionAgent nos ReckonAgent, e para procura de componentes para envio de *outputs*, nos ExecutionAgent. É importante referir que em todos os testes, a execução da simulação produzia *logs* para ficheiros de texto, para facilitar a traçagem do funcionamento do modelo. Essa ativação de *logs* prejudica inevitavelmente o funcionamento da simulação, afastando, ainda que escassamente, a qualidade dos resultados comparativamente aos que são produzidos num cenário mais realístico, isto é, sem a criação de *logs* de traçagem.

Em alguns dos testes foram utilizadas máquinas virtuais. Nestes casos é o *host* físico *Laptop 1* que suporta a execução destas máquinas e que pede a formação da coligação. As características das máquinas, físicas e virtuais, utilizadas para a realização dos testes encontram-se apresentadas na Tabela 4.1.

Máquina	CPU	RAM	Rede
Laptop 1	Intel Core i5, 2.53 GHz	4.0 GB DDR3	*Wireless* IEEE 802.11n, 72.0 Mbps
Laptop 2	Intel Core i5, 2.50 GHz	6.0 GB DDR3	*Wireless* IEEE 802.11n, 72.0 Mbps
PC	Intel Pentium 4, 2.66 GHz	1.792 GB DDR1	*Ethernet* 100.0 Mbps
vmA	N/a (1 *core*)	456 MB	*Auto-ethernet* partilhado por NAT
vmB	N/a (1 *core*)	456 MB	*Auto-ethernet* partilhado por NAT
vmC	N/a (1 *core*)	456 MB	*Auto-ethernet* partilhado por NAT

Tabela 4.1 - Recursos de máquinas utilizadas em testes

Foi utilizado o protocolo NTP para sincronização dos relógios de todas as máquinas. O NTP foi configurado com o servidor time.windows.com e foi definido um *fudge* de 10 (parâmetro de sincronização de tempo).

No que toca à avaliação de tempos de migração de agentes, foram efetuados dois tipos de testes: um no âmbito da recolha do tempo de migração de um agente e o outro para recolha do tempo de migração do agente original *versus* o agente clone. No primeiro verificou-se através do registo dos tempos que os agentes ReckonAgent eram mais lentos na migração do que agentes ExecutionAgent. Estes resultados eram expectáveis visto que os ARs têm de armazenar mais informação e processar mais cálculos, para efeitos de coordenação do modelo.

A Tabela 4.2 apresenta alguns resultados relativos a tempos de migração de agentes.

De	Para	Agente	Tempo (ms)
Laptop 1	*PC*	RECKON2	616
Laptop 1	*PC*	EXECUTION1	83

Laptop 1	*PC*	EXECUTION2	237
PC	*Laptop 1*	EXECUTION1	93
Laptop 1	*PC*	EXECUTION4	76
PC	*Laptop 1*	EXECUTION2	101
Laptop 1	*vmA*	RECKON2	512
Laptop 1	*vmB*	RECKON3	639
Laptop 1	*PC*	RECKON4	537
Laptop 1	*vmA*	EXECUTION3	800
Laptop 1	*vmB*	EXECUTION1	69
vmB	*Laptop 1*	EXECUTION1	629
PC	*Laptop 1*	EXECUTION1	166

Tabela 4.2 - Tempo de migração de agentes

Como já foi referido, após um determinado AE findar uma negociação em que o nível de QoS proposto é aceite, informa a sua réplica de que vai migrar e envia-lhe o *container* para onde a réplica o deve acompanhar. Como é evidente, ocorre sempre um atraso na migração do original relativamente ao clone.

A Tabela 4.3 apresenta alguns exemplos de atrasos recolhidos em testes para avaliação de tempos de migração entre agente original e réplica.

Original	**Clone**	**Diferença (ms)**
EXECUTION1	EXECUTION1-clone	87
EXECUTION2	EXECUTION2-clone	298
EXECUTION4	EXECUTION4-clone	54

Tabela 4.3 - Diferença de tempos de migração (original *vs* clone)

Conclui-se que este atraso não é significativo pois não tem qualquer impacto no funcionamento da execução primária do serviço, cujo ciclo produz os resultados efetivos, recebidos pelo utilizador. Mesmo em casos de maior exigência temporal, com prazos mais restritos, se um AE primário falhar, a duração da execução do componente no seu clone é encurtada desde o momento em que o primário falha, ao contrário do que aconteceria com réplicas passivas. Isto acontece pois o clone já iniciou a execução na sua máquina anterior, ainda que com um nível inferior de QoS. Logo, o cumprimento da *deadline* está efetivamente ao alcance desta nova réplica primária. Quando a réplica retoma a sua execução na nova máquina, só na iteração seguinte é que assume o novo nível de QoS proposto.

Foram realizados dois testes à memória RAM das máquinas aderentes à coligação, em função do tempo. Em ambos os testes o serviço é composto por 5 componentes. A Figura 4.16 ilustra o consumo de memória RAM (MB) feito pela simulação ao longo da sua execução. Neste primeiro teste é utilizada uma configuração de QoS mais simples, apenas com uma dimensão denominada "geração de caracteres por segundo". Como se pode visualizar, na representação gráfica a memória RAM ocupada é extremamente maior no *host* local, que despoleta a formação da coligação, do que nas máquinas que aderem posteriormente, cuja memória RAM utilizada não ultrapassa os 7 MB. Sempre que uma nova máquina entra na coligação dá-se um aumento bastante acentuado na quantidade de memória RAM ocupada no *localhost*. Acredita-se que o sucedido deve-se à inicialização dos agentes ReckonAgent e à atividade algorítmica intensa na sua fase inicial: carregamento e instanciação do serviço e descrição de QoS, comunicação com a AMS na busca de localizações (*containers*) e de agentes e interações com agentes ExecutionAgent.

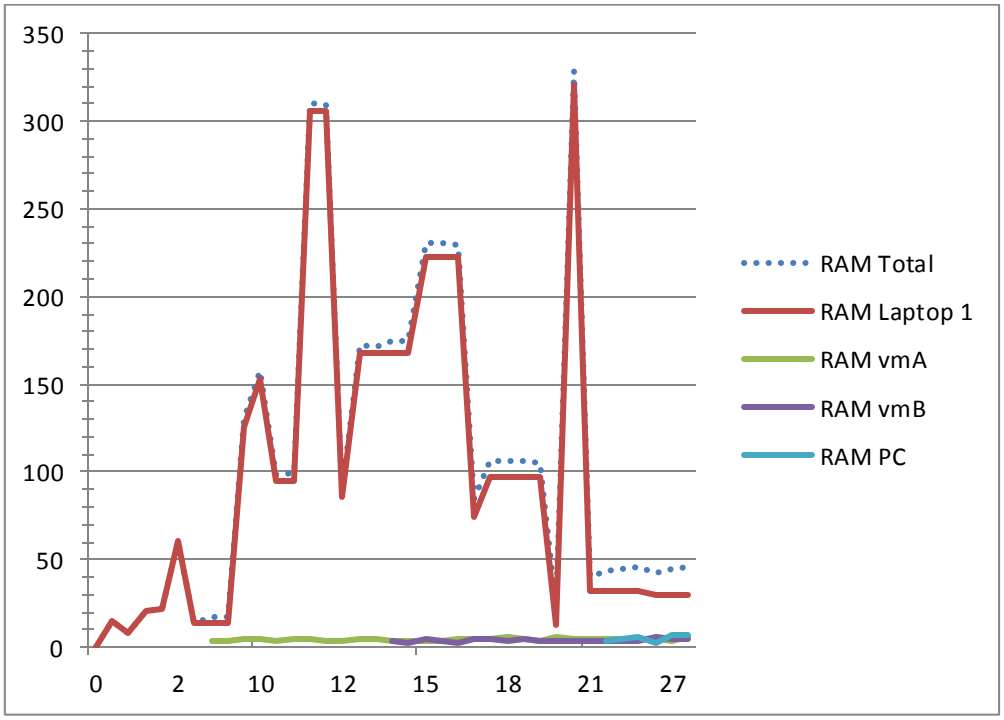


Figura 4.16 - Memória RAM com configuração de QoS simples

No segundo teste, utilizando uma configuração mais complexa, verificam-se igualmente a ocorrência súbita de aumentos momentâneos na memória do *Laptop 1*. Nas outras máquinas a memória RAM ocupada varia entre os 3 e os 7 MB.

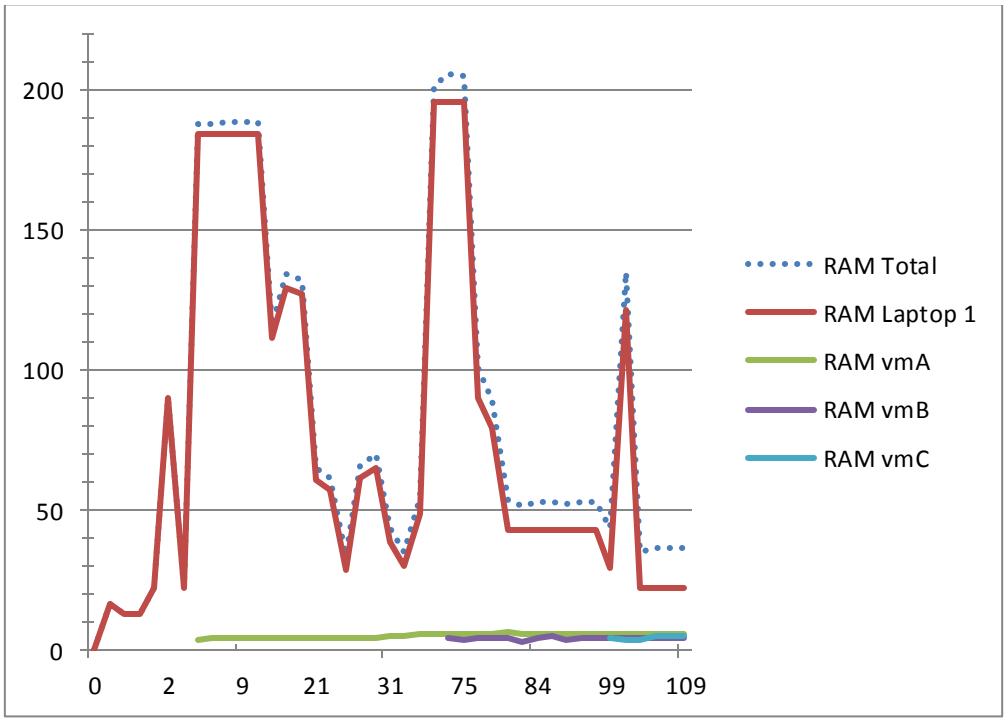


Figura 4.17 - Memória RAM com configuração de QoS de multimédia

O facto de sobrevirem alterações bruscas na memória RAM ocupada no Laptop 1, bem como de esta ser maior do que nos outros *hosts*, também se deve, em parte, à gestão de memória na *framework* JADE. Através da observação dos gráficos, é possível perceber, que é precisamente nos momentos em que as outras máquinas criam um *container* e aderem à AP do *localhost* (*Laptop 1*) que ocorrem aumentos na sua RAM ocupada. Através da análise dos testes é possível concluir que uma configuração de QoS mais complexa não tem um impacto significativo na memória RAM ocupada num membro da coligação.

Uma estimativa fiável e consistente por que se pode guiar a avaliação do modelo é a qualidade geral produzida pela coligação (ver Definição 8). O seu cálculo é semelhante ao da *local reward*, partindo do grau de significância do componente e do nível de serviço que lhe é fornecido. Cada agente de reconhecimento possui a informação necessária para calcular a sua *local reward* oferecida aos seus agentes de execução, num dado momento. Tudo junto resulta num valor que indica a qualidade geral do serviço que é produzida pela coligação.

Este tipo de teste tem em vista a obtenção da qualidade da coligação em função do tempo, dado um conjunto de *availabilities*. Tal como nos anteriores, realizou-se um teste com uma configuração de QoS simples e outro com uma configuração de QoS mais elaborada.

Na Figura 4.18 é apresentada a qualidade produzida por uma coligação, utilizando uma configuração de QoS simples. Neste gráfico são também apresentados os instantes em que uma nova máquina se junta à coligação e os instantes em que são ativados cada um dos componentes que formam o serviço, que correspondem ao início das suas execuções.

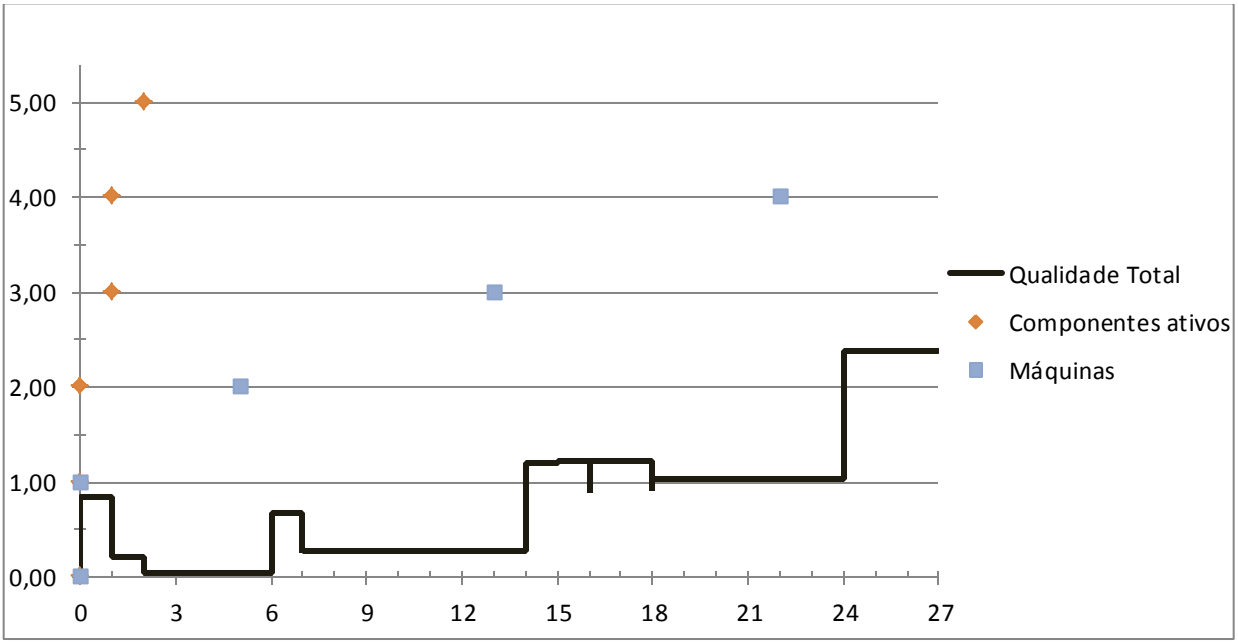


Figura 4.18 - Qualidade da coligação com configuração de QoS simples

As máquinas que participaram na formação desta coligação foram, por ordem de entrada, *Laptop 1*, *vmA*, *vmB* e *vmC*, com as *availabilities* de 85%, 65%, 96% e 74%, respetivamente, resultando numa média $\bar{x}_A = 80{,}00\%$.

Inicialmente, o *host Laptop 1* consegue negociar e iniciar com sucesso os 5 componentes do serviço, aos 3 segundos após o início da formação da coligação. Não obstante sofrer algumas variações no sentido positivo e negativo, a qualidade fornecida pela coligação é iterativamente melhorada. No início a qualidade total diminui, à medida que os componentes são iniciados e ativados, pois a coligação é composta apenas por uma máquina, o que significa que o balanceamento de carga é pobre. Como se pode observar neste gráfico e no seguinte, sempre que uma nova máquina se torna membro da coligação a qualidade aumenta rapidamente. Finalmente, 4 segundos após a adesão da última máquina, aos 27 segundos, a coligação encontra-se no seu estado final. Efetivamente, neste instante, todos os ARs já interrogaram com sucesso todos os AEs, logo o número de agentes na *queried list* de cada AR é igual ao total de AEs instanciados na coligação. A qualidade final é a máxima obtida ao longo da execução, e tem o valor de 2,380.

A Figura 4.19 apresenta os dados de um teste com uma configuração de QoS mais complexa relativamente ao primeiro. As *availabilities* das máquinas *Laptop 1*, *vmA*, *vmB* e *vmC* são, respetivamente, 85%, 65%, 96% e 74%, com uma média $\bar{x}_A = 75{,}75\%$. Neste teste a coligação converge em menos de 1 segundo após a entrada da última máquina na coligação, cerca de 100 segundos após o arranque da simulação. Foram realizadas um total de 44 negociações entre os agentes. *Laptop 1* processa 24 negociações, *vmA* processa 7, *vmB* 8 e *vmC* 5.

Conclui-se que, numa fase inicial da formação da coligação, uma máquina que contenha bastantes componentes é a que negoceia mais vezes, após a adesão de novas máquinas.

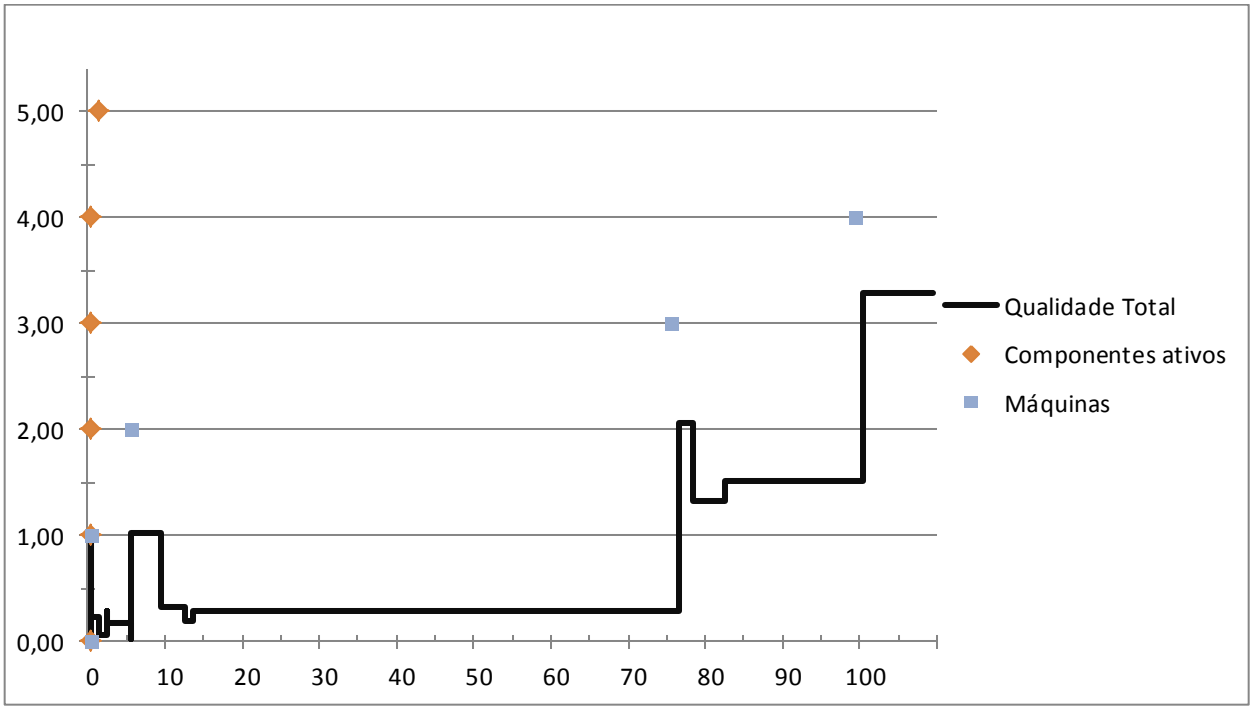


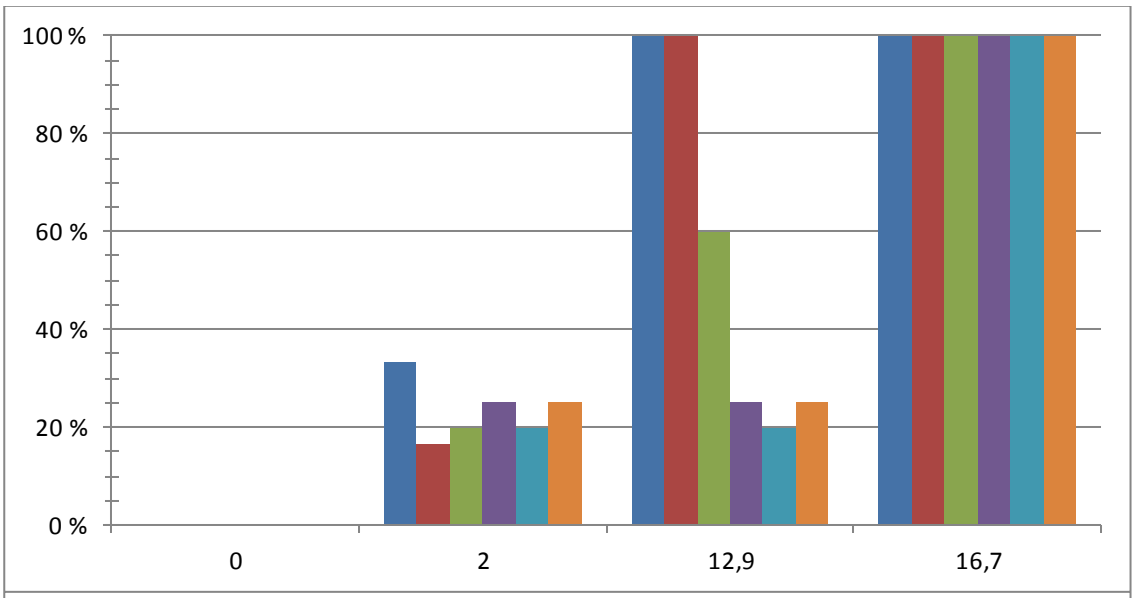
Figura 4.19 - Qualidade da coligação com configuração de QoS de multimédia

Através da realização destes testes foi possível concluir que no final da sua formação, independentemente da disponibilidade das máquinas, a qualidade fornecida pela coligação tende a aumentar consideravelmente. No entanto, existem algumas contrapartidas. Em primeiro lugar, algumas máquinas executam um único componente, cujo nível de QoS é muito elevado, ao passo que, em máquinas que hospedam vários componentes, estes são praticamente todos executados com o nível mínimo de QoS. Na simulação não se considera a priorização do negociamento dos componentes que ainda não se encontram a executar. A inconveniência que advém desta situação é a eventualidade de, numa fase inicial, os ARs recém-ingressados na coligação negociarem por componentes que já se encontram ativos em detrimento de outros que ainda não o estão. Desta forma, podem ocorrer atrasos na execução do serviço.

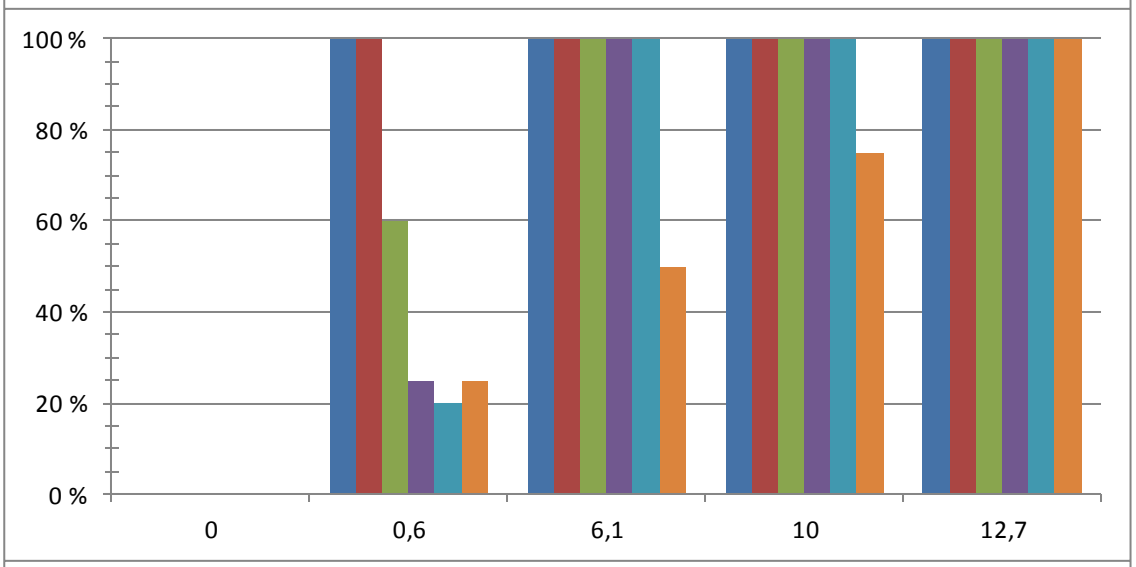
Foram realizados testes no âmbito da avaliação dos níveis de QoS configurados em cada componente. Com isto pretende-se avaliar os níveis produzidos individualmente por cada componente, durante a execução do serviço, mostrando que a qualidade fornecida é melhorada, em função do número de máquinas aderentes e das suas *availabilities*. Estes dados foram retirados da mesma execução que compõem o teste anteriormente referido, apresentado Figura 4.19, logo, para efeitos de análise, as circunstâncias foram as mesmas – máquinas, *availabilities* e tempos.

A Figura 4.20 apresenta 5 gráficos de barras com a variação dos níveis de QoS fornecidos a cada um dos 5 componentes, desde o início da formação da coligação até à sua convergência.

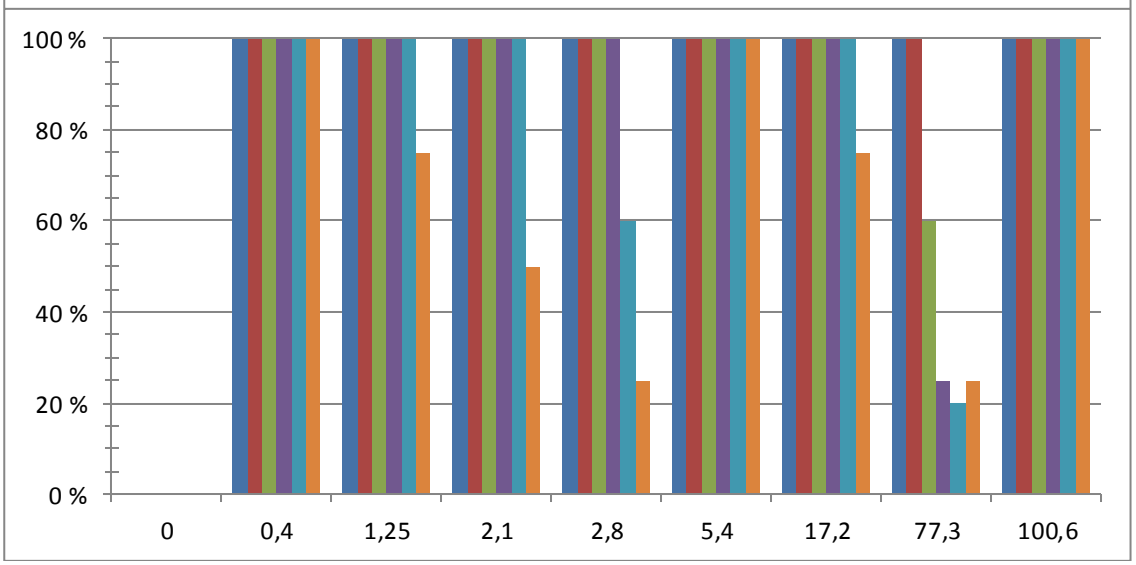
A


100 %
80 %
60 %
40 %
20 %
0 %
0
2
12,9
16,7

B


100 %
80 %
60 %
40 %
20 %
0 %
0
0,6
6,1
10
12,7

C


100 %
80 %
60 %
40 %
20 %
0 %
0
0,4
1,25
2,1
2,8
5,4
17,2
77,3
100,6

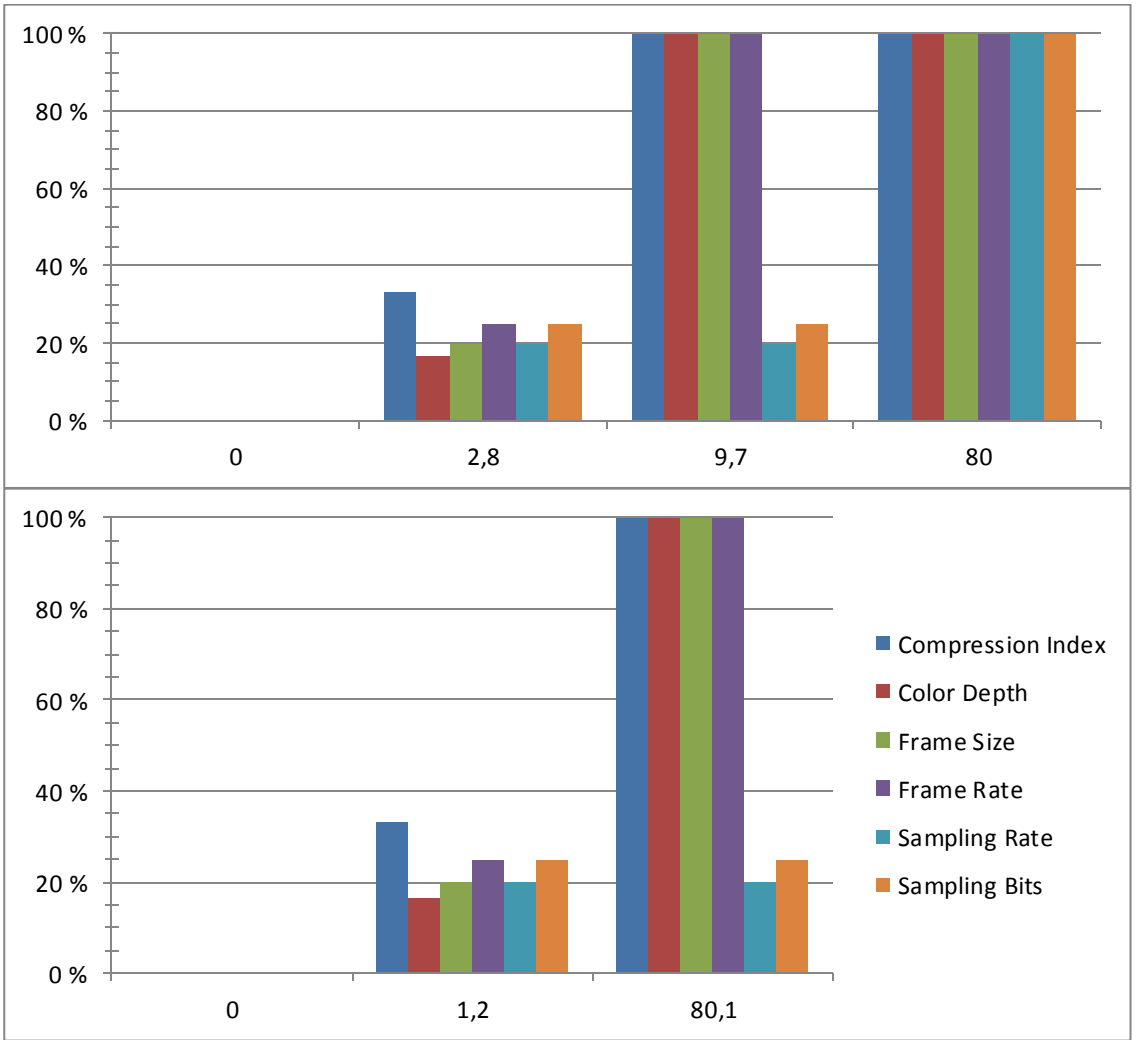


Figura 4.20 - Níveis de QoS nos componentes de *A* a *E*

A entrada de uma nova máquina origina quase sempre um novo SLA para cada componente e, como se pode verificar através da Figura 4.19 e da Figura 4.20, os níveis de QoS rapidamente se alteram imediatamente após a entrada de uma máquina. Quando os níveis não se alteram, significa que o componente convergiu. Ou seja, considerando o conjunto de máquinas que, naquele momento, compõem a coligação, o nível do componente atingiu o máximo que lhe pode ser fornecido. Por outras palavras, nenhuma outra máquina o consegue obter, pois o nível configurado no componente seria inferior ao atual. Considerando ainda as figuras, apenas as dimensões menos significativas sofrem alterações, e, neste caso, é possível manter a qualidade máxima de execução na dimensão de vídeo, à exceção do componente C que sofre algumas alterações nos dois atributos menos significativos da dimensão da qualidade de vídeo. Apenas aos 99 segundos, com a adesão da máquina *vmC* na coligação, é que é possível configurar este componente com a qualidade máxima em todas as suas dimensões.

O tempo de negociação de componentes é muito baixo pelo que não tem qualquer impacto significativo nas suas execuções, como por exemplo atrasos em efeito dominó. De facto, o tempo de negociação de componentes entre ARs e AEs não influencia a *deadline*, pois a execução do componente prossegue simultaneamente com a negociação, à exceção da negociação inicial em que os componentes negociados ainda não se encontram com o QoS configurado. Contudo, para efeitos de avaliação, considera-se que o serviço só se encontra ativo quando a totalidade dos seus componentes se encontra a executar, ou seja, após cada um dos componentes ser negociado com sucesso pelo menos uma vez.

Uma forma verosímil de verificar os resultados produzidos pela simulação é comparar os seus resultados com os de um cenário tradicional em que uma só máquina executa um serviço inteiro. Para isso foi criada uma aplicação que gera 5 *threads*, cada uma com um componente, simulando a execução do serviço. No final da sua execução, cada *thread* envia o seu *output* para as *threads* dos componentes sucessores. Para este teste foram utilizadas máquinas reais (*Laptop 1*, *Laptop 2* e *PC*) visto que os resultados produzidos por uma execução com máquinas virtuais são imensamente prejudicados relativamente a um cenário mais realista, composto apenas por máquinas físicas. Neste último, a performance obtida é mais confiável e autêntica.

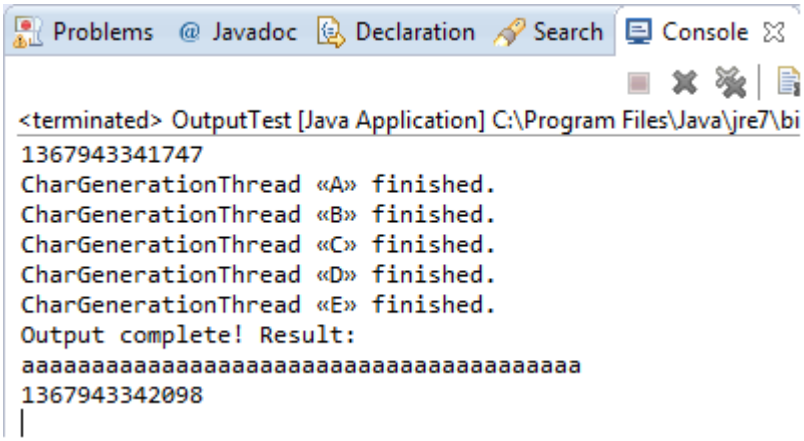


Figura 4.21 – Exemplo de *output* de cenário tradicional

Na Figura 4.21 o tempo total de *output* (duração) é de 351ms e a execução decorreu em *Laptop 1*. De destacar que na aplicação tradicional a sincronização das *threads* criadas foi feita através de *race conditions*, utilizando semáforos; por ordem, cada *thread* adicionava os carateres produzidos a uma variável concorrentemente acedida - o conjunto de carateres gerados pela aplicação. Convém também referir que não foram utilizados *logs* para ficheiros, apenas *outputs* para a consola da ferramenta Edipse. Verificou-se que o número total de carateres imprimidos após a execução foram 40: todos os componentes, de A a E, executavam no nível de QoS mínimo, produzindo cada um 8 carateres. De notar que nesta aplicação todos os componentes do serviço eram executados localmente e não era necessário proceder ao arranque de máquinas virtuais.

Relativamente aos resultados produzidos pela simulação do modelo, foi lançado um arranque de uma coligação inicialmente composta por uma só máquina: *Laptop 1*, o *localhost*. A esta juntar-se-iam nos segundos seguintes duas outras máquinas: *Laptop 2* e *PC*. Verificaram-se, posteriormente, alguns resultados como tempo total de execução do serviço, desde o primeiro ao último componente. Inicialmente os resultados não superam os do cenário tradicional, em termos de qualidade de *output*. Efetivamente, a qualidade produzida é equivalente à do cenário tradicional, dado que em ambos os cenários a *availability* do *localhost* (*Laptop 1*) é igual e, nesses instantes, a coligação é composta apenas por essa máquina. No entanto, após a entrada das restantes máquinas, a coligação produz melhores resultados e de uma forma mais eficiente, isto é, em menos tempo. A Tabela 4.4 mostra alguns resultados com uma coligação formada por 3 máquinas, produzidos numa única execução com múltiplas iterações. Nestes testes foi utilizada a configuração de QoS de geração de caracteres. A primeira coluna da tabela remete para aspetos qualitativos e a segunda para aspetos temporais.

Nº de caracteres	**Duração (ms)**
64	336
64	118
280	229
280	94
280	70
280	108
280	87
280	73
280	70

Tabela 4.4 - Resultados de *output* da simulação

Comparativamente com o cenário tradicional os resultados obtidos mostram que foram conseguidas reduções de aproximadamente 20% no tempo total de execução. Em termos de qualidade produzida os ganhos estão na ordem dos 700%. O nível de qualidade mais alto foi atingido da segunda para a terceira iteração, aos 454ms, após o serviço estar completamente operacional. Neste instante considera-se a coligação convergida.

Por último, foram realizados testes relativos ao tempo de recuperação de falhas nas réplicas. Com a ajuda do ficheiro de configuração, definiram-se *timeouts* de *heartbeats* para *t* ms. Através da definição deste parâmetro teoriza-se que a ocorrência de uma falha é detetada num espaço de pelo menos *t* ms. Na Tabela 4.5 existem dois tipos de agente: o faltoso, aquele que falha, e o perceptor, aquele que, através da troca de *heartbeats*, se apercebe da falha do parceiro. A injeção de falhas foi simulada através da pausa da atividade do agente.

Agente faltoso	**Agente perceptor**	***Timeout* definido (ms)**	***Timeout* real (ms)**	**Frequência de *heartbeats* definida (ms)**
EXECUTION2	EXECUTION2-clone	400	423	50
EXECUTION3	EXECUTION3-clone	400	404	50
EXECUTION2-clone	EXECUTION2	400	403	50
EXECUTION5	EXECUTION5-clone	200	203	20
EXECUTION1	EXECUTION1-clone	200	223	20
EXECUTION2	EXECUTION2-clone	200	203	20
EXECUTION5-clone	EXECUTION5	200	429	20
EXECUTION5	EXECUTION5-clone	600	774	20

Tabela 4.5 - Tempos de recuperação de *timeouts* de agentes

O intervalo de frequência de envio de *heartbeats* não deve ter um valor muito próximo do valor de *timeout*, pois isso pode provocar o surgimento de atrasos no envio de *heartbeats* e, consequentemente, conduzir à deteção de falhas que não existem; falsos-positivos.

A verificação do *timeout* é uma tarefa prioritária que não deve ser relegada para segundo plano no escalonamento das tarefas executadas pelos agentes. O instante em que uma falha é efetivamente detetada tem um ligeiro atraso relativamente ao valor do parâmetro definido, tal como era expectável. Isto é, o *timeout* definido é sempre menor do que o *timeout* real. O valor de *timeout* foi definido tendo em consideração esse atraso. No pior dos casos o *timeout* real tem uma diferença de 174ms relativamente ao *timeout* definido, enquanto no melhor dos casos a diferença é de apenas 3ms. Assim sendo, considera-se que a *framework* JADE prejudica o modelo nestes casos, na medida em que não permite gerir aspetos de escalonamento de *behaviours* nos agentes.

5 Conclusão

Este é o capítulo final onde são apresentadas conclusões acerca do trabalho desenvolvido. São também apresentadas as principais dificuldades encontradas ao longo da sua evolução e as limitações de que, por diversas razões, o modelo proposto sofre. Por último, são referidas algumas propostas para possíveis desenvolvimentos futuros ao modelo e à implementação.

5.1 Conclusão da dissertação

Este trabalho foi desenvolvido no âmbito de sistemas embebidos de tempo-real que se enquadram em ambientes distribuídos, dinâmicos e abertos.

A integração do modelo em sistemas multiagente favoreceu em grande escala o desenvolvimento deste trabalho pois trouxe vantagens em aspetos fulcrais como a descentralização e a heterogeneidade do ambiente - características que devem ser consideradas nos sistemas em questão. Para além disso, a utilização de agentes móveis possibilita o funcionamento do modelo numa arquitetura híbrida, isto é, seja por comunicação baseada em tempo (TTA) como em eventos (ETA). Esta metodologia dota o modelo com a capacidade de responder às diversas alterações que possam ocorrer no meio através da migração de componentes para máquinas mais apropriadas, a fim de se alcançar uma salubre execução do serviço. Relativamente a técnicas de tolerância a falhas, *e.g.* a promoção de réplicas, utiliza-se uma arquitetura TTA que beneficia o modelo em termos de reatividade. O esquema baseado no EDRB auxilia o modelo em aspetos como a descentralização, pois as réplicas sentem as ações que são realizadas no meio e que as afetam. As réplicas têm autonomia suficiente para assumirem papéis primários e secundários na execução do seu componente. Por outro lado, a replicação física e virtual colmata falhas do tipo aplicacional e também ao nível do *hardware*. A troca de *heartbeats* funciona como um sensor do meio que deteta eventuais falhas e permite reagir atempadamente ao meio faltoso. A coordenação *auction-bidding* do sistema, isto é, em formato de leilão, permite aos agentes

de reconhecimento formularem propostas que expressem de uma forma realística o nível que irá ser efetivamente fornecido ao componente negociado, através de uma descrição do tipo formal e uniformizada. No final das negociações por um componente, o melhor agente é selecionado para a sua execução. Os benefícios verificam-se ao longo da execução do serviço e, simultaneamente, da formação da coligação e refletem-se na melhoria incremental da qualidade produzida.

A nível global pode classificar-se a sincronização de *inputs* e *outputs* entre os agentes de execução como uma técnica de *communication-induced checkpointing*, na medida em que existe uma conformidade na atividade de um grupo de agentes.

Relativamente aos resultados finais obtidos, o modelo dotado de configurações dinâmicas faz face a um ambiente mudável e permite obter melhores resultados em menos tempo. A nível individual, cada componente sai favorecido pois existe um esforço no sentido de maximizar, dentro das possibilidades da coligação, o nível de QoS fornecido.

5.2 Limitações

Foram encontrados diversos impedimentos no desenvolvimento deste trabalho, em grande parte na implementação da simulação devido à *framework* utilizada. As principais dificuldades encontram-se enumeradas no parágrafo seguinte.

Considerando as características do ambiente de execução da JADE é necessário um esforço para sincronizar leituras e escritas às variáveis dos agentes, através da aplicação de *behaviours* do tipo FSM, num ambiente onde há concorrência de execução, apesar de cada agente executar numa única *thread* (à exceção dos *threaded behaviours*, em que um behaviour é lançado numa *thread* independente). A *framework* não gere aspetos de escalonamento, pelo que muitas vezes a criação dos clones sofre alguns atrasos. Não existe a possibilidade de definir prioridades entre *behaviours* pois o seu escalonamento é feito na *queue* de *behaviours* de cada agente, que é um processo feito internamente pela JADE. Na simulação não existe nenhuma preocupação em coordenar os agentes de reconhecimento da coligação de forma a tornar prioritária a execução de componentes com grau de significância maior; a sua execução dos componentes com um grau de significância maior deve preceder a execução de componentes com um grau de significância menor. Logo, os componentes com maior grau de significância devem ser iniciados em primeiro lugar.

5.3 Trabalho futuro

Primeiramente, pelo facto de haver atributos dependentes entre si, o modelo deve ser completado com a inserção de dependências de QoS entre atributos. Isto sucede porque existem cenários em que é inútil um componente executar a um determinado nível de

QoS se receber *inputs* com qualidade insuficiente. A utilização de dependências é detalhada em [17].

A limitação relativa à priorização da execução dos componentes com maior grau significância pode ser facilmente resolvida com a incorporação de uma fase intermediária no processo de negociação (ver Figura 4.10). Nessa nova etapa, que sucederia ao passo 2 (*Component informing*), após saber qual o componente a executar, um AR calcularia, através do objeto do serviço, o grau de significância desse mesmo componente. De seguida, trocaria informações com os ARs parceiros para saber qual o menor grau de significância do conjunto de componentes que já se encontrariam a executar dentro da coligação. Desta forma, saber-se-ia se se devia prosseguir a negociação, pelo respetivo componente, com o AE ou então se contactar a AMS para encontrar novos AEs com componentes com um grau de significância maior, isto é, mais significativos para o funcionamento do serviço.

Atualmente, nas redes de computadores, é frequente a existência de bastantes máquinas, *e.g.* dispositivos portáteis, servidores, dispositivos *lightweight*. No entanto, esse tipo de dispositivos podem não estar presentes, e, nesses casos, a coligação formada será prejudicada justamente pela carência de máquinas que, eventualmente, estariam disponíveis participar na execução distribuída. Portanto, uma boa forma continuar o desenvolvimento deste trabalho seria a expansão do modelo para ambientes *cloud-computing*. Assim, o número de máquinas aptas para auxiliar a execução do serviço seria extraordinariamente maior. Contudo, seria preciso preparar o modelo em outros aspetos como a segurança. Nesses termos, a JADE viabiliza o envio de ACL *messages* num procedimento PKI, dado que é possível a introdução de uma chave pública na mensagem. Esse procedimento confere maior segurança aos diálogos entre agentes pois são atestadas propriedades como a autenticidade, a integridade e a irretratabilidade.

Acredita-se que a integração do modelo de tolerância a falhas, desenvolvido neste trabalho, com uma rede GMPLS como a referida em [22] é um caminho assumptível para continuar o desenvolvimento de um modelo completo. Teoricamente, essa aglomeração, corresponde à junção de heurísticas aplicacionais com heurísticas ao nível da rede. Essa melhoria no modelo permitiria alcançar uma melhor performance, pois seria mais tolerante a falhas que ocorrem tipicamente em sistemas *real-time*. Por conseguinte, obter-se-iam resultados mais satisfatórios. Possivelmente os algoritmos teriam de ser ajustados com a adição de heurísticas de rede, *e.g. paths* que contivessem *links* com maior largura de banda teriam um peso maior no cálculo das *rewards*. Dever-se-ia considerar e averiguar o *tradeoff* entre a disponibilidade da máquina e a largura de banda do caminho de rede a ser adotado na transferência de dados.

Uma outra possibilidade, não exclusiva e facilmente conjugada com as outras hipóteses, seria a integração de computação evolucionária como algoritmos genéticos (DNA *algorithms*) aplicados à tolerância a falhas. Esta solução é ideal para resolver problemas *NP-complete*, frequentemente encontrados neste tipo de modelos. Contudo, a computação

baseada em DNA ainda está a evoluir e pode ser eventualmente prematura a aplicação desta técnica na área de tolerância a falhas em RTDS.

A aplicação do modelo proposto neste documento a um *software* é algo atingível, nomeadamente em meios com as características mencionadas. Assim sendo, futuramente o modelo pode ser integrado com diversas aplicações de tempo-real baseadas em componentes que permitam uma configuração de QoS em tempo de execução.

Referências

[1] M. Hiller, "Software fault-tolerance techniques from a real-time systems point of view," Chalmers University of Technology, Department of Computer Engineering, Goteborg, Sweden, 1998.

[2] J. Laprie, "Dependability: basic concepts and terminology," *Dependable Computing and Fault-tolerant Systems Series,* vol. 5, 1992.

[3] H. Kopetz, Real-time systems: design principles for distributed embedded applications, 2nd edition, Virginia, USA: Springer, 2011.

[4] J. Guerrero e G. Oliver, "Multi-robot coalition formation in real-time scenarios," *Robotics and Autonomous Systems,* vol. 60, pp. 1295-1307, 2012.

[5] T. Anderson e C. Hsiao, "Estimation of dynamic models with error components," *Journal of the American Statistical Association,* vol. 76, pp. 568-606, 1981.

[6] A. Avizienis e L. J., "Dependable computing: from concepts to design diversity," *Proceedings of the IEEE,* vol. 74, pp. 629-638, 1986.

[7] B. Gerkey e M. Mataric, "A formal analysis and taxonomy of task allocation in multi-robot systems," *International Journal of Robotics Research,* vol. 23, pp. 939-954, 2004.

[8] F. Bellifemine, G. Caire e D. Greenwood, Developing multi-agent systems with JADE, Italy: Wiley, 2007.

[9] M. Minsky, The society of mind, Simon & Schuster, 1986.

[10] P. Maes, "Artificial life meets entertainment: life like autonomous agents," *Communications of the ACM,* vol. 38, pp. 108-114, 1995.

[11] M. Becker, G. Signh, B. Wenning e C. Gorg, "On mobile agents for autonomous logistics: an analysis of mobile agents considering the fan out and sundry strategies," *International Journal of Services Operations and Informatics,* vol. 1, 2007.

[12] E. Durfee, "Multiagent systems: a modern approach to distributed artificial intelligence," *Distributed Problem Solving and Planning, MIT Press,* pp. 121-164, 1999.

[13] J. Mir, "Protocolos criptográficos para canales de comunicación anónimos y protección de itinerarios en agentes móviles," PhD thesis, 2004.

[14] A. Dubey, S. Nordstrom, T. Keskinpala, S. Neema, T. Bapty e G. Karsai, "Towards a verifiable real-time, autonomic, fault mitigation framework for large scale real-time systems," Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, USA, 2006.

[15] L. Khalouzadeh, N. Nematbacksh e K. Zamanifar, "A decentralized coalition formation algorithm among homogeneous agents," *Journal of Theoretical and Applied Information Technology ,* pp. 35-42, 2010.

[16] L. Nogueira, L. Pinho e J. Coelho, "Flexible and dynamic replication control for interdependent distributed real-time embedded systems," em *IFIP International Federation of Information Processing* , 2010.

[17] L. Nogueira e L. Pinho, "Time-bounded distributed qos-aware service configuration in heterogeneous cooperative environments," *J. Parallel Distrib. Comput.,* vol. 69, pp. 491-507, 2009.

[18] L. Nogueira, L. Pinho e J. Coelho, "A feedback-based decentralised coordination model for distribution open real-time systems," *The Journal of Systems and Software,* vol. 85, pp. 2145-2159, 2012.

[19] A. Kumar, R. Yamav, Ranvijay e J. A., "Fault tolerance in real time distributed system," *International Journal on Computer Science and Engineering,* vol. 3, pp. 933-939, 2011.

[20] Y. Gong, L. Yao, W. Zhang, J. Sha e C. Wang, "Research of coalition formation in time-bounded multiagent systems," em *Proceedings of the 1st International Conference on Machine Learning and Cybernetics*, Changsha, China, 2002.

[21] O. Shehory e S. Kraus, "Coalition formation among autonomous agents: strategies and

complexity," Ramat Gan, Israel.

[22] L. Valcarenghi, F. Cugini, F. Paolucci e P. Castoldi, "Quality-of-service-aware fault tolerance for grid-enabled applications," *Optical Switching and Networking,* vol. 5, pp. 150-158, 2008.

[23] E. Werner, "Towards a theory of communication and cooperation for multiagent planning," pp. 129-143.

[24] "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Anytime_algorithm. [Acedido em Fevereiro 2013].

[25] FIPA, "Foundation for Intelligent Physical Agents," [Online]. Available: www.fipa.org. [Acedido em Abril 2013].

[26] G. Caire e F. Pieri, LEAP user guide, Italy, 2011.

[27] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar e J. Hansen, "A scalable solution to the multi-resource qos problem," *Proceedings of the 20th IEEE Real-Time Systems Symposium,* pp. 315-326, 1999.

[28] P. Troger, "Dependable systems, Software dependability," 2010.

[29] A. Fernandes, "Agente de contexto para dispositivos móveis," Porto, 2009.

[30] R. T., R. S., J. N. e G. A., "An anytime algorithm for optimal coalition structure generation," *Journal of Artificial Intelligence Research,* vol. 34, pp. 521-567, 2009.

[31] K. Kim, "The distributed recovery block scheme," California, Irvine, 1995, pp. 191-209.

[32] Y. Gong, "CORBA application in real-time distributed embedded systems," *Real-time Systems Design,* 2003.

[33] K. Kim e H. Welch, "Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Transactions on Computers,* vol. 38, pp. 626-636, 1989.

[34] W. Yeugn e S. Schneider, "Design and verification of distributed recovery block with CSP".

[35] M. Klusch e A. Gerber, "Issue of dynamic coalition formation among rational agents," *German Research Centre for Artificial Intelligence, Multiagent Systems Group,* pp. 91-102.

[36] Patterson, "Computer systems architecture: reliability and performance".

[37] R. de Mello e L. Senger, "Modelo de migração baseado na avaliação da carga e tempo de vida de processos em ambientes distribuídos heterogêneos," *IEEE Latin America Transactions,* vol. 4, pp. 373-378, 2006.

[38] C. Wang e Z. Li, "Parametric analysis for adaptive computation offloading," *ACM,* pp. 119-130, 2004.

[39] A. Avinash, "Fault tolerant real time dynamic scheduling algorithm for heterogeneous distributed system," Orissa, India, 2007.

[40] P. Ahuja e V. Sharma, "A JADE implemented mobile agent based host platform security," *Computer Engineering and Intelligent Systems,* vol. 3, pp. 8-20, 2012.

[41] V. Barros, "Agentes móveis," Porto, 2008.

[42] D. de Souza, "Agentes inteligentes: uma abordagem prática com o java agent development framework".

[43] E. Lourenço, "Agentes móveis," 2006.

[44] F. Bellifemine, A. Poggi e G. Rimassa, "JADE - a FIPA-compliant agent framework," Torino, Italy.

[45] J. Bigus e J. Wiley, "Constructing intelligent agents using java," 2001.

[46] T. Lab, "JADE: java agent development framework".

[47] L. Jade Software Corportaion, "Environmental considerations for deploying JADE," 2011.

[48] F. Bellifemine, G. Caire e D. Greenwood, "Appendix A, command line options," em *Developing multi-agent systems with JADE*, Wiley, 2007, pp. 259-270.

[49] F. Bellifemine, G. Caire, T. Trucco e G. Rimassa, JADE programmer's guide, Italy, 2010.

[50] M. Dastani, "Multi-agent systems".

[51] M. Gawinecki, "Software agent computing, 1st laboratory activities," Warsaw University of

Technology, Poland.

[52] C. Ramos, "Sistemas baseados em agentes," DEI-ISEP, Porto, 2008.

[53] C. Ramos e A. Silva, "Sistemas baseados em agentes," DEI-ISEP, Porto, 2011.

[54] L. Vogel, "Vogella," 2009. [Online]. Available: http://www.vogella.com/articles/JavaPerformance/article.html. [Acedido em Março 2013].

[55] "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Component-based_software_engineering. [Acedido em Março 2013].

[56] A. Kshemkalyani e M. Singhal, "Distributed computing, principles, designs and systems," Cambridge University Press, 2008.

[57] W. Brendel, "StackOverflow," [Online]. Available: http://stackoverflow.com/questions/25552/using-java-to-get-os-level-system-information. [Acedido em Abril 2013].

[58] J. Jovanovic, "Slideshare," [Online]. Available: http://www.slideshare.net/tmtm99/jade-v. [Acedido em Dezembro 2012].

[59] K. Leung, "FTS Framework for JADE," [Online]. Available: http://www.cse.cuhk.edu.hk/~kwng/FTS.html. [Acedido em Janeiro 2013].

[60] D. Grimshaw, "Running multiple JADE platforms," [Online]. Available: http://jade.tilab.com/doc/tutorials/JADEAdmin/index.html. [Acedido em Novembro 2012].

[61] S. Raisamo, "JADE Containers, Local and Remote," [Online]. Available: http://www.cs.uta.fi/kurssit/AgO/harj/jade_harkat/modified_JadeContainerTutorial.html. [Acedido em Novembro 2012].

[62] S. Raisamo, "Using the HTTP MTP for Inter Platform Communication," [Online]. Available: http://www.cs.uta.fi/kurssit/AgO/harj/jade_harkat/doc/tutorials/JADEAdmin/HttpMtpTutorial.html. [Acedido em Novembro 2012].

[63] R. Kessler, J. Vaucher e A. Ncho, "FAQ, General questions about JADE," [Online]. Available: http://jade.tilab.com/community-faq.htm. [Acedido em Novembro 2012].

[64] FIPA, "Interaction protocol specifications," [Online]. Available: http://www.fipa.org/repository/ips.php3. [Acedido em Abril 2013].

[65] Wikipedia, "Computador de DNA," [Online]. Available: http://pt.wikipedia.org/wiki/Computador_de_DNA. [Acedido em Maio 2013].

[66] L. Adleman, "Molecular computation of solutions to combinatorial problems," Department of Computer Science and Institute of Molecular Medicine and Technology, University of Southern California, Los Angeles, California.